Otto von Guericke University Magdeburg
Faculty of Mathematics
Institute for Mathematical Optimization

Diploma Thesis

Polyhedral Description Conversion up to Symmetries

Author:

Thomas Rehn

November 8, 2010

Supervisor:

Prof. Dr. rer. nat. habil. Achill Schürmann

Institute for Mathematics
University Rostock
D-18051 Rostock

Co-Assessor:

Prof. Dr. rer. nat. habil. Volker Kaibel

Faculty of Mathematics
Institute for Mathematical Optimization
Otto von Guericke University Magdeburg
Postfach 4120, D-39016 Magdeburg

Rehn, Thomas:
Polyhedral Description Conversion up to Symmetries
Diploma Thesis, Otto von Guericke University Magdeburg, November 2010.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1. Motivation

Symmetric polyhedra have fascinated humans for a long time. Some of the most prominent are regular polytopes, which have been known in two and three dimensions since ancient times. Figure 1.1 shows the five regular three-dimensional polytopes, also known as Platonic solids.

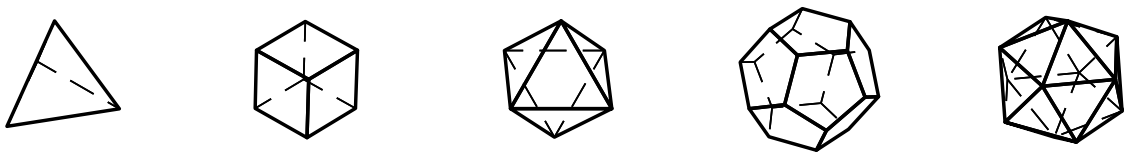

Figure 1.1.: The five Platonic solids: tetrahedron, cube, octahedron, dodecahedron, icosahedron

The study and classification of regular polytopes in general was revived by Schläfli and later Coxeter [Cox73]. His research about symmetry groups of regular polytopes led to new insights in algebra and other fields where so called Coxeter or reflection groups arise. A more general classification of symmetric polytopes was attempted by Robertson in [Rob84]. He succeeded in classifying all polytopes in low dimensions up to geometric symmetries. In his book he lists all possible classes of symmetries for polygons, which are polytopes in dimension two, and all polytopes in dimension three that have the same combinatorial structure as a cube.

With the rise of linear and integer optimization, the study of polyhedra got one more and also from a practical point of view important application. Significant progress in integer and combinatorial optimization in general has been made by exploring the structure of related polytopes (cf. [GLS93, Sch98]), a branch known as polyhedral combinatorics. These polytopes usually have two properties: they are highly symmetric and they suffer from "combinatorial explosion". This means that for small combinatorial problem size the related polytopes are readily tractable but very quickly grow beyond the realm of feasible computational methods. If symmetries of these polytopes can be found, bigger instances can be worked on up to symmetries, exploiting symmetries. In combinatorial optimization structural knowledge about even some parts of polyhedra may help to improve algorithm performance. For instance, the best solvers for the traveling salesman problem use partial knowledge of facets for their cutting plane approach (cf. [ABCC06]).

Besides optimization, symmetric polyhedra also occur as parts of many research problems from other areas of mathematics and science: The symmetry of a specially crafted

polytope was a key to a counterexample to the famous Hirsch conjecture by Santos [San10]. Symmetric polyhedra were also used by Kumar in algebraic geometry [Kum]. Some relevant polyhedra are already symmetric by construction, for instance, permutation polytopes as described in [BHNP09]. These are only some examples of polyhedra that the author is aware of for which symmetry was successfully exploited. They may motivate to look for symmetries also in other polytopes that are subject to research.

The rest of Chapter 1 formally introduces fundamental definitions of polyhedra and group theory. With the help of these we discuss a hierarchy of symmetry types that we may look for at polyhedra. In Chapter 2 we start with useful concepts and algorithms from computational group theory. We use these to analyze one well-known and one new approach to compute symmetries of polyhedra. At the beginning of Chapter 3 we look at well-known methods for general description conversion of polyhedra without using symmetries. On top of these we discuss different methods to exploit symmetries for description conversion by decomposition. Chapter 4 presents the author's C++ implementation `SymPol` of the described methods. We analyze the performance of the implementation and compare the discussed alternatives for computing symmetries and performing a description conversion on several different polyhedra.

Appendix A gives a proof of an open conjecture about the symmetries of permutation polytopes. The idea of this proof was motivated by experiments with `SymPol`. The manual of `SymPol` can be found in Appendix B.

1.2. Polyhedra

We continue this chapter with the necessary background on polytopes. We follow the notation of [Zie95], which the interested reader may also consult for more theory and details of polyhedra. First we look at two definitions for a polyhedron.

Definition 1.1 (H-polyhedron). Let $A \in \mathbb{R}^{m\times d}$ and $b \in \mathbb{R}^m$. We call the solution set $P(A,b) := \{x \in \mathbb{R}^d \ : \ Ax \leq z\}$ of the linear inequality system $Ax \leq b$ an **$\mathcal{H}$-polyhedron**.

For a second description of a polyhedron we need the notions of a convex and conical hull. For a finite set $V := \{v_1, \ldots, v_n\} \subset \mathbb{R}^d$ of points we define the **convex hull** of V as the set

$$\operatorname{conv} V := \Big\{ \sum_{i=1}^{n} \lambda_i v_i \ : \ \lambda_i \in \mathbb{R}, \lambda_i \geq 0, \sum_{i=1}^{n} \lambda_i = 1 \Big\}$$

and the **conical hull** (or positive hull) as the set

$$\operatorname{cone} V := \Big\{ \sum_{i=1}^{n} \lambda_i v_i \ : \ \lambda_i \in \mathbb{R}, \lambda_i \geq 0 \Big\}.$$

Later we will also need the **affine hull**, which is the set

$$\operatorname{aff} V := \Big\{ \sum_{i=1}^{n} \lambda_i v_i \ : \ \lambda_i \in \mathbb{R}, \sum_{i=1}^{n} \lambda_i = 1 \Big\}.$$

Definition 1.2 (V-polyhedron). Let $V := \{v_1, \ldots, v_n\} \subset \mathbb{R}^d$ and $R := \{r_1, \ldots, r_l\} \subset \mathbb{R}^d$. We call $P := \operatorname{conv} V + \operatorname{cone} R$ a $\mathcal{V}$**-polyhedron**.

By the Farkas-Minkowski-Weyl Theorem or main theorem for polyhedra (see e. g. [Sch98, Sec. 7.2] or [Zie95, Thm. 1.2]) the terms $\mathcal{H}$- and $\mathcal{V}$-polyhedron are equivalent. This means that for every $P := P(A, b)$ we can find finite sets V and R such that $P = \operatorname{conv} V + \operatorname{cone} R$ and vice versa. Because the terms are equivalent, we simply speak of a polyhedron. If a polyhedron is **bounded**, i. e. contained in a **ball** $B_M := \{x \in \mathbb{R}^d \ : \ \|x\| \leq M\}$ for some $M \in \mathbb{R}$, we call the polyhedron a **polytope**. A polytope P is thus always of the form $P = \operatorname{conv} V$ for a finite set V. If on the other hand a polyhedron $P = \operatorname{cone} R$ is a conical hull, we call it a (polyhedral) **cone**.

We denote by $\langle x, y\rangle := x^T y$ the standard scalar product of two vectors $x, y \in \mathbb{R}^d$. For a polytope $P \subseteq \mathbb{R}^d$ we call an inequality $\langle c, x\rangle \leq c_0$ **valid** if $\langle c, x\rangle \leq c_0$ holds for all $x \in P$. We call any set F of the form

$$F = P \cap \{x \in \mathbb{R}^d \ : \ \langle c, x\rangle = c_0\}$$

a **face** of P if $\langle c, x\rangle \leq c_0$ is valid for P. The dimension of a face F is the dimension of its affine hull: $\dim F := \dim(\operatorname{aff} F)$. We can see easily that P is always a face of P because $\langle 0, x\rangle \leq 0$ is a valid inequality for P. Similarly, $\emptyset$ is a face since $\langle 0, x\rangle \leq -1$ is a valid inequality.

Some faces have special names. We call the faces of

- dimension 0 **vertices**,
- dimension 1 **edges**,
- dimension $\dim(P) - 2$ **ridges**,
- dimension $\dim(P) - 1$ **facets**.

We can apply these terms also for the unbounded case of a polyhedron. Here we call unbounded edges (extreme) **rays**. Throughout this thesis we will usually identify a ray $r\mathbb{R}_0^+ = \operatorname{cone}\{r\} \subset \mathbb{R}^d$ with a representative $r_0 \in \operatorname{cone}\{r\}$. If a polyhedron P has vertices V and rays R we can decompose $P = \operatorname{conv} V + \operatorname{cone} R$ (cf. [Zie95, Thm. 2.15]).

We call a d-dimensional polytope P **simple** if every vertex lies in exactly d facets. We say a d-dimensional cone is simple if every ray lies in exactly $d - 1$ facets. Determining whether $P(A, b)$ is simple is NP-hard [Dye83, Prop. 1]. If a polytope is given as convex hull $P = \operatorname{conv} V$ then simplicity of P can be decided in polynomial time [BFM98, Thm. 4]. As we will see below, knowing whether a polytope P is simple or not may be useful to estimate the running time of algorithms on P.

We say a cone $P \subset \mathbb{R}^d$ is **pointed** if there is no $v \in \mathbb{R}^d$ such that the line $v\mathbb{R}^d$ is contained in P. An example for a pointed two-dimensional cone is $\{(x, y) \in \mathbb{R}^2 \ : \ x \leq 0, y \leq 0\}$. The half-space $\{(x, y) \in \mathbb{R}^2 \ : \ x \leq 0\}$ is not pointed because it contains the whole y-axis.

For a polyhedron $P = \operatorname{conv}\{v_1, \ldots, v_k\} + \operatorname{cone}\{r_1, \ldots, r_l\} \subseteq \mathbb{R}^d$ we define the **homogenization** $\operatorname{homog}(P)$ of P as

$$\operatorname{homog}(P) := \operatorname{cone}\left\{ \begin{pmatrix} 1 \\ v_1 \end{pmatrix}, \begin{pmatrix} 1 \\ v_2 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ v_k \end{pmatrix}, \begin{pmatrix} 0 \\ r_1 \end{pmatrix}, \begin{pmatrix} 0 \\ r_2 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ r_l \end{pmatrix} \right\} \subseteq \mathbb{R}^{d+1}.$$

We get $P \cong \text{homog}(P) \cap \{x \in \mathbb{R}^{d+1} \ : \ x_1 = 1\}$ as intersection of the homogenization with a hyperplane. In many situations it is easier to work with the homogenization of a polyhedron because it is a polyhedral cone with apex 0. By this we may treat polytopes, unbounded polyhedra and cones uniformly.

For every polytope $P \subseteq \mathbb{R}^d$ we can define its **polar** P^Δ as

$$P^\Delta := \{y \in \mathbb{R}^d \ : \ \langle y, x\rangle \leq 1 \text{ for all } x \in P\}.$$

We can iterate this construction and obtain the double-polar $P^{\Delta\Delta}$.

Theorem 1.3. Let $P = P(A, 1) = \text{conv}\, V$ be a polytope that has 0 as interior point. Then $P^\Delta = \text{conv}\, A = P(V, 1)$ and $P^{\Delta\Delta} = P$.

In this formulation we switch seamlessly between the interpretation of a set S of n vectors in dimension d (for the convex hull) and the $d \times n$ matrix S consisting of the elements of S as rows (for the inequalities). For a proof of Theorem 1.3 the reader may consider [Zie95, Thm. 2.11] or [Sch98, Thm. 9.1].

Let $P = P(A, b)$ be an $\mathcal{H}$-polytope. We call the task of computing a set V such that $P = \text{conv}\, V$ **vertex enumeration**. Given a $\mathcal{V}$-polytope $Q = \text{conv}\, W$, we name the task of computing facet inequalities B, c such that $Q = P(B, c)$ **facet enumeration**. More generally, we speak of a **description conversion** problem if we have to convert $\mathcal{H}$ to $\mathcal{V}$ or vice versa.

Because of the polarity theorem 1.3 it suffices to have an algorithm for either facet or vertex enumeration. For instance, suppose we have a vertex enumeration algorithm and a polytope $Q = \text{conv}\, V$. For the polar Q^Δ we formally obtain $Q^\Delta = P(V, 1)$. We can apply our vertex enumeration algorithm to Q^Δ and get a set B such that $Q^\Delta = \text{conv}\, B$. Polarizing again yields $Q = Q^{\Delta\Delta} = P(B, 1)$, the sought facet description of Q.

Although the $\mathcal{H}$- and $\mathcal{V}$-polyhedra are equivalent regarding their descriptive powers, these are different from a computational point of view. A first obvious point to note is that the sizes of $\mathcal{H}$- and $\mathcal{V}$-description may differ substantially. Let $C_d := [-1, 1]^d \subseteq \mathbb{R}^d$ be a d-cube. The cube C_d has $2d$ facets, so it can be described by only $2d$ inequalities. But C_d has 2^d vertices, which is exponential in the number of facets. On the other hand, a simplex $\text{conv}\{0, e_1, \ldots, e_d\} \subset \mathbb{R}^d$ has as many vertices as facets.

Because there may be a huge gap between the size of $\mathcal{H}$- and $\mathcal{V}$-description, complexity analysis of polyhedral description conversion usually looks at the time needed in terms of combined input and output size. However, all algorithms today still may need super-polynomial time (in input plus output size) to perform a description conversion of a general polytope (cf. [ABS97] and [FLM97]). More recently, [KBB+06] have shown that enumerating all vertices of an *unbounded* polyhedron is NP-hard. However, their result leaves the bounded case of a polytope and enumerating together vertices and rays of a polyhedron still open. For some special cases polynomial algorithms are known. For simple $\mathcal{H}$-polyhedra Avis and Fukuda give a polynomial time vertex enumeration algorithm (cf. [AF92, Avi00]). A polynomial time vertex enumeration algorithm for polytopes with vertices $v \in \{0, 1\}^d$, so called 0/1-polytopes, is known as well (cf. [BL98]). We will look more closely at description conversion algorithms in Section 3.1.

1.3. Groups and group actions

Before we can start with symmetries of polyhedra, we have to remember some concepts from group theory. We say a group G acts on a set Ω if there is a binary operation $G \times \Omega \to \Omega$ such that

- $x^e = x$ for the identity $e \in G$ and every $x \in \Omega$, and
- $x^{gh} = (x^g)^h$ for $g, h \in G$ and $x \in \Omega$.

Common examples for group actions are permutation and matrix groups.

Permutation groups consist of permutations of a finite set Ω, which we usually identify with the set of natural numbers $\{1, \dots, n\}$ for some $n \in \mathbb{N}$. We write S_n for the **symmetric group** of n elements, that is for the group of all permutations of n elements. In this thesis we will always use lowercase Greek letters to denote permutations. For a permutation σ we write $\sigma(i)$ for the image of $i \in \mathbb{N}$ under the action of σ. **Matrix groups** consist of matrices over a field and act by matrix-vector multiplication on the underlying vector space. In this thesis we usually work with the group of all invertible $d \times d$-matrices over the reals or over the rationals, denoted by $\mathrm{GL}(\mathbb{R}, d)$ and $\mathrm{GL}(\mathbb{Q}, d)$, respectively.

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 by $G \leq H$ 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.

The **orbit** of $x \in \Omega$ under G is the set of images $x^G := \{x^g : g \in G\}$. 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 $e_U \in U$.

The **stabilizer** $\mathrm{Stab}(G, x)$ of of $x \in \Omega$ in G is defined as the set $\mathrm{Stab}(G, x) := \{g \in G : x^g = x\}$ and forms a subgroup of G. Similarly, we write $\mathrm{Stab}(G, \Gamma) := \{g \in G : x^g \in \Gamma \text{ for all } x \in \Gamma\}$ for the setwise stabilizer of a set $\Gamma \subseteq \Omega$.

1.4. Polyhedra and symmetry

There are different kinds of symmetries we can look for at a polyhedron. We can distinguish four types: congruences, affine and projective symmetries, which have some kind of geometric realization, and combinatorial symmetries. Throughout this section we assume that the **vertex barycenter** $B(P)$ of our polyhedron P, i.e. $B(P) := \frac{1}{m} \sum_{i=1}^{m} v_i$ for all vertices $v_1, \dots, v_m$ of P, is the origin. This allows us to consider the linear symmetries, congruences and affine symmetries, without translation term.

The most accessible example for geometric symmetries are congruences. Let $P \subseteq \mathbb{R}^d$ be a polyhedron and $O \in \mathrm{GL}(\mathbb{R}, d)$ be an orthogonal transformation. We say that O is a **congruence** if O leaves P setwise invariant. Of course, the set of all congruences of P

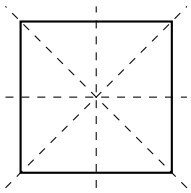

Figure 1.2.: Example of a polygon with maximal congruence symmetry group

constitute a matrix group. All congruences are well-known geometric transformations like reflections and rotations. The square in Figure 1.2 has eight congruences which are given by reflections at the diagonals and axes. The group that is generated by these eight congruences is generally known as the dihedral group D_4.

A generalization of congruences are affine symmetries. We say that $A \in \mathrm{GL}(\mathbb{R}, d)$ is an **affine symmetry** of a polyhedron P if it leaves P setwise invariant. The rectangle R

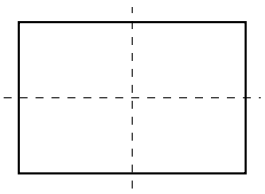

Figure 1.3.: Example of a polygon with maximal affine symmetry group

in Figure 1.3 has only four congruences generated by reflections. The 90 degree rotations about the center, which are congruences of the square, are not applicable to this rectangle. But we can combine a scaling by the matrix $A_s = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 2 \end{pmatrix}$ with a 90 degree rotation O_{90} to obtain an affine symmetry $A := O_{90}A_s$ of the figure, which is no congruence. Thus the group of affine symmetries of R still is the dihedral group D_4.

A further generalization are projective symmetries. A **projective symmetry** of a polyhedron P is a projective transformation that leaves P setwise invariant. We can identify this symmetry group with the affine symmetries of $\mathrm{homog}(P)$. The trapezoid T in Fig-

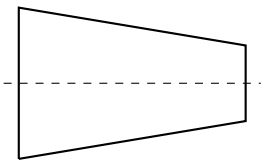

Figure 1.4.: Example of a polygon with maximal projective symmetry group

ure 1.4 has the dihedral group D_4 as projective symmetry group because $\mathrm{homog}(T)$ has the same affine symmetries as $\mathrm{homog}(R)$ where R is the rectangle from Figure 1.3. For a brief introduction to projective transformations the interested reader may also consider [Zie95, Sec. 2.6].

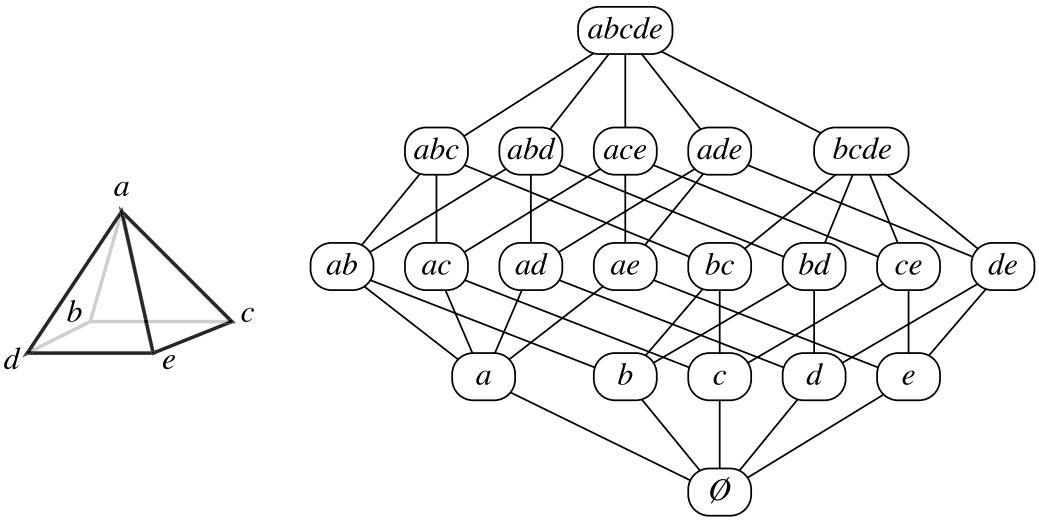

Figure 1.5.: Hasse diagram of a pyramid (source: `http://en.wikipedia.org/wiki/File:Pyramid_face_lattice.svg`)

We now turn to a different kind of symmetries: combinatorial symmetries. Before we can properly define them, we need more notation. The set of all faces of a polyhedron forms a partially ordered set by inclusion, a so called **poset**. Because the intersection of two faces of a polyhedron yields another face, the poset of faces forms a so called lattice and we speak of the **face lattice** $L(P)$ of P. [Zie95, Sec. 2.2] contains an introduction into poset terminology and further statements about the face lattice structure. The face lattice embodies the complete combinatorial structure of a polyhedron. An extension of the concept of a polyhedron are **abstract polyhedra**, which are algebraic objects inspired from face lattices of geometric polyhedra (cf. [MS02]). In this thesis we will only be concerned with geometric polyhedra.

A common visualization of a face lattice, or a poset in general, is a **Hasse diagram**. The Hasse diagram depicts a graph that has the elements of a poset as vertices. Two vertices x, y are connected by an edge if and only if $x \subsetneq y$. Figure 1.5 shows the face lattice of a three-dimensional pyramid. All vertices correspond to faces of the pyramid, and edges indicate setwise inclusion.

Based on the face lattice of a polyhedron we can define the most general form of its symmetry: combinatorial symmetry. A **combinatorial symmetry** of a polyhedron P is an automorphism f of its face lattice $L(P)$. Generally, we say that f is a face lattice isomorphism between two face lattices $L(P)$, $L(Q)$ if f is a bijection of the faces of P to the faces of Q such that for all faces F, G of P it holds that

$$F \subseteq G \quad \Longleftrightarrow \quad f(F) \subseteq f(G).$$

A face lattice isomorphism f between $L(P)$ and $L(P)$ is a face lattice automorphism. We write in short $\mathrm{Aut}\, P$ for the combinatorial automorphism group of a polyhedron P.

Note that the groups $\mathrm{Aut}\, P$ and $\mathrm{Aut}(\mathrm{homog}\, P)$ are isomorphic because P and $\mathrm{homog}\, P$ essentially have the same combinatorial structure.

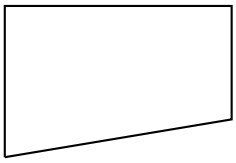

Figure 1.6.: Example of a polygon with maximal combinatorial symmetry group

The polygon P in Figure 1.6 continues our series of four-sided polygons. It has only trivial geometric symmetries as the identity is the only congruence and affine and projective symmetry. But its combinatorial symmetry group is isomorphic to the dihedral group D_4 as P has the same combinatorial structure as a square.

Every geometric symmetry is also a combinatorial symmetry because it permutes all faces of a polyhedron while preserving inclusion. The reverse does not hold as there are combinatorial symmetries which are not geometric as we have seen in Figure 1.6. In general, combinatorial symmetries are also more powerful than geometric symmetries in the following sense: [BEK84] give an example of a 4-dimensional polytope P with a combinatorial symmetry f which cannot be realized as affine transformation. That is, there is no polytope P' with the same face lattice as P such that f is induced by any orthogonal transformation on P'.

2. Computing Polyhedral Symmetries

We will see shortly that it is quite obvious how we can obtain combinatorial symmetries of P if vertices and facets of a polyhedron P are known. However, the task of description conversion implies that often only partial information is available. In this chapter we will thus look at methods to obtain combinatorial symmetries even if either vertices or facets of P but not both are available. Before we can discuss two different approaches for computing a subgroup of $\mathrm{Aut}(P)$, we have to look at some basics from the field of computational group theory.

2.1. Methods from computational group theory

Computational group theory provides the necessary means to work efficiently with mathematical groups algorithmically. In his thesis [Reh10] the author gave a brief overview of algorithms for permutation groups, which are at the heart of computations around any kind of symmetries. In this section we will briefly remind ourselves of some important concepts that we will use throughout this chapter to compute symmetries of polyhedra. We will also extend the scope of algorithms to treat both permutation and finite matrix groups. Besides the author's thesis, the interested reader may find all required computational aspects of permutation groups in [HEO05] and [Ser03].

2.1.1. Bases, strong generating sets and backtrack search

In this section we discuss a structure for representing a symmetry group so that we can treat it algorithmically. We always assume that G is a finite group acting faithfully on a set Ω. The usual case is to consider G as a permutation group with Ω being a set of numbers. But we also can and will use it for a (finite) matrix group $G \leq \mathrm{GL}(\mathbb{R}, d)$ where $\Omega = \mathbb{R}^d$ is the set of d-dimensional vectors, on which G acts naturally by matrix-vector multiplication.

Definition 2.1. Let G be a finite group acting on the set Ω. We call a sequence of elements $B := (\beta_1, \beta_2, \ldots, \beta_m) \subseteq \Omega$ 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**:

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

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 $G^{[i+1]}$. By Lagrange's Theorem, every $g \in G$ can uniquely be decomposed into

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

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 2.2. 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

$$G^{[i]} = \langle S \cap G^{[i]} \rangle, \quad \text{for } 1 \leq i \leq m+1. \tag{2.3}$$

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

A base together with a strong generating set enables us to compute the transversals along the stabilizer chain. By enumerating all possible transversal combinations

$$u_n u_{n-1} \cdots u_2 u_1 \quad \text{with } u_i \in U^{(i)}, \tag{2.4}$$

we enumerate all group elements according to (2.2). If we want to search for group elements with a specific mathematical property $\mathcal{P}$, we can list all elements and filter those satisfying $\mathcal{P}$. In the following we write $G(\mathcal{P})$ for all elements of G fulfilling property $\mathcal{P}$.

The decomposition based on (2.4) has the advantage that for many common problems we do not have to consider all combinations. Equation (2.4) is actually about base point images. The choice of u_1 fixes the image of the first base point, the choice of $u_2 u_1$ fixes the image of the first two base points, and so on until the complete image is known. For the problem of searching elements with some mathematical property $\mathcal{P}$, it is often possible to infer from a partial base image given by $u_i' u_{i-1}' \cdots u_1'$ that it cannot be extended to a group element satisfying $\mathcal{P}$. In this case we can thus skip all combinations that start with $u_i' u_{i-1}' \cdots u_1'$. This is the key to backtrack search in a group given by a BSGS. In Section 2.3.2 we will see an example where we search for symmetries of polyhedra.

2.1.2. 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 speed up the search for $G(\mathcal{P})$.

Definition 2.3. An **ordered partition** $\Pi = (\Pi_1, \ldots, \Pi_k)$ 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: $\Pi^g := (\Pi_1^g, \ldots, \Pi_k^g)$ 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 2.4. 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 2.5. 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

$$\mathcal{R}(\Pi)^g = \mathcal{R}(\Pi^g). \tag{2.5}$$

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 2.6. 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 2.7. 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 2.8. 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 $\Pi \wedge \mathrm{I}_5 = (5 \,|\, 1\,3 \,|\, 2\,4 \,|\, 6\,7)$.

For technical details of a partition backtrack search the interested reader may consider [Reh10] or [Leo91, Leo97]. A full description would go beyond the scope of this thesis so we briefly look at the idea of this algorithm. 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, $\Pi^g := (\Pi_1^g, \ldots, \Pi_k^g)$ 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 (2.5). 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. 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 $g \in G(\mathcal{P})$.

With this sketch of partition backtracking we conclude this excursion into computational group theory and return to the main task of computing symmetries of polyhedra.

2.2. Transforming the problem

It is well-known how to obtain combinatorial symmetries of a polyhedral cone if all rays and facets are known. Because every face of a cone is the conical hull of its rays and every face is the intersection of facets, every combinatorial symmetry is already determined by the relationship between rays and facets (cf. [KS03, p. 216]). We consider the bipartite graph $I(P)$ that has the rays and facets of P as nodes. A ray r and a facet F are joined by an edge in $I(P)$ if and only if $r \subseteq F$. We call $I(P)$ the incidence graph of P since it records the incidence between rays and facets. Every face lattice automorphism of P and thus every combinatorial symmetry of P corresponds to an automorphism of the graph $I(P)$ and vice versa. A graph automorphism f is a permutation of its node set V such that $n_1, n_2 \in V$ are joined by an edge if and only if $f(n_1), f(n_2) \in V$ are joined by an edge. Thus we can compute combinatorial automorphisms of polyhedra by computing automorphisms of graphs, which is a well-studied problem. It is still an open problem whether computing graph automorphism has polynomial complexity or not. Despite that, there exist fast implementations like [`nauty`] that are widely used in practice. [KS03] also show that computing combinatorial symmetries from the vertex/facet incidences is graph isomorphism complete, i. e. it is always "as hard" as computing isomorphisms of graphs.

Since our central problem in this thesis is description conversion of polyhedra, it may often be the case that we have only information about either the vertices or the facets. Although for some families of polytopes the full combinatorial automorphism group is

known (cf. [KW10], [Fio01]), this is not the general case. Thus we are interested in computing at least a subgroup of the general combinatorial symmetry group of a polyhedron when only partial information is available.

As the full combinatorial symmetry group is beyond our reach, we start by looking at geometrical symmetries of a polyhedron P. Because the symmetries of P are isomorphic to the symmetries of its homogenization $\mathrm{homog}(P)$, we will only consider the case of a polyhedral cone P. So let P be given by its rays $r_1, \ldots, r_n$. Then any affine symmetry $A \in \mathrm{GL}(\mathbb{R}, d)$ of P is characterized by

$$Ar_i = \lambda_i r_{\sigma(i)} \quad \text{for all } 1 \leq i \leq n \tag{2.6}$$

for a $\sigma \in S_n$ depending on A. For such an A we clearly have $A\,\mathrm{cone}\{r_1, \ldots, r_n\} = \mathrm{cone}\{Ar_1, \ldots, Ar_n\} = \mathrm{cone}\{r_1, \ldots, r_n\}$. Equivalently, we could also consider a cone P given as non-redundant $\mathcal{H}$-description $\{x \in \mathbb{R}^d \ : \ \langle c_i, x\rangle \leq 0 \text{ for } 1 \leq i \leq m\}$. Then an affine symmetry A of P is given by

$$c_i^T A = \lambda_i c_{\sigma(i)}^T \quad \text{for all } 1 \leq i \leq m \tag{2.7}$$

and A^T is an affine symmetry of the polar P^Δ. We see that the equations (2.6) and (2.7) are the same up to transposition for $\mathcal{V}$- and $\mathcal{H}$-description. Thus we can conclude that for computing affine symmetries the knowledge of either $\mathcal{V}$- or $\mathcal{H}$-description is sufficient, provided one can solve (2.6) or (2.7). This problem is still not suitable for our purpose as there seems to be no known algorithm for computing A satisfying this relationship (cf. [BDS09, Sec. 3.1]). An indicator for why this problem is hard is that the group of matrices A may be infinite. For instance, for the cone $P := \mathrm{cone}\{e_1, e_2\}$ every matrix $A_\lambda := \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}$ or $A'_\lambda := \begin{pmatrix} 0 & \lambda \\ \lambda & 0 \end{pmatrix}$ for every $\lambda > 0$ satisfy (2.6).

Therefore we look at a solution to a simplified version of (2.6). Like the authors of [BDS09] we call $A \in \mathrm{GL}(\mathbb{R}, d)$ a **restricted symmetry** of P if

$$Ar_i = r_{\sigma(i)} \quad \text{for all } 1 \leq i \leq n \tag{2.8}$$

for a $\sigma \in S_n$ depending on A. We write $\mathrm{RAut}(P)$ for the group of restricted symmetries of P. Note that if P is the homogenization of a polytope P' then $\mathrm{RAut}(P)$ is the group of affine symmetries of P' and (2.6) yields the projective symmetries of P'.

We also see that we can regard the group of restricted symmetries $\mathrm{RAut}(P)$ as matrix group and as permutation group. So far we have mostly been concerned with the matrix group representation. When we regard $\mathrm{RAut}(P)$ as a subgroup of the combinatorial symmetries $\mathrm{Aut}(P)$, any restricted symmetry A acts on the face lattice as well. By (2.8) we see how A permutes the rays of P. This is enough to determine how A acts on all other faces of P since every face is characterized by its incidence with the rays of P. We will switch transparently been the interpretations of a restricted symmetry as a matrix and a permutation of rays/facets, which induces a face lattice automorphism. The usage will always be clear from the context.

After these general remarks we now turn to the actual computation of restricted symmetries. From the definition in (2.8) we do not get any information about how the matrices A or the permutations σ look like. As an example for a restricted symmetry we consider the following triangle T of Figure 2.1 with vertices $o := (0,0)^T$, $a := (1,0)^T$ and $b := (0,2)^T$.

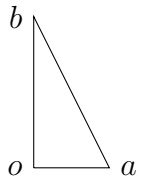

Figure 2.1.: Example for restricted symmetries

Then a simple calculation shows that the matrix $A := \begin{pmatrix} 0 & \frac{1}{2} \\ 2 & 0 \end{pmatrix}$ is a restricted symmetry of the triangle T. We see immediately that this transformation A is neither a reflection nor orthogonal. In general, angles may change under a restricted symmetry. To overcome this we can apply a transformation to the input.

In the following we look for restricted symmetries of the set $V := \{v_1, \ldots, v_n\} \subset \mathbb{R}^d$. This set V may represent the vertices or rays of a polyhedron or normal vectors of the facets of a polyhedron. A restricted symmetry for V is given by $Av_i = v_{\sigma(i)}$ for a $\sigma \in S_n$, as introduced before in (2.8). We assume that V is full-dimensional, i.e. $\dim(\operatorname{aff} V) = d$. We can always achieve this by a suitable projection. For V we consider the following matrix:

$$Q = \sum_{i=1}^{n} v_i v_i^T \in \mathbb{R}^{d \times d}. \tag{2.9}$$

We can easily see that this matrix is positive definite if V is full-dimensional. Hence we can compute the inverse Q^- and its Cholesky decomposition $R^T R := Q^-$. We apply R to the vectors in V and obtain vectors $w_i := Rv_i$, which make up the set $W := \{w_1, \ldots, w_n\}$.

Angles between the transformed vectors w_i will not change under a restricted symmetry $A \in \mathrm{GL}(\mathbb{R}, d)$ since

$$\begin{aligned} \langle w_i, w_j \rangle = w_i^T w_j = v_i^T R^T R v_j &= v_i^T Q^- v_j \\ &= v_i^T A^T Q^- A v_j \\ &= v_{\sigma(i)}^T Q^- v_{\sigma(j)} \\ &= \langle w_{\sigma(i)}, w_{\sigma(j)} \rangle. \end{aligned} \tag{2.10}$$

Here we have used the following observation or, more precisely, its inverse:

$$AQA^T = A \left(\sum_{i=1}^{n} v_i v_i^T \right) A^T = \sum_{i=1}^{n} A v_i v_i^T A^T = \sum_{i=1}^{n} v_{\sigma(i)} v_{\sigma(i)}^T = Q. \tag{2.11}$$

For our example above we compute $Q = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$, $Q^- = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{4} \end{pmatrix}$ and $R = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$. Transforming Figure 2.1 with R, we obtain an isosceles triangle as depicted in Figure 2.2.

The restricted symmetry A for the vectors V transforms into an orthogonal matrix $T := RAR^-$. For this matrix we have $Tw_i = w_{\sigma(i)}$ if $Av_i = v_{\sigma(i)}$. Note that we have the following identity for the transformed vectors w_i

$$\sum_{i=1}^{n} w_i w_i^T = \sum_{i=1}^{n} R v_i v_i^T R^T = RQR^T = \mathrm{Id} \tag{2.12}$$

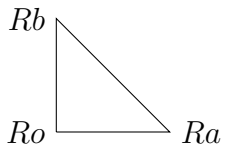

Figure 2.2.: Example for restricted symmetries (continued)

This means that the $w_1, \ldots, w_n$ mimic an orthonormal base for $\mathbb{R}^d$. In fact, [Had40, Satz 1] shows that the w_i are a projection of n orthonormal unit vectors $e_1, \ldots, e_n \in \mathbb{R}^n$ into the $\mathbb{R}^d$.

To actually compute restricted symmetries of a polyhedron we look at two ways to exploit equation (2.10). We already have seen that a restricted symmetry A, σ implies

$$\langle w_i, w_j \rangle = \langle w_{\sigma(i)}, w_{\sigma(j)} \rangle \tag{2.13}$$

for all i, j. Likewise the reverse is true. If (2.13) holds for a $\sigma \in S_n$ and all i, j then we can find a matrix $T \in \mathrm{GL}(\mathbb{R}, d)$ such that $Tw_i = w_{\sigma(i)}$ for all i.

To see this we choose an $\mathbb{R}^d$-basis from vectors in W. Without loss of generality we can assume that $w_1, \ldots, w_d$ constitute a basis. Then we write P for the matrix consisting of $w_1, \ldots, w_d$ as columns and P_σ for the matrix with $w_{\sigma(1)}, \ldots, w_{\sigma(d)}$ as columns. Because $w_1, \ldots, w_d$ make up a basis, the equation (2.13) is equivalent to $P^T P = P_\sigma^T P_\sigma$. This means that $T := P_\sigma P^-$ is an orthogonal matrix. For $1 \leq i \leq d$ we have $Tw_i = P_\sigma P^- w_i = P_\sigma e_i = w_{\sigma(i)}$. Thus for $1 \leq i \leq d$ and $1 \leq j \leq n$ it holds that

$$w_{\sigma(i)}^T T w_j = (Tw_i)^T T w_j = w_i^T w_j = w_{\sigma(i)}^T w_{\sigma(j)}.$$

So we have $w_{\sigma(i)}^T (Tw_j - w_{\sigma(j)}) = 0$. Because the $w_{\sigma(i)} = Tw_i$ form an $\mathbb{R}^d$-basis, this implies

$$Tw_j = w_{\sigma(j)} \quad \text{for all } 1 \leq j \leq n.$$

This shows that for finding restricted symmetries it is enough to search permutations $\sigma \in S_n$ such that $\langle w_i, w_j \rangle = \langle w_{\sigma(i)}, w_{\sigma(j)} \rangle$ for all i and j. In the following section we will discuss one solution to find these permutation directly and one new algorithm that approaches the problem from the matrix side.

2.3. Algorithms for computing restricted symmetries

2.3.1. Graph and matrix automorphisms

In [BDS09] the problem (2.13) is interpreted as a graph automorphism problem. The advantage of this approach is that this is a quite well-studied problem and there also exists software to compute graph automorphisms like [`nauty`]. So in order to obtain restricted symmetries as graph automorphisms we consider the edge-weighted complete graph $G(W)$ with vertex set W and edge weights $c(w_i, w_j) = \langle w_i, w_j \rangle$. Every automorphism of this graph $G(W)$ corresponds to a restricted symmetry of W. However, for implementations

it may be necessary to transform the edge-weighted into a vertex-weighted graph. For instance, `nauty` only handles vertex-weighted graphs. The `nauty` manual contains a description of how to transform a graph accordingly, requiring $|W|\lceil \log_2(c+1) \rceil$ vertices where c is the number of different edge weights in the graph.

In this section we follow a more direct path to solve this problem and avoid the preprocessing transformation. Instead of interpreting $\langle w_i, w_j \rangle = \langle w_{\sigma(i)}, w_{\sigma(j)} \rangle$ as a graph automorphism problem, we also can consider it directly as a matrix automorphism problem. We can state the problem we want to solve as follows. Given a symmetric matrix $A \in \mathbb{N}^{n \times n}$, find generators of the group $\mathrm{Aut}(A) := \{\sigma \in S_n \ : \ a_{ij} = a_{\sigma(i)\sigma(j)}\}$. Of course, this is still a graph automorphism problem in disguise but we will not need a further transformation of the input.

Solving this matrix automorphism problem can easily be fitted into a partition backtracking framework that we had a glimpse on in Section 2.1.2. To state again all details of partition backtracking would go beyond the scope of this thesis and the interested reader may consider the author's other thesis [Reh10]. Instead, we will only look at the fundamental structures that we need to solve our problem. That is, we need specific partition refinement procedures for the matrix automorphism problem. We do not have to start from scratch as Leon has already outlined in [Leo91] how such a refinement can look like.

According to Definition 2.5 from page 11, we discuss two $\mathcal{P}$-refinements that we can plug into a partition backtrack search to compute $\mathrm{Aut}(A) = G(\mathcal{P})$ as a subgroup of S_n. Let $\Omega = \{1, \ldots, n\}$ be the set of row or column indices. Without loss of generality we can assume that $A \in \{1, \ldots, v\}^{n \times n}$ for some $v \in \mathbb{N}$ such that for every $w \in \{1, \ldots, v\}$ there exists a matrix element a_{ij} with $a_{ij} = w$. First we remind ourselves how we can isolate elements or sets from a partition by intersecting with a special partition.

Definition 2.9. Let $\Gamma \subseteq \Omega$. Then we define the **isolating partition** $I(\Gamma) := (\Gamma \,|\, \Omega \setminus \Gamma)$.

With this notation at hand we can describe the first $\mathcal{P}$-refinement.

Lemma 2.10. Let $A \in \{1, \ldots, v\}^{n \times n}$ be a symmetric matrix and $G(\mathcal{P}) \leq \mathrm{Aut}(A)$ Then for every $w \in \{1, \ldots, v\}$

$$\mathcal{R}_{\mathrm{diag},w}(\Pi) := \Pi \wedge I(\{i \in \Omega \ : \ a_{ii} = w\}) \tag{2.14}$$

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

Proof. To show that $R_{\mathrm{diag},w}$ is a $\mathcal{P}$-refinement we have to prove that $\Pi^g \wedge I(\{i \in \Omega \ : \ a_{ii} = w\})^g = \Pi^g \wedge I(\{i \in \Omega \ : \ a_{ii} = w\})$ for every $g \in G(\mathcal{P})$. This is easy to see because these isolating partitions are g-invariant:

$$\begin{aligned} I(\{i \in \Omega \ : \ a_{ii} = w\})^g &= I(\{i^g \in \Omega \ : \ a_{ii} = w\}) = \\ &= I(\{i \in \Omega \ : \ a_{i^{g^-} i^{g^-}} = w\}) \\ &= I(\{i \in \Omega \ : \ a_{ii} = w\}) \end{aligned} \tag{2.15}$$

since $g^- \in \mathrm{Aut}(A)$. □

The second $\mathcal{P}$-refinement due to [Leo91] requires more notation.

Definition 2.11. Let $A \in \{1,\dots,v\}^{n\times n}$ be a symmetric matrix. Then we define the **fingerprint** function $f_A : \Omega \times 2^\Omega \to \mathbb{N}^v$ where 2^Ω denotes the power set of Ω. The value of $f_A(r,S)$ for $r \in \Omega$ and $S \subset \Omega$ is a v-dimensional vector z whose entries z_w are defined for every $w \in \{1,\dots,v\}$ as:

$$z_w := |\{i \in S \ : \ a_{ri} = w\}|. \tag{2.16}$$

In other words, we look at a subset of the r-th row of A, defined by the column set S, and count how many times each possible value $w \in \{1,\dots,v\}$ occurs.

With this fingerprint we can define the second $\mathcal{P}$-refinement.

Lemma 2.12. Let $A \in \{1,\dots,v\}^{n\times n}$ be a symmetric matrix and $G(\mathcal{P}) \le \mathrm{Aut}(A)$ Then for every $z \in \mathbb{N}^v$ and $1 \le k \le |\Pi|$

$$\mathcal{R}_{\text{fingerprint},z,k}(\Pi) := \Pi \wedge I(\{i \in \Omega \ : \ f_A(i,\Pi_k) = z\}) \tag{2.17}$$

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

Proof. As in the proof of the last lemma we show that the isolating partitions are g-invariant. This time we have to be careful because the isolating partitions depend on Π. Thus we have to show that $\Pi^g \wedge I(\{i \in \Omega \ : \ f_A(i,(\Pi_k)^g) = z\}) = \Pi^g \wedge I(\{i \in \Omega \ : \ f_A(i,\Pi_k) = z\})^g$ for $g \in G(\mathcal{P})$.

We first remember that the components of the fingerprint vector $f_A(r,S)$ count elements of the set $\{i \in S \ : \ a_{ri} = w\}$. Because

$$\begin{aligned} |\{i \in S \ : \ a_{ri} = w\}| &= |\{i^g \in S \ : \ a_{ri} = w\}| \\ &= |\{i^g \in S \ : \ a_{r^g i^g} = w\}|, \\ &= |\{i \in S^g \ : \ a_{r^g i} = w\}| \end{aligned} \tag{2.18}$$

it holds that $f_A(r,S) = f_A(r^g, S^g)$ for $g \in \mathrm{Aut}(A)$. Thus the equation

$$\begin{aligned} I(\{i \in \Omega \ : \ f_A(i,\Pi_k) = z\})^g &= I(\{i^g \in \Omega \ : \ f_A(i,\Pi_k) = z\}) = \\ &= I(\{i \in \Omega \ : \ f_A(i^{g^-},\Pi_k) = z\}) \\ &= I(\{i \in \Omega \ : \ f_A(i,(\Pi_k)^g) = z\}) \end{aligned} \tag{2.19}$$

holds and $\mathcal{R}_{\text{fingerprint},z,k}$ is a $\mathcal{P}$-refinement. □

The two $\mathcal{P}$-refinements from Lemma 2.10 and 2.12 are enough to compute $\mathrm{Aut}(A)$ in a partition backtracking framework. They have been implemented by the author in [`PermLib`] to be used by the polyhedral description software `SymPol`, which accompanies this thesis. In Section 4.2 we will see how this rather simple approach compares to the established graph-based solution using `nauty`.

2.3.2. Lattice automorphisms

For this section we assume that $V \subseteq \mathbb{Q}^d$, that means we just consider polyhedra with rational coefficients.

The matrix A of a restricted symmetry $Av_i = v_{\sigma(i)}$ is not unique. Every conjugate $A' := T^{-}AT$ for any $T \in \mathrm{GL}(\mathbb{R}, d)$ also satisfies $A'v_i = v_{\sigma(i)}$. Thus we can apply a linear transformation $T \in \mathrm{GL}(\mathbb{R}, d)$ to V without changing the restricted symmetries in their permutation representation. We choose d linear independent vectors from V, say $v_1, \ldots, v_d$. Then we set T as the inverse of the matrix with columns $v_1, \ldots, v_d$. For the transformed vectors $W := \{Tv \ : \ v \in V\}$ we obtain that $w_1 = e_1, \ldots, w_d = e_d$ where the e_i are the standard unit normal vectors in $\mathbb{R}^d$.

This implies that every restricted symmetry B of W, satisfying $Bw_i = w_{\sigma(i)}$, has $w_{\sigma(1)}, \ldots, w_{\sigma(d)}$ as columns. It is difficult to exploit this in our search for restricted symmetries. Choosing some columns of B yields no direct information whether this partial selection can be extended to a restricted symmetry. However, the equation $Bw_i = w_{\sigma(i)}$ yields restrictions on the rows of B. In the following we will discuss a backtracking algorithm that iteratively builds B row by row.

Suppose that $\{e_1, \ldots, e_d\} \subseteq W \subseteq \mathbb{Z}^d$. Then, of course, every vector in W is a member of the $\mathbb{Z}$-lattice $\mathbb{Z}^d$. In this case also every possible restricted symmetry B has to be an element of $\mathrm{GL}(\mathbb{Z}, d)$. This follows directly from $W \subseteq \mathbb{Z}^d$ and $|\det B| = 1$. We remember the necessary condition for the matrix B

$$BQB^T = Q, \tag{2.20}$$

which also is equation (2.11) from page 14 with $Q = \sum_i w_i w_i^T$. A different view on (2.20) is the following. Let $b_1, \ldots, b_d \in \mathbb{Z}^d$ be the columns of B^T. Naturally, these are also the rows of B but because we want to work with column vectors only we regard them as columns of B^T. Thus we are looking for integer vectors $b_1, \ldots, b_d$ such that $b_i^T Q b_j = e_i^T Q e_j = q_{ij}$ for all i, j where $e_1, \ldots, e_d$ are the unit normal vectors and q_{ij} are the entries of the matrix Q. This problem is called a **lattice automorphism problem**.

Plesken and Souvignier give in [PS97] a backtracking algorithm to compute lattice automorphisms. This algorithm computes

$$\mathrm{Aut}(\mathbb{Z}^d, Q) := \{B \in \mathrm{GL}(\mathbb{Z}, d) \ : \ BQB^T = Q\}. \tag{2.21}$$

So we obtain the restricted symmetries $\mathrm{RAut}(W)$ as the stabilizer of the set W in the lattice automorphism group $\mathrm{Aut}(\mathbb{Z}^d, Q)$. This gives us two ways to compute $\mathrm{RAut}(W)$. First, we can dedicatedly compute the stabilizer of $\mathrm{Aut}(\mathbb{Z}^d, Q)$ after we have computed $\mathrm{Aut}(\mathbb{Z}^d, Q)$ completely. Second, we can modify the algorithm of [PS97] to compute $\mathrm{RAut}(W)$ directly. In the following we discuss this second way and extend the original backtracking algorithm by Plesken and Souvignier. Our task is to find all matrices $B \in \mathrm{GL}(\mathbb{Z}, d)$ such that $Bw_i = w_{\sigma(i)}$. We try to accomplish this by iteratively computing columns $b_1, \ldots, b_d$ of B^T.

We already know that the columns of B, which are the rows of B^T, are some vectors $w_{\sigma(1)}, \ldots, w_{\sigma(d)}$. But because we have no information about the permutation σ, it is hard to obtain information about the product Bw_i, so we take another approach. Regardless of our knowledge of the rows of B^T, we know that not all vectors from $\mathbb{Z}^d$ are suitable

matrix columns. Looking again at (2.20), we can obviously restrict our search for columns to the set

$$S := \{b \in \mathbb{Z}^d \ : \ b^T Q b \leq \max_i q_{ii}\}. \tag{2.22}$$

We can compute S for instance with the Fincke-Pohst algorithm as described in [FP85]. Computing S may not be feasible for all matrices Q as the best known algorithms require exponential time in the worst case.

Because we will often switch between a list of vectors $a_1, \ldots, a_k$ and the matrix A which has $a_1, \ldots, a_k$ as columns, we write for short $A = \mathrm{mat}(a_1, \ldots, a_k)$. For our backtrack search we need criteria to decide early if a certain set of m possible columns $b_1, \ldots, b_m$ can be extended to $b_1, \ldots, b_d$ such that $B = \mathrm{mat}(b_1 \ldots b_d)^T$ is a restricted symmetry. In the following we discuss several necessary conditions for such an extension to be possible. These conditions fall into two categories. First those described in [PS97], which are generally applicable to lattice automorphism problems. Second those that are specific for the restricted symmetry problem of vectors sets.

Of the three criteria presented in [PS97] we will only describe one. The other two may only be useful for difficult cases and the interested reader will find them in [PS97, Sec. 5–6]. For $1 \leq m \leq d$ we call $(b_1, \ldots, b_m)$ an m-partial automorphism if $b_i^T Q, b_j = q_{ij}$ for all $1 \leq i, j \leq m$. As Plesken and Souvignier observe, the number of extensions from an m-partial to an $(m+1)$-partial automorphism is an invariant of all lattice automorphisms.

Additionally, we can use our condition $Bw_i = w_{\sigma(i)}$ for restricted symmetries to state another necessary condition. For every $w_i \in W$ we define $w_i' := B_m w_i \in \mathbb{Z}^m$ where $B_m^T := \mathrm{mat}(b_1, \ldots, b_m)$. Let $\Pi_m : \mathbb{Z}^d \to \mathbb{Z}^m$ be the projection to the first m coordinates. Then there has to exist a permutation $\sigma \in S_n$ such that

$$\Pi_m(w_{\sigma(i)}) = w_i' \tag{2.23}$$

for every $1 \leq i \leq n$. Thus, one necessary and easy to check condition for (2.23) is that the list $(w_i')_i$ has the same elements in the same quantity as $(\Pi_m(w_i))_i$. In the following we write $s_1 \sim s_2$ if two sequences $s_1, s_2 \subset \mathbb{Z}^m$ contain the same elements in the same quantity.

Putting these two criteria together leads to a backtrack algorithm to compute restricted symmetries. As preparation for this algorithm we compute two things. First, the sets

$$S_i = \{b \in S \ : \ b^T Q b = q_{ii} \text{ and } \exists \sigma \in S_n \, \langle b, w_j \rangle = \langle e_i, w_{\sigma(j)} \rangle \text{ for all } 1 \leq j \leq n\} \tag{2.24}$$

for $1 \leq i \leq d$. This is a restriction on the possible matrix columns based on a relaxed version of (2.23). Every i-th matrix column has to be an element of S_i. We can test membership in S_i easily by comparing the elements of the sequences $(\langle b, w_j \rangle)_j$ and $(\langle e_i, w_j \rangle)_j$.

As a second step we compute the number of extensions of i- to $(i+1)$-partial automorphisms for every $1 \leq i \leq d$. While doing so, Plesken and Souvignier suggest to find a better ordering of the base vectors $e_1, \ldots, e_d$. We begin with computing $f_{1i} := |S_i|$. We choose as first base vector e_{j_1} an index j_1 such that $f_{1j_1} = \min_i f_{1i}$. This ensures the number of choices at the first backtracking level is minimal.

Given reordered base vectors $e_{j_1}, \ldots, e_{j_l}$, we compute j_{l+1} as follows. We set $f_{l+1,i} := 0$ for $i \in \{j_1, \ldots, j_l\}$ and compute for all other values of i

$$f_{l+1,i} := |\{b \in S_i \ : \ b^T Q e_{j_k} = q_{ij_k} \text{ for all } 1 \leq k \leq l\}|. \tag{2.25}$$

Again we choose j_{l+1} such that $f_{l+1,j_l} = \min_i f_{l+1,i}$ among all $f_{l+1,i} > 0$. This ensures that we minimize our backtrack choices based on our previous selection of base vectors. When we finish this process, we obtain a probably improved ordering $j_1, \ldots, j_d$. In the following we assume that $j_i = i$ for all i and set $f_i := f_{ii}$. The values f_i give the number of possible extensions of an i-partial automorphism under the new base ordering. We now formulate a first version of our backtracking algorithm.

Input: Set of vectors $W = \{w_1, \ldots, w_n\} \subset \mathbb{Z}^d$, recursion level i, selected rows $B = \{b_1, \ldots, b_{i-1}\}$, candidate rows C for level $i+1$
Output: List $\mathcal{B}$ of matrices B such that $\mathcal{B} = \mathrm{RAut}(W)$

if $i = d+1$ **then**
  return $\{\mathrm{mat}(B)\}$
end
$\mathcal{B} \leftarrow \emptyset$
forall $b \in C \setminus B$ **do**
  $B' \leftarrow B \cup \{b\}$
  $C' \leftarrow \emptyset$
  if $i < d$ **then**
    `// check whether` B' `can be extended to an` $(i+1)$`-partial automorphism`
    $C_0 \leftarrow \{b' \in S_{i+1} \ : \ b'^T Q b = q_{i+1,i}$ and $b'^T Q b_j = q_{i+1,j}$ for all $1 \le j < i\}$
    if $|C_0| \neq f_{i+1}$ **then**
      next
    end
    $s_0 \leftarrow ((\Pi_{i+1}(w) \ : \ w \in W))$
    forall $b' \in C_0$ **do**
      $s_1 \leftarrow ((\mathrm{mat}(B' \cup \{b'\}) \cdot w \ : \ w \in W))$
      if $s_0 \sim s_1$ **then**
        $C' \leftarrow C' \cup \{b'\}$
      end
    end
  end
  $\mathcal{B} \leftarrow \mathcal{B} \cup$ `Backtrack`$(i+1, B', C')$
end
return $\mathcal{B}$

Algorithm 2.1: First version of backtrack search for restricted symmetries

Algorithm 2.1 will find all integer restricted symmetries of a set W. It expects as input the matrix $Q = \sum_i w_i w_i^T$ and the candidate set of short vectors S_i according to (2.24). Besides these, the algorithm, which is recursively referred to as `Backtrack`, needs the recursion level i, a set of already selected rows B and candidates for the i-th matrix row. We may start `Backtrack` with $i = 1$, $B = \emptyset$, $C = S_1$. Because Algorithm 2.1 computes all restricted symmetries and not only generators of $\mathrm{RAut}(W)$, it is only of theoretical interest. In practice, we only require a generating set of the restricted symmetries as matrix or as permutation representation. To improve Algorithm 2.1 we review it with Section 2.1.1 in mind.

In the group of all integer invertible matrices $G := \mathrm{GL}(\mathbb{Z}, d)$, we are looking for generators of the subgroup $H := \{B \in G \ : \ \exists \sigma \in S_n \ : \ Bw_i = w_{\sigma(i)} \text{ for all } i\} = \mathrm{RAut}(W)$. The unit normal vectors $e_1, \ldots, e_d$ are a base of $\mathrm{GL}(\mathbb{Z}, d)$ and thus of H as well (in the sense of Definition 2.1 from page 9). This is because for $I_d := \mathrm{mat}(e_1, \ldots, e_d)$ we have $BI_d = I_d$ if and only if $B = I_d$.

What we have done so far in Algorithm 2.1 is an implicit backtrack search in $\mathrm{GL}(\mathbb{Z}, d)$ with base $e_1, \ldots, e_d$. At each recursion level i we choose the image of the i-th base point e_i. Thus we can use techniques from general subgroup search in groups with bases. Note that in our case we do not need to know a strong generating set for G because we explicitly know a subset of G that we search in. That is, we restrict our search to matrices where the i-th column is a member of the set S_i. From the possible improvements of our search explained in [Reh10, Ser03] we discuss one simple enhancement, given by the following lemma.

Lemma 2.13. Let G be a group with base $\beta_1, \ldots, \beta_m$. Let $G^{[i]}$ be the pointwise stabilizer of $\beta_1, \ldots, \beta_{i-1}$ in G. Let H be a subgroup of G and suppose that for some i all elements of $K := G^{[i]} \cap H$ are known. Then for every $h, h' \in H$ 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 $u \in G^{[i]} \cap H$ such that $h = uh'$. By definition of K we have $u \in K$ and thus h and h' are in the same coset $Kh = Kh'$. □

This suggests the following improvement. At each recursion level i we choose as first candidate vector e_i, which clearly is in S_i. In this manner we find all elements of $G^{[i]} \cap H$ before $G \setminus G^{[i]}$ for every i. Suppose for some i we know $K := G^{[i]} \cap H$ because we have considered all extensions of $e_1, \ldots, e_{i-1}$. When we find an element $h \in H$ during our backtrack search as extension of $e_1, \ldots, e_{i-2}, b_{i-1}$ for some b_{i-1}, then we can skip all other extensions of $e_1, \ldots, e_{i-2}, b_{i-1}$ and proceed directly to the next $e_1, \ldots, e_{i-2}, b'_{i-1}$. This is because Lemma 2.13 states that every second extension of $e_1, \ldots, e_{i-2}, b_{i-1}$ in H can already be generated from $\langle K, h\rangle$.

Algorithm 2.2 is an improvement over Algorithm 2.1 as it computes only a generating set for the group of restricted symmetries. We add a global variable $i_{\text{completed}}$ shared among all recursion levels which contains the least i such that $G^{[i]} \cap H$ is known. Whenever we find a new group generator $\mathrm{mat}(B)$, we jump back straight to this recursion level.

Computing $G^{[i]} \cap H$ before $G \setminus G^{[i]}$ also has the advantage that our generating set of H will automatically be a strong generating set relative to the base $e_1, \ldots, e_d$ because it contains generators for all $H^{[i]}$. When we consider the permutation group representation $H' \leq S_n$ of H, we automatically obtain a strong generating set for it as well. The permutation group H' has a base consisting of the points $1, 2, \ldots, d$.

In general, the running time of this backtracking algorithm very much depends on the cardinality of the set of short lattice vectors S or, more precisely, the corresponding S_i. This is because $|S_1||S_2| \cdots |S_d|$ is an upper bound on the order of $\mathrm{RAut}(W)$ and the number of possible elements constructed in the backtrack search. Thus one may decide in advance based on the size of S whether a lattice backtrack search for restricted symmetries seems feasible.

Input: Set of vectors $W = \{w_1, \ldots, w_n\} \subset \mathbb{Z}^d$, recursion level i, selected rows $B = \{b_1, \ldots, b_{i-1}\}$, candidate rows C for level $i+1$
Output: List $\mathcal{B}$ of matrices B such that every $\langle \mathcal{B} \rangle = \mathrm{RAut}(W)$

```
1  if i = d + 1 then
2  |  return {mat(B)}, i_completed
3  end
4  B ← ∅
5  forall b ∈ C \ B do
6  |  B' ← B ∪ {b}
7  |  C' ← ∅
8  |  if i < d then
   |  |  // check whether B' can be extended to an (i+1)-partial
   |  |     automorphism
9  |  |  C_0 ← {b' ∈ S_{i+1} : b'^T Q b = q_{i+1,i} and b'^T Q b_j = q_{i+1,j} for all 1 ≤ j < i}
10 |  |  if |C_0| ≠ f_{i+1} then
11 |  |  |  next
12 |  |  end
13 |  |  s_0 ← ((Π_{i+1}(w) : w ∈ W))
14 |  |  forall b' ∈ C_0 do
15 |  |  |  s_1 ← ((mat(B' ∪ {b'}) · w : w ∈ W))
16 |  |  |  if s_0 ∼ s_1 then
17 |  |  |  |  C' ← C' ∪ {b'}
18 |  |  |  end
19 |  |  end
20 |  end
21 |  T, j ← Backtrack(i + 1, B', C')
22 |  B ← B ∪ T
23 |  if j < i then
24 |  |  return B, j
25 |  end
26 end
27 i_completed ← min{i_completed, i}
28 return B, i
```

Algorithm 2.2: Improved version of backtrack search for restricted symmetries

For a very large set of low-dimensional vectors, i.e. $n \gg d$, performing the sequence image checks in lines *13* to *19* may be quite expensive. The key elements are computing s_1 for each b' and comparing the two sequences, which amounts to a sorting operation on s_0 and s_1. This means there are at least n scalar products of d-dimensional vectors to be evaluated, plus an $O(n \log n)$ sorting effort in line *16*. In a trade-off between speed per node and total number of nodes visited during the backtrack search we may thus perform this check only for $i = d + 1$, for which we must, and for small i.

One last thing that we have to look at is our assumption that $\{e_1, \ldots, e_d\} \subseteq W$ is integer. We always can achieve $\{e_1, \ldots, e_d\} \subseteq W$, but then W needs not to be integer in general. In this non-integer case we may not obtain all restricted symmetries by solving

(2.20) for integer matrices A. Reviewing the algorithm, we note that we require W to be integer for only one thing: to fully enumerate all short lattice vectors $S = \{b \in \mathbb{Z}^d \ : \ b^T Q b \leq \max_i q_{ii}\}$ in (2.22). Let $\lambda \in \mathbb{Z}$ be the smallest integer such that $\lambda W \subseteq \mathbb{Z}^d$. Then we have to compute all vectors

$$\begin{aligned} S &= \{b \in \frac{1}{\lambda}\mathbb{Z}^d \ : \ b^T Q b \leq \max_i q_{ii}\} \\ &= \frac{1}{\lambda}\{b \in \mathbb{Z}^d \ : \ b^T Q b \leq \max_i \lambda^2 q_{ii}\} \end{aligned} \tag{2.26}$$

instead. This extended short lattice vector problem may be impractical to solve for $\lambda > 1$ or cause a very large set S.

2.3.3. Comparison

At the end of this chapter we briefly compare the two algorithms for computing restricted symmetries. As both algorithms are backtracking algorithms, we focus on the preprocessing steps and upper bounds on the number N of elements checked during the backtrack search. Let $V = \{v_1, \ldots, v_n\} \subseteq \mathbb{R}^d$ be the vectors we want to compute $\mathrm{RAut}(V)$ for.

The graph or matrix automorphism approach requires the computation and inversion of the $d \times d$-matrix $Q = \sum v_i v_i^T$ as a preprocessing step. Then the automorphisms of a symmetric $n \times n$-matrix A, or equivalently the automorphisms of a vertex-labeled graph with at least n vertices, are computed with a partition backtracking approach. Since the image of d base vectors determines the image of the remaining $n-d$ we obtain an obvious upper bound for the backtrack search by $N \leq n(n-1)\cdots(n-d+1)(n-d)$. In most cases this bound can be improved by looking at the specific R-base induced by the matrix A (cf. [Reh10, Sec. 3.2.3]).

As mentioned before, the performance of the lattice automorphism approach is determined by two things. First, a set of short lattice vectors $S = \{b \in \mathbb{Z}^d \ : \ b^T Q b \leq \max_i q_{ii}\}$ has to be computed. This may require exponential time depending on d and $\max_i q_{ii} \geq n$. If we do not know a-priori whether V admits an integer lattice representation, we have to invert a $d \times d$-matrix to get an answer to this question. Second, a backtrack search based on S is conducted. Clearly, we have $N \leq |S_1||S_2|\cdots|S_d| \leq |S|^d$ where $S_i \subseteq S$ is the set of vectors suitable for the i-th column of the lattice automorphism matrix. We can obtain a better estimation once we have computed the fingerprint matrix according to (2.25). This yields the bound $N \leq f_1 f_2 \cdots f_d$.

Thus a possible advantage of the second approach is that, if we briefly ignore S, it works with a $d \times d$ instead of a $n \times n$ matrix. If $n = |V|$ is much larger than d, then the graph in the first approach may contain too many vertices to be tractable in practice. In this case we may hope for a relatively small set of short vectors S to compute $\mathrm{RAut}(V)$ by lattice automorphisms. We will see in Section 4.2 one particular problem class for which these conditions hold and the lattice automorphism approach is clearly superior.

3. Description Conversion

3.1. Classic methods

In this section we will look at algorithms for the description conversion problem. As we have already seen in the discussion after Theorem 1.3 on page 4, the facet enumeration and vertex enumeration problem are polar to each other. Thus we can concentrate on one of these two and in the following we will be concerned only with vertex enumeration. This means, we are given a polytope $P = P(A, b)$ and want to find a set of vertices V such that $P = \operatorname{conv} V$. Sometimes it will be more convenient to work with the homogenized version: given a cone $P = P(A, 0)$, find a set of rays R such that $P = \operatorname{cone} R$.

In his overview [Sei04], Seidel identifies three categories into which almost all the available algorithms for the vertex enumeration problem can be put. In this section we will briefly discuss representatives of two of the three categories: incremental and graph traversal methods.

3.1.1. Incremental methods

Let $A \in \mathbb{R}^{n\times d}$ be a matrix with rows $a_1, \ldots, a_n$. Then for a set $K \subseteq \{1, \ldots, n\}$ we denote with A_K the sub-matrix of A that consists of the rows indexed by K. In order to perform a description conversion of a polytope $P = P(A, b)$, incremental methods iteratively construct a $\mathcal{V}$-representation for polytopes $P(A_{K_1}, b_1) \supset P(A_{K_2}, b_2) \supset \cdots \supset P(A_{K_l}, b_l) = P(A, b) = P$ where $K_1 \subset K_2 \subset \cdots \subset K_l = \{1, \ldots, n\}$ is an ascending chain of index sets. One of the easiest to describe incremental algorithms is the **double description method**. This algorithm was first described in [MRTT53] and has modern implementations, for example [`cdd`]. In the following, we stick to the notation of [FP96], which is the basis for `cdd`.

Let $A \in \mathbb{R}^{n\times d}$ and $R \in \mathbb{R}^{d\times m}$ be two matrices. We call the pair (A, R) a **double description pair** if $P(A, 0) = \operatorname{cone} R$. Here we interpret again $\operatorname{cone} R$ as the conical hull of the columns of R. In this context, a vertex enumeration problem for a cone $P = P(A, 0)$ is to determine a double description pair (A, R) where R is an unknown matrix to be computed. As the double description method is an incremental method, we will use a pair (A_K, R) to compute (A_{K+i}, R') where we write A_{K+i} short for $A_{K\cup\{i\}}$. Before we look at this iterative process we examine how to find a first double description pair to start from.

Lemma 3.1. Let $P = P(A, 0)$ be a d-dimensional polyhedral cone and A be matrix $n \times d$ matrix. Select a $d \times d$ sub-matrix A_K of A which consists of linearly independent rows (indexed by a set K). Then $(A_K, -A_K^{-})$ is a double description pair.

Proof. For a $\lambda \geq 0, \lambda \in \mathbb{R}^d$ we have

$$x \in P(A_K, 0) \;\Leftrightarrow\; A_K x \leq 0 \;\Leftrightarrow\; A_K x = -\lambda \;\Leftrightarrow\; x = -A_K^- \lambda \;\Leftrightarrow\; x \in \operatorname{cone}(-A_K^-).$$

□

Thus we know how to compute a starting point for the double description method and will now look at the iteration step.

We start with a double description pair (A_K, R). When we add a matrix row with index i to A_K, we intersect $P(A_K, 0) = \operatorname{cone} R$ with a half-space $\{x \in \mathbb{R}^d \;:\; a_i x \leq 0\}$. The corresponding hyperplane partitions the space into three parts:

$$\begin{aligned} H_i^- &= \{x \in \mathbb{R}^d \;:\; a_i x < 0\} \\ H_i^0 &= \{x \in \mathbb{R}^d \;:\; a_i x = 0\} \\ H_i^+ &= \{x \in \mathbb{R}^d \;:\; a_i x > 0\}. \end{aligned}$$

Let J be the set of column indices of R. Then also the rays, the matrix columns r_j with $j \in J$, are partitioned into three parts:

$$\begin{aligned} J^- &= \{j \in J \;:\; r_j \in H_i^-\} \\ J^0 &= \{j \in J \;:\; r_j \in H_i^0\} \\ J^+ &= \{j \in J \;:\; r_j \in H_i^+\} \\ &. \end{aligned}$$

For a new matrix R' that is to form a double description pair (A_{K+i}, R') we have to do two things. Obviously, we can discard the rays from R with indices in J^+ because they do not lie in $P(A_{K+i}, 0)$. But we can use these to compute new rays that lie on the new hyperplane H_i^0. The way to do this is the dual of the so called Fourier-Motzkin elimination. For more information about the Fourier-Motzkin elimination the interested reader may consult [Sch98, Sec. 12.2], which contains a description of the Fourier-Motzkin elimination with historical pointers, and [Zie95, Sec 1.2,1.3], which present both the Fourier-Motzkin elimination method and its dual. This gives us the following central lemma which is Lemma 3 from [FP96].

Lemma 3.2 (Main Lemma for Double Description Method). Let (A_K, R) be a double description pair and let i be a row index of A not in K. Then the pair (A_{K+i}, R') is a double description pair, where R' is the $d \times |J'|$ matrix with column vectors r'_j $(j \in J')$ defined by

$$\begin{aligned} J' &= J^- \cup J^0 \cup (J^+ \times J^-), \text{ and} \\ r'_j &= r_j \text{ for } j \in J^- \cup J^0, \\ r'_{jj'} &= (A_i r_j) r_{j'} - (A_i r_{j'}) r_j \text{ for each } (j, j') \in J^+ \times J^-. \end{aligned}$$

Proof. [FP96, p. 96] □

So with the help of Lemma 3.2 and Lemma 3.1 we can compute a double description pair (A, R) and thus solve the vertex enumeration problem, at least in theory.

In practice, however, the number of rays quickly runs beyond all tractable limits because many of the constructed $r'_{jj'}$ rays are redundant. [FP96] give several improvements, one of

which is how to compute in each step a minimal matrix R. However, [Bre99] showed that the running-time of all incremental algorithms is super-polynomial in input and output size in the worst case. Despite this bad worst case behavior the double description method may still be the best choice for degenerate, i.e. non-simple input (cf. [Sei04, p. 511]).

3.1.2. Graph traversal or pivoting methods

The second kind of vertex enumeration algorithms traverses the graph of a polytope.

Definition 3.3. Let P be polytope. We define the **graph of a polytope** $G(P)$ as the undirected, finite graph consisting of the vertices and edges of P. Thus two nodes u, v are adjacent in $G(P)$ if there is an edge in P joining u and v.

Sometimes this graph is also referred to as **adjacency graph**. By traversing the graph $G(P)$ of a polytope $P := P(A, b)$ all vertices of P can be enumerated. One of the graph traversal algorithms with a good software implementation is the **reverse search algorithm**.

The reverse search algorithm is due to Avis and Fukuda [AF92] and has an implementation in `[lrs]`. The latest improvements of this reverse search algorithm and `lrs` are described in [Avi00]. To get a better understanding of how this algorithm works we have a brief look at linear programming.

Given a polytope $P := P(A, b) \subset \mathbb{R}^d$ and a vector $c \in \mathbb{R}^d$, we call the task of finding an $x \in P$ such that $\langle c, x\rangle$ is maximal a **linear program**. One of the most common methods to solve a linear program is the simplex algorithm that we will briefly sketch here. The interested reader may find in [Sch98, Ch. 11] a more formal description.

The simplex algorithm works as follows. Let $x \in \mathbb{R}^d$ be a vertex of $P := P(A, b)$ where $A \in \mathbb{R}^{n\times d}$ and $b \in \mathbb{R}^n$. For x we can find an index set B with such that $A_B x = b_B$ and $A_B \in \mathrm{GL}(\mathbb{R}, d)$. We call B a basis for x. The remaining indices $N := \{1, \ldots, m\} \setminus B$ form a non-basis. We can take one index $j \in N$ and put it into the basis and choose $k \in B$ to put in N so that we obtain $B' := (B \cup \{j\}) \setminus \{k\}$ and $N' := (N \cup \{k\}) \setminus \{j\}$. We call this process **pivoting**. By pivoting we get from a vertex $x \in P$ to an adjacent vertex $x' := A^-_{B'} b_{B'}$. It may also happen that $x = x'$ and we will discuss this problem later. By repeated pivoting following a fixed pivot rule, the simplex algorithm finds a path $x_0 x_1 \ldots x_l$ on the graph $G(P)$ such that the function value increases, i. e. $\langle c, x_i\rangle < \langle c, x_{i+1}\rangle$. If for a node x_l no adjacent node with greater function value can be found, then x_l is an optimal solution for the linear program.

Thus the simplex algorithm traverses the graph $G(P)$ in order to find a path to an optimal vertex. The set of all such paths from all vertices of P forms a forest. The reverse search algorithm traverses each subtree T of the forest, beginning at the root T, which is an optimal vertex. A depth first traversal of T by reversing the pivot rule enumerates all vertices of P and thus solves the vertex enumeration problem. Because $G(P)$-traversal algorithms are usually based on variations of the simplex algorithms and its pivoting steps, the traversal algorithms are also often called pivoting methods.

The running time of the reverse search algorithm is proportional to the number of bases that are computed. For a d-dimensional polytope P with m facets the algorithm finds all vertices in time $O(md^2)$ per basis and $O(md)$ space. For non-simple polytopes many

bases may belong to the same vertex and the traversal stays at a node for longer than necessary. The number of bases to consider can be reduced by working only with so called lex-min bases (cf. [Avi00, Sec. 5]). Nevertheless, the number of bases may render very degenerate instances intractable for a reverse search. However, if P is simple and every vertex corresponds to a base then the vertex enumeration problem can even be solved in $O(nmd)$ time (cf. [Avi00, Thm. 6.2]). One big advantage of the reverse search in general is that it requires only $O(md)$ space.

Another feature of the reverse search algorithm is that it allows estimates of the running time and the number of vertices. This estimation works by performing a randomized depth-first traversal of the tree and counting the number of alternative bases and vertices along the way. The results of several independent traversals can be combined to reduce the variance of the estimation. To limit the running time of the estimate a maximal depth for the traversal can be specified. The interested reader may find the details in the article by Avis and Devroye [AD00].

3.2. Enumeration up to symmetries

The action of a group $G \leq \mathrm{Aut}(P)$ on the face lattice of a polyhedron P induces an equivalence relation. Two faces F_1, F_2 are G-equivalent if F_1 and F_2 are in the same orbit under the action of G. We say we perform a **description conversion up to symmetries** if the output contains one representative of every equivalence class. So before we look at algorithms for this in the next section we have to solve another problem first. A crucial part is to actually decide if two faces F_1, F_2 are equivalent.

We can describe the action of G on the face lattice of P as permutation of the face-vertex incidences (if P is given as $\mathcal{V}$) or of the face-facet incidences (if P is given as $\mathcal{H}$). Both cases are equivalent and we only discuss the latter situation. This means that if P is a polyhedron with n facets, we can regard each face F of P as a subset $\Gamma_F \subseteq \{1, \ldots, n\}$. In this notion G is isomorphic to a subgroup of S_n. Thus for the computational aspect of whether two faces of a polyhedron are equivalent we just have to decide whether two sets $\Gamma, \Delta \subseteq \{1, \ldots, n\}$ are in the same orbit of a permutation group $G \leq S_n$. In this section we discuss the situation that we know some sets $\Delta_1, \ldots, \Delta_k$, corresponding to faces of P. For other, probably many, sets Γ we have to decide whether $\Gamma \in \Delta_i^G$, i.e. Γ is equivalent to any of the Δ_i.

3.2.1. Methods from computational group theory

One way to decide if $\Gamma \in \Delta^G$ is to explicitly search for such a group element. The author has already presented ways to search for specific elements of a permutation group in [Reh10], based on [HEO05, Ser03, Leo91]. In this case we look for a representative of a coset $\mathrm{Stab}(\Delta, G)h$ such that $\Gamma \in \Delta^{\mathrm{Stab}(\Delta,G)h}$. This search has a worst case complexity of $O(|G|)$, so this might no be our best method for deciding equivalence.

A second way to decide quickly whether $\Gamma \in \Delta_i^G$ for some i is to explicitly compute the orbits Δ_i^G once for every i. We can store $S := \bigcup_i \Delta_i^G$ in a set data structure. Once such a set data structure is built, it allows to test for membership in S in $O(\log |S|)$ time (see [CLRS09, Ch. 13]). The disadvantage is that the full orbit Δ_i^G has to be com-

puted and the set data structure has to be set up. For every known face Δ this means an $O(|\Delta^G| \log |S|)$ time effort. Sometimes worse than time is the amount of memory needed to store all orbit elements. For large problems we may not be able to store the whole set S in fast RAM so that searching in the set of orbits would become relatively slow.

An alternative to storing the complete orbit is to store a **canonical representative** of each orbit. A **canonical map** c is a function with the following property. For every orbit Δ^G there exists a $\Delta_0 \in \Delta^G$ such that $c(\Delta') = \Delta_0$ for all $\Delta' \in \Delta^G$. A common choice is to define $c(\Delta) := \min_{\prec} \Delta^G$ where we use a lexicographic ordering $\prec$ on the subsets of $\{1, \ldots, n\}$. In [Lin04] Linton describes a fast algorithm to find the lexicographically smallest element of an orbit. This algorithm also has a worst case complexity of $O(|G|)$.

The canonical representative approach is a mixture of the two methods described before. It may take some time and memory to compute $c(\Delta_i)$ for each i. But once these are known we only have one expensive computation, namely $c(\Gamma)$, which can easily be compared to the known values of $c(\Delta_i)$. In the first method we have potentially k expensive operations for each Γ because for each Δ_i a coset representative may have to be searched.

In the next section we look at invariants, which may be used in combination with any of the methods discussed in this section to speed up equivalence testing.

3.2.2. Invariants and invariant theory

For deciding whether two faces of a polyhedron P are in the same orbit under a symmetry group $G \leq \mathrm{Aut}(P)$ we may use invariants, which are able to distinguish orbits. In our setting we consider two types of invariants: geometric and algebraic.

Geometric invariants arise when the action of G on P preserves a scalar product between vectors. This is the case for the restricted symmetries $\mathrm{RAut}(P)$ that we discussed in the last chapter. For $G \leq \mathrm{RAut}(P)$ and every $x, y \in \mathbb{R}^d$ we have $x^T Q^- y = (Ax)^T Q^- (Ay)$ where $A \in G$ is a restricted symmetry (cf. Equation 2.11 on page 14). Thus, for two faces F_1, F_2 of P being in the same G-orbit it is necessary that the inner angles of F_1 and F_2 are the same with respect to Q^-. These invariants are also described and used in [BDS09].

For algebraic invariants we consider the ring of polynomials $\mathbb{Q}[x_1, \ldots, x_n]$ in n variables over the rational numbers. Then the symmetric group S_n acts on a polynomial $f(x_1, \ldots, x_n) \in \mathbb{Q}[x_1, \ldots, x_n]$ by $f(x_1, x_2, \ldots, x_n)^\sigma := f(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)})$ for every $\sigma \in S_n$. The **invariant ring** $\mathbb{Q}[x_1, \ldots, x_n]^G$ of a group $G \leq S_n$ is the set of polynomials which are invariant under the action of G: $\mathbb{Q}[x_1, \ldots, x_n]^G := \{f \in Q[x_1, \ldots, x_n] : f^\sigma = f\}$. Every invariant ring is a sub-ring of $\mathbb{Q}[x_1, \ldots, x_n]$.

Let $S \subseteq \{1, \ldots, n\}$ be a set. We can associate a characteristic vector $\chi_S \in \{0, 1\}^n$ with S where the i-th component of χ_S is 1 if $i \in S$ and 0 otherwise. Further let $f \in Q[x_1, \ldots, x_n]^G$ be a polynomial from the invariant ring of G. If we let G act on S elementwise, denoted by S^G, then we clearly have $f(\chi_S) = f(\chi_T)$ for every $T \in S^G$ in the orbit of S. This gives us a necessary condition to check whether two sets S, T lie in the same G-orbit.

The easiest way to obtain an invariant $f \in \mathbb{Q}[x_1, \ldots, x_n]^G$ is to take any set $S \subseteq \{1, \ldots, n\}$ and compute its orbit sum. That means we set $f(x_1, \ldots, x_n) = \sum_{T \in S^G} \prod_{j \in T} x_j$. Clearly, this f is invariant under the action of G.

It can also be shown that there exist finitely many polynomials $f_1, \ldots, f_m \in \mathbb{Q}[x_1, \ldots, x_n]^G$ such that every invariant polynomial $g \in \mathbb{Q}[x_1, \ldots, x_n]$ can be written as a polynomial in $f_1, \ldots, f_m$, i.e. we write $Q[x_1, \ldots, x_n]^G = Q[f_1, \ldots, f_m]$. Theorem 10 of [CLO08, Sec. 7.4] shows that in fact $f_i(\chi_S) = f_i(\chi_T)$ for all m invariants f_i that generate the invariant ring is also sufficient for S and T to lie in the same orbit. We can use the same theory to work with the affine automorphisms as matrix group and compute its invariant ring. This has the advantage that the dimension of an interesting polyhedron is usually much smaller than the number of rays or facets, which reduces the problem dimension for invariants. At least in theory these invariants are a powerful instrument to check whether two elements are in the same orbit. The set of generating invariants f_i yields a hash function for orbit membership. We just have to evaluate all m generator polynomials and compare the values with those of an element of the orbit. In practice, however, things tend to be more difficult.

Recent research has created good algorithms to compute generators of invariant rings of finite linear group action and permutation groups, many of which are nicely explained in [Stu08] and [DK02]. The special case of permutation groups is also described in [Neu07, Ch. 4]. For a permutation group $G \leq S_n$ the generating invariants are a combination of the elementary symmetric function $s_n = x_1 x_2 \cdots x_n$ and orbit sums of so called special monomials with degree at most $\binom{n}{2}$ (cf. [Neu07, Thm. 4.22]). Because for our application we consider invariants as an alternative to storing the whole orbit, the total number and degree of the generating invariants is a crucial aspect.

Good algorithms that compute polynomials of low degrees involve computations with so called Gröbner bases. The reader may find an extensive introduction into Gröbner bases in [BW98] or [CLO08]. Computations with Gröbner bases tend to be computationally expensive and may be quite slow even on groups of moderate size (see also the experiments conducted in [Kem99] and [Thi01]).

From these considerations it remains unclear how useful the knowledge of all generators of the invariant ring is for orbit calculations alone. Nevertheless, if we know at least some parts of the invariant ring of a group in form of some invariants of low degree we may use them to speed up same-orbit tests.

3.3. Description conversion up to symmetries

In this section we analyze algorithms to perform a description conversion of a polyhedron up to symmetries. More specifically, we consider the following problem: Given a polyhedral cone $P := P(A, 0)$ and a symmetry group $G \leq \mathrm{Aut}(P)$, find rays $R := \{r_1, \ldots, r_l\} \subset P$ such that

$$P = \mathrm{cone} \bigcup_{i=1}^{l} r_i^G.$$

In other words, for every ray r generating P there is an $r_i \in R$ that is G-equivalent to r. In the following, we discuss three different methods to solve this task.

3.3.1. Direct method

The first algorithm we look at is the most trivial one can conceive. We first perform a full description conversion, for example with one of the methods described in Section 3.1. We then filter the complete list of rays that we obtained to include only one representative for each G-orbit.

Input: Polyhedral cone $P = P(A, 0)$, permutation group G
Output: List of G-inequivalent rays R

```
1 R ← ∅
2 compute set of rays R_0 such that P = cone(R_0)
3 forall r ∈ R_0 do
4 |   if r^G ∩ R = ∅ then
5 |   |   R ← R ∪ {r}
6 |   end
7 end
```

Algorithm 3.1: Direct method for description conversion up to symmetry

Algorithm 3.1 formalizes this method. In line *3* we check if r is equivalent to any other ray in the output list R. For this check we can use any of the method discussed in the previous section.

The obvious disadvantage of this direct method to obtain G-inequivalent rays is that all rays of P have to be computed before. By this requirement we forfeit a possible running time advantage that we might have because we are not interested in all rays. The following two methods therefore reduce a ray enumeration problem to smaller ones in order to take advantage of known symmetries.

3.3.2. Incidence Decomposition Method

The **Incidence Decomposition Method (IDM)** or Face Decomposition Method (cf. [BDS09], [CR01]) divides the problem of finding a ray description $\operatorname{cone} R$ for a polyhedron $P = P(A, 0)$ into subproblems. To find R we can also try to compute all rays R_i that are incident to a facet F_i of P. We then obtain R by merging all R_i for all facets F_i.

If we are interested in rays of P up to symmetries, given by a group $G \leq \operatorname{Aut}(P)$, we do not have to consider all facets. Let F_i and F_j be two facets of P that are in the same G-orbit. For every face H of P that is incident to F_i we can find a face H' incident with F_j such that H and H' are in the same G-orbit. This works because every $\sigma \in G$ is a combinatorial automorphism and thus an automorphism of the face lattice. If two facets F_i and F_j are in the same G-orbit, then also the incident rays R_i and R_j are in the same G-orbit. Thus we only have to compute rays for one representative of every G-orbit.

Grishukhin [Gri92] used this decomposition technique to compute all rays of the metric cone for seven points. The metric cone M_n for n points is given by the following $3\binom{n}{3}$ facet inequalities

$$x_{ij} - x_{ik} - x_{jk} \leq 0 \quad \text{for all triples } \{i, j, k\} \subseteq \{1, \ldots, n\} \tag{3.1}$$

and has dimension $\binom{n}{2}$. By permutation of the set $\{1, \ldots, n\}$ the symmetric group S_n acts on the facets of M_n defined in (3.1). For every two facets $x_{ij} - x_{ik} - x_{jk} \leq 0$ and $x_{ab} - x_{ac} - x_{bc} \leq 0$ there is a permutation $\sigma \in S_n$ with $\sigma(i) = a$, $\sigma(j) = b$, $\sigma(k) = c$. Thus all facets of M_n are in the same S_n-orbit. If we want to perform a vertex enumeration on M_n, it is enough to compute all rays that are incident to one arbitrary facet.

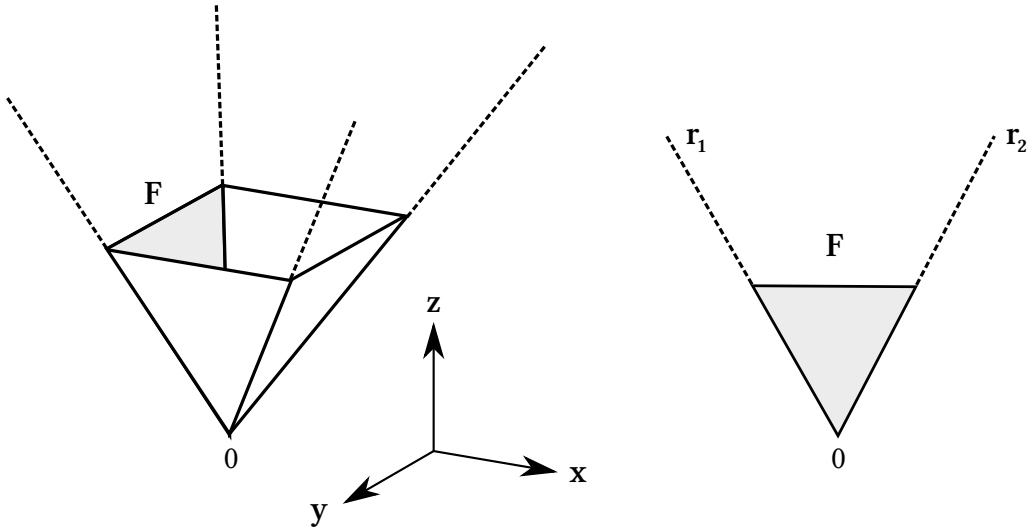

Figure 3.1.: Pyramidal cone with decomposition by facets

Example 3.4. Figure 3.1 gives an example in three dimensions. We see that the pyramidal cone P on the left is highly symmetric. By 90 degree rotations about the z-axis every face of P is mapped onto another face. In particular, there is only one orbit of facets. To compute all rays of P up to symmetries it is enough to compute the rays of a single facet So we solve the two-dimensional problem on the right-hand side of the figure and obtain two rays r_1, r_2 for F. Back in P with its symmetry group, we see that r_1 and r_2 are in the same orbit and conclude that P has one ray up to symmetries.

We now formalize the Incidence Decomposition Method in Algorithm 3.2. Our goal is to compute a set R containing all G-inequivalent rays of a polyhedron $P := P(A, 0)$ with symmetry group $G \leq \mathrm{Aut}(P)$. We can identify each facet

$$F_i := P \cap \{x \in \mathbb{R}^d \ : \ \langle a_i, x \rangle = 0\} \tag{3.2}$$

with the hyperplane normal vector a_i, which is a row of the matrix A. For the scalar product in (3.2) we regard a_i as a column. To enumerate all facets we loop over all matrix rows a_i and work with the face according to (3.2). If the matrix-description of P is non-redundant, then every face F_i is a facet.

For every G-inequivalent facet F_i, which we associate with the matrix row a_i, we compute an $\mathcal{H}$-description of F_i based on P. After adding the row $-a_i$ to A the polyhedron $P(A_i, 0)$ contains the equality $\langle a_i, x \rangle = 0$. Thus $P(A_i, 0)$ is an $\mathcal{H}$-description of F_i, but it still may contain redundant rows. Here we call a row redundant if removing it does not change the polyhedron. We will discuss below how to detect redundancy. After we have

removed all redundant rows from A_i, we obtain a $(d-1)$-dimensional polyhedron that is defined by less inequalities than P. Therefore we can expect that $F_i := P(A', 0)$ is easier to work with than P.

Input: Polyhedral cone $P = P(A, 0)$, permutation group G
Output: List of G-inequivalent rays R

$R \leftarrow \emptyset$
$F \leftarrow \emptyset$ `// list of G-inequivalent facets`
forall row $a_i \in A$ **do**
  $F_i \leftarrow P \cap \{x \in \mathbb{R}^d \ : \ \langle a_i, x\rangle = 0\}$
  if $F_i^G \cap F = \emptyset$ **then**
    $F \leftarrow F \cup \{F_i\}$
    $A_i \leftarrow A \cup \{-a_i\}$
    remove redundant rows from A_i
    compute set of rays R_0 such that $P(A_i, 0) = \text{cone}(R_0)$
    forall $r \in R_0$ **do**
      if $r^G \cap R = \emptyset$ **then**
        $R \leftarrow R \cup \{r\}$
      end
    end
  end
end

Algorithm 3.2: Incidence Decomposition Method for description conversion up to symmetry

We can identify redundant rows by solving linear programs. Let P be the polyhedron defined by the m inequalities $\langle a_i, x\rangle \leq b_i$ for $1 \leq i \leq m$. To check if a row j is redundant we can solve the following linear program:

$$\begin{aligned} \max_x \quad & \langle a_j, x\rangle \\ \text{s.t.} \quad & \langle a_i, x\rangle \leq b_i \quad \text{for all } 1 \leq i \leq m \text{ and } i \neq j. \\ & \langle a_j, x\rangle \leq b_j + 1 \end{aligned} \tag{3.3}$$

If and only if (3.3) has a solution x_0 and the relation $\langle a_j, x_0\rangle \leq b_j$ holds, then a_j is a redundant row for P.

How well the IDM works is mainly influenced by two factors: the number of G-orbits of facets and the number of redundant rows we can identify for the sub-problems. If there are only a few G-orbits of facets of P, which also have a much smaller $\mathcal{H}$-description than P, the sub-problems may in total be a lot faster to complete than the original problem P.

3.3.3. Adjacency Decomposition Method

Another way to divide a description conversion problem into smaller sub-problems is the **Adjacency Decomposition Method (ADM)**. Instead of searching all rays that are incident to a given facet, the ADM computes rays that are adjacent to each other. The ADM was

used in [DFPS01] to compute the vertices of metric polytopes. It was also described in [CR01] as a more competitive alternative to the IDM. As with all methods before, we only discuss the case of a polyhedral cone $P := P(A, 0)$ here. It can easily be carried over to the case of a polyhedron but treating vertices and rays together would blow up the notation so we stick to the simple cone case here.

The idea of the ADM is again quite natural. We say two rays r, r' of a d-dimensional cone $P \subset \mathbb{R}^d$ are **neighbors** if there is a 2-face F of P such that $r, r' \in F$ and $r \neq r'$. Suppose we are given a cone P with a symmetry group $G \leq \mathrm{Aut}(P)$ and already know a set R of (G-inequivalent) rays of P. We compute all neighbors R' of rays in R and discard all rays in R' that are G-equivalent to other rays in $R \cup R'$. We add all remaining rays to R and again compute all neighbors R' of R. If all neighbors R' are G-equivalent to rays in R, then R is a complete list of G-inequivalent rays of P. This holds because the action of G preserves neighborhood: r,r' are neighbors if and only if $g(r)$,$g(r')$ are neighbors for every $g \in \mathrm{Aut}(P)$.

In the dual problem, the facet enumeration, computing adjacent facets is well-known as gift-wrapping (cf. [CK70]). For the ray enumeration problem the method to compute neighboring rays is slightly different.

Definition 3.5. Let $r \in P$ be a ray of a d-dimensional cone $P = P(A, 0)$. We call the inequality $\langle a_i, x\rangle \leq 0$ **active** in r if it is strictly fulfilled in r, i.e. $\langle a_i, r\rangle = 0$. If an inequality is not active we call it **inactive**.

Definition 3.6. We define the **axis** a_P of a polyhedral cone $P := P(A, 0)$ as

$$a_P := \sum_{i=1}^{m} a_i \tag{3.4}$$

where a_i are the m columns of A^T.

The axis of a full-dimensional polyhedral cone P never equals 0. If P is full-dimensional, then the interior of P cannot be empty and P must contain a ray r such that all inequalities are inactive in r. This yields $\sum_i \langle a_i, r\rangle = \langle a_P, r\rangle < 0$, so especially $a_P \neq 0$.

Definition 3.7. We define the **support cone** $C(P, r)$ of a ray r in P as

$$C(P, r) := \{x \in \mathbb{R}^d \ : \ \langle a_j, x\rangle \leq 0 \text{ for all } j \in I\} \tag{3.5}$$

where $I = \{i \ : \ \langle a_i, r\rangle = 0\}$ denotes the index set of inequalities active in r.

The support cone $C(P, r)$ of a ray r is a d-dimensional cone because it is a cone and a superset of P. However, $C(P, r)$ is not pointed because it contains the line $r\mathbb{R}$. To make it pointed we can intersect it with any hyperplane $H := \{x \in \mathbb{R}^d \ : \ \langle c, x\rangle = 0\}$ that does not contain r. Then we can decompose $C(P, r)$ into the Minkowski sum $r\mathbb{R} + (C(P, r) \cap H)$. We observe that $C_H := C(P, r) \cap H$ is again a cone and C_H and $C(P, r)$ have the same combinatorial structure. Each k-dimensional face of $C(P, r)$ uniquely corresponds to a $(k-1)$-dimensional face of C_H. We will see shortly that it is advantageous to choose $H_P := \{x \in \mathbb{R}^d \ : \ \langle a_P, x\rangle = 0\}$. The intersection of $C(P, r)$ and H_P is a $(d-1)$-dimensional cone $\bar{C}(P, r)$, which we call the **reduced support cone** of r in P.

Lemma 3.8. Let r, r' be two neighborly rays of the cone $P := P(A, 0)$. Then, after scaling r', the difference $r' - r$ is a ray of the reduced support cone $\bar{C}(P, r)$.

Proof. Without loss of generality we can assume that

$$\sum_{i=1}^{m} \langle a_i, r' - r \rangle = \langle a_P, r' - r \rangle = 0. \tag{3.6}$$

Because $\sum_{i=1}^{m} \langle a_i, r \rangle < 0$ and $\sum_{i=1}^{m} \langle a_i, r' \rangle < 0$, we can always find a suitable representative r' such that (3.6) holds.

Let the support cone $C(P, r)$ be given by the $d-1$ inequalities $\langle a_1, x \rangle \leq 0, \ldots, \langle a_{d-1}, x \rangle \leq 0$. Because $C(P, r)$ is the support cone of r, all inequalities are active in r. The rays r, r' are neighbors, so for r' there are only $d-2$ active inequalities, say $\langle a_2, r' \rangle = 0, \ldots, \langle a_{d-1}, r' \rangle = 0$. For the remaining a_1 we must thus have $\langle a_1, r' \rangle < 0$. This together with (3.6) shows that $(r' - r) \in \bar{C}(P, r)$. Because for $r' - r$ there are $d-2$ active inequalities in $\bar{C}(P, r)$, we conclude that $r' - r$ is a ray of $\bar{C}(P, r)$. □

This shows that we can find the neighbors of a ray r by enumerating the rays of the reduced support cone of r. Every neighbor r' has to be a combination $r' = r + \lambda s$ where s is a ray of the reduced support cone of r and $\lambda \in \mathbb{R}^+$. For an inequality $\langle a_j, x \rangle \leq 0$ that is active in r' we can solve for λ:

$$\lambda = -\frac{\langle a_j, r \rangle}{\langle a_j, s \rangle} > 0. \tag{3.7}$$

Because $r \in P$, we must have $\langle a_j, r \rangle < 0$ and thus $\langle a_j, s \rangle > 0$. Among all the inequalities for which (3.7) holds we must pick the one with the smallest λ since $r' \in P$ implies

$$\begin{aligned} & \forall k: \quad \langle a_k, r' \rangle = \langle a_k, r + \lambda s \rangle \leq 0 \\ \iff \quad & \forall k: \quad \lambda \leq -\frac{\langle a_k, r \rangle}{\langle a_k, s \rangle} \quad \text{where } \langle a_k, s \rangle > 0 \end{aligned}$$

For such a minimal λ, there are $d-1$ active inequalities in $r' := r + \lambda s$. One equality, $\langle a_j, r' \rangle = 0$, is given by (3.7). The other $d-2$ come from the restricted support cone: s is a ray of $\bar{C}(P, r)$, hence there are $d-2$ equalities

$$\begin{aligned} \langle a_{i_1}, r \rangle = \cdots = \langle a_{i_{d-2}}, r \rangle = 0 \\ \langle a_{i_1}, s \rangle = \cdots = \langle a_{i_{d-2}}, s \rangle = 0 \end{aligned}$$

which also hold in r'. Hence, r' is contained in $d-1$ facets and is thus a ray of P.

It remains to show that the set $I := \{j \ : \ \langle a_j, r \rangle < 0 \text{ and } \langle a_j, s \rangle > 0\}$ is not empty so that we can pick a suitable λ. We remember that s is a ray of the reduced support cone of r, so in particular $\langle a_P, s \rangle = \sum_{i=1}^{m} \langle a_i, s \rangle = 0$. There is at least one inequality of the support cone that is inactive in s, say $\langle a_{i_0}, s \rangle < 0$. Hence, there must exist another inequality j such that $\langle a_j, s \rangle > 0$. This inequality j cannot be an inequality of the support cone and thus $\langle a_j, r \rangle < 0$. Therefore the set I always contains at least one element.

Algorithm 3.3 displays the method that we have discussed so far to compute all neighbors of a ray r. We enumerate all rays of the reduced support cone of r and compute

Input: Polyhedral cone $P = P(A, 0)$, ray r
Output: List N of all neighbors of r in P

$N \leftarrow \emptyset$
$P' \leftarrow$ reduced support cone $\bar{C}(P, r)$ of r
compute set of rays R' of P'
forall $s \in R'$ **do**
  $I \leftarrow \{j \ : \ \langle a_j, r\rangle < 0 \text{ and } \langle a_j, s\rangle > 0\}$
  $j \leftarrow \min_{i \in I} -\frac{\langle a_i, r\rangle}{\langle a_i, s\rangle}$
  $r' \leftarrow r - \frac{\langle a_j, r\rangle}{\langle a_j, s\rangle} s$
  $N \leftarrow N \cup \{r'\}$
end

Algorithm 3.3: Computing neighbors of a ray in a polyhedral cone

for every ray the associated neighbor of r. We can use this method to compute all G-inequivalent rays of a polyhedral cone. Algorithm 3.4 formalizes this idea to compute all G-inequivalent rays of a polyhedral cone. We split the problem of computing the rays of a d-dimensional cone into several smaller problems of computing rays of cones in dimension $d - 1$.

Input: Polyhedral cone $P = P(A, 0)$, permutation group G
Output: List of G-inequivalent rays R

$r_0 \leftarrow$ one ray of P
$R \leftarrow \{r_0\}$
$T \leftarrow \{r_0\}$ `// todo-list of rays`
while $T \neq \emptyset$ **do**
  choose $r \in T$
  $T \leftarrow T \setminus \{r\}$
  $P' \leftarrow$ reduced support cone $\bar{C}(P, r)$ of r
  compute set of rays R' of P'
  forall $s \in R'$ **do**
    compute the neighboring ray r' of r in direction of s
    if $r'^G \cap R = \emptyset$ **then**
      $R \leftarrow R \cup \{r'\}$
      $T \leftarrow T \cup \{r'\}$
    end
  end
end

Algorithm 3.4: Adjacency Decomposition Method for description conversion up to symmetry

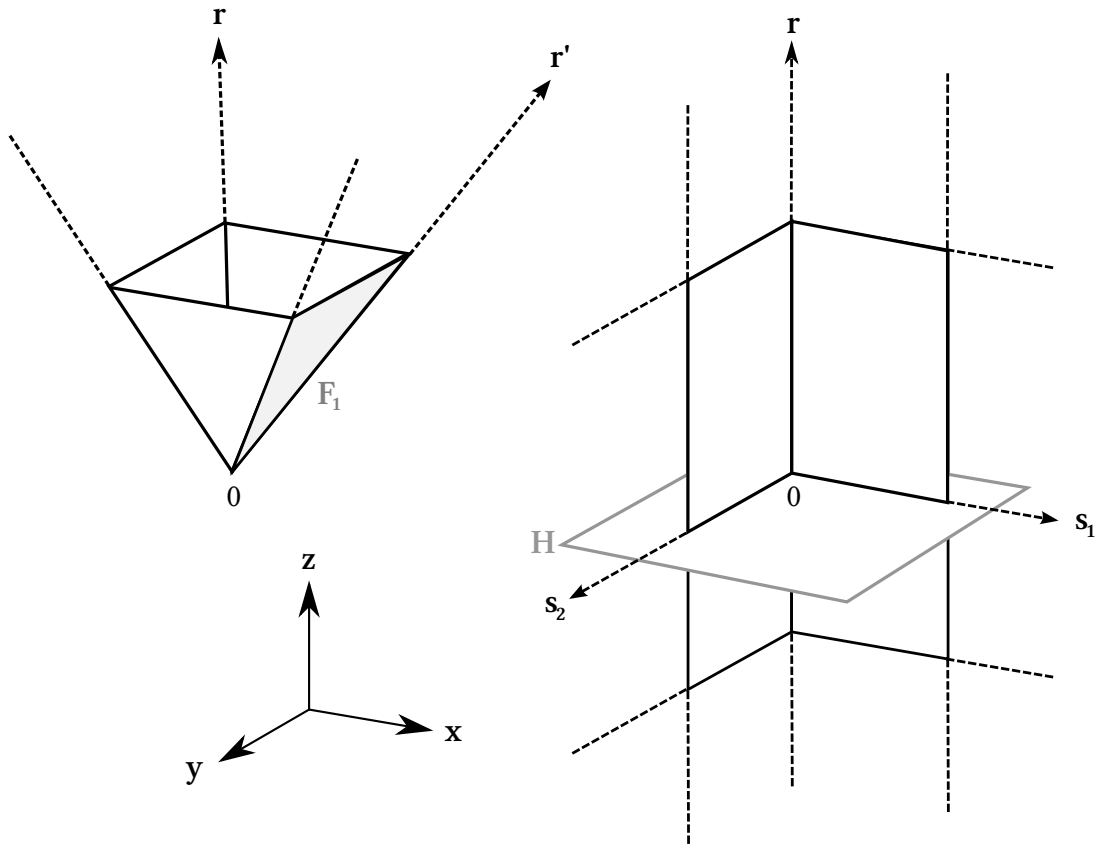

Figure 3.2.: Pyramidal cone with support cone of its ray r

Example 3.9. Consider again the pyramidal cone on the left-hand side of Figure 3.2. It can be described by the matrix

$$A = \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & -1 & -1 \end{pmatrix}.$$

From this we compute the axis $a_P = (0, 0, -4)^T$. We start the ADM at ray $r = (-1, -1, 1)^T$ and compute its support cone $C(P, r) = P(\begin{pmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \end{pmatrix}, 0)$. For the restricted support cone, intersected with the axis hyperplane H, which is the xy-plane in this example, we obtain

$$\bar{C}(P, r) = P(\begin{pmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix}, 0) \cong P(\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, 0).$$

The cone $\bar{C}(P, r)$ has two rays: $s_1 = (1, 0, 0)^T$ and $s_2 = (0, 1, 0)^T$, thus r has two neighboring rays. We compute the neighbor of r in direction of $s := s_1$ by first computing the index set I. A quick calculation shows that only one inequality, a_1 corresponding to F_1 in the figure, comes into consideration: $I = \{1\}$ because $\langle a_1, r \rangle = -2 < 0$ and

$\langle a_1, s\rangle = 1 > 0$. It is easy to find the minimizing hyperplane from I and we obtain the neighboring ray

$$r' = r - \frac{\langle a_1, r\rangle}{\langle a_1, s\rangle} s = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix} - (-2) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}.$$

The last open problem we have to discuss is how we find a first ray of a cone P to start the ADM. This is also a common problem in linear programming where a vertex is sought to start the simplex algorithm. For instance, we could minimize the function $\langle a_P, x\rangle$ over the non-empty polyhedron $P \cap \{x \in \mathbb{R}^d \ : \ \langle a_P, x\rangle \geq -1\}$. Every vertex solution of this problem corresponds to a ray of P. We could also run one iteration of a pivoting description conversion method as discussed in Section 3.1.2 to find one ray of P.

At the end of this section we discuss an improvement of the Adjacency Decomposition Method which may help to reduce the number of support cone calculations. We say a graph G is d-**connected** if the removal of any $d-1$ vertices, and the edges incident to them, leaves G connected. With this definition and Definition 3.3 on page 27 for the graph $G(P)$ of a polytope in mind we can state the following theorem due to Balinski.

Theorem 3.10 (Balinski's theorem [Bal61]). The graph $G(P)$ of a d-dimensional polytope P is d-connected.

Proof. See, for instance, [Zie95, Thm. 3.14]. □

Corollary 3.11. Let P be a pointed d-dimensional polyhedral cone. Then the neighborhood graph of rays is $(d-1)$-connected.

Proof. Because P is pointed, we can find a $(d-1)$-dimensional hyperplane H such that $P' := H \cap P$ is a $(d-1)$-dimensional polytope. We can apply Theorem 3.10 and identify the vertices of P' and the rays of P. □

This means that we do not have to work with all support cones to reach all rays. For a d-dimensional cone P we can skip the neighborhood computation for $d-2$ rays and still cover P. This fact can be helpful if we may omit the computation of the most difficult cones, provided we know (or guess) that they are difficult to treat. We will come back to this problem in the next section.

3.4. Recursion

3.4.1. General remarks

We saw in the last section how we can split the description conversion up to symmetries of a d-dimensional cone into smaller subproblems in dimension $d-1$. There may be cases where one of these subproblems is still too difficult to be solved with one of the approaches of Section 3.1. Thus we may consider to apply again one of the decomposition methods to cut the problem into smaller pieces. The only difference is that we have not introduced any symmetries of the subproblem in the formulations of Algorithm 3.2 and Algorithm 3.4 yet.

In the following we again consider the description conversion problem up to symmetries of a polyhedral cone $P := P(A, 0)$ with a symmetry group $G \leq \mathrm{Aut}(P)$.

In the Incidence Decomposition Method it would be enough to obtain for every facet P' a list of G-inequivalent rays. The group G itself is no subgroup of $\mathrm{Aut}(P')$ so we cannot use it directly for the subproblem. However, we may choose the maximal subgroup of G which is also a subgroup of $\mathrm{Aut}(P')$: the stabilizer $\mathrm{Stab}(G, P')$ of P' in G.

For the Adjacency Decomposition Method the situation is less obvious. We can easily construct an isomorphism between the action of G on the faces of P which are incident to a ray r and the support cone $C(P, r)$. Furthermore, we already have seen that the cones $\bar{C}(P, r)$ and $C(P, r)$ are combinatorially equivalent. This also means that the actions of G on $\bar{C}(P, r)$ and $C(P, r)$ are isomorphic.

Let r be a ray of P then two neighbor rays r_1, r_2 are G-inequivalent if and only if the 2-dimensional faces F_1, F_2 between r and r_1, r_2, respectively, are G-inequivalent. In turn, the two faces $F_1, F_2 \subset C(P, r)$ are G-inequivalent if and only if the two rays $s_1, s_2 \in \bar{C}(P, r)$ are G-inequivalent. Thus for the ADM it is enough to enumerate the rays of $\bar{C}(P, r)$ up to equivalence under the action of G. The suitable symmetry group to choose is the stabilizer $\mathrm{Stab}(G, \bar{C}(P, r))$, which is a subgroup of $\mathrm{Aut}(\bar{C}(P, r))$ and also isomorphic to a subgroup of G.

We have seen that in both IDM and ADM the subproblems consist of enumerating the $\mathrm{Stab}(G, P')$-inequivalent rays of a $(d-1)$-dimensional cone P'. Thus we can use one of the methods discussed in the last section 3.3 for P'. To speed up the computation with P' we may even consider to choose a larger symmetry group H with $\mathrm{Stab}(G, P') \leq H \leq \mathrm{Aut}(P')$. For instance, if P' is one highly symmetric facet of a rather asymmetric P we may solve the problem much more quickly by considering and computing the restricted symmetries $\mathrm{RAut}(P')$ as discussed in Chapter 2. We may then use $H := \mathrm{Stab}(G, P') \cup \mathrm{RAut}(P')$ as symmetry group for the computation with P'.

Suppose we choose a group $H \gneq \mathrm{Stab}(G, P')$ and obtain a complete list R_H of H-inequivalent rays of P'. Then we have to transform R_H into a complete list of $\mathrm{Stab}(G, P')$-inequivalent rays of P'. There are at least two ways to solve this problem. The first one is to compute for every $r \in R_H$ the full orbit r^H and split it into smaller orbits $r_1^{\mathrm{Stab}(G,P')}, \ldots, r_l^{\mathrm{Stab}(G,P')}$. The second, more sophisticated method is double coset decomposition as described in [BDS09].

Let $r \in R_H$ be a ray from the subproblem result list. Suppose we have a set of elements $h_1, \ldots, h_l \in H$ such that

$$H = \bigcup_{i=1}^{l} \mathrm{Stab}(H, r)\, h_i\, \mathrm{Stab}(G, P'). \tag{3.8}$$

Then we can split the orbit of r under H into the required $\mathrm{Stab}(G, P')$-orbits by

$$r^H = \bigcup_{i=1}^{l} r^{(h_i\, \mathrm{Stab}(G,P'))} = \bigcup_{i=1}^{l} r_i^{\mathrm{Stab}(G,P')}$$

where $r_i := r^{h_i}$ are the new orbit representatives. A decomposition into double cosets as in (3.8) is a well known problem in computational group theory. For instance, [HEO05,

Sec. 4.6.8] lists two approaches of how to solve this problem. Double coset decomposition is also implemented in standard computer algebra software like [GAP] or [Magma].

We will discuss in the experimental section 4.2.4 under what circumstances a group $H \gneq \mathrm{Stab}(G, P')$ may help to accelerate the description conversion process.

3.4.2. Recursion and solution strategy

The previous section showed that description conversion up to symmetry can be turned into a recursive process. We can use the decomposition methods to split a description conversion problem up to symmetries into smaller pieces that also can be computed up to symmetries. At all stages we have to decide which of the methods that we looked at in Section 3.3 we use for the polyhedron. In the following we will discuss some heuristics which may guide us in the decision process.

If we can solve a problem in reasonable time with a description conversion algorithm without symmetries, like `cdd` and `lrs` from Section 3.1, then we should do without a decomposition method. For small problems the computational overhead in decomposition methods will be larger than the potential gain by exploiting symmetries. Thus we may split the main question, which method to use, into two parts. First, we have to decide whether we attempt a direct conversion or use a decomposition method. Second, in the latter case we must decide which decomposition method we use. For the decision of the core description conversion algorithm without symmetries, i.e. `cdd`, `lrs` or something else, the reader may consider the results of [ABS97].

We begin with the first step: deciding whether to attempt a direct conversion or not. One of the advantages of pivoting algorithms for description conversion such as `lrs` is that they allow an estimation of the expected running time. Although these estimations are by the nature of their construction not very accurate, they still give an impression of the magnitude of running time. For more information and experiments with the `lrs` estimator the interested reader may consider [AD00]. We should note though that the estimation itself takes some time. Especially for small problems the time to get an estimate and the time to solve the problem are of similar order, so this kind of estimation needs some care, in particular if `lrs` is not the preferred description conversion algorithm for a polyhedron.

Another possible indicator of the difficulty of a problem is the **incidence number** $\mathrm{inc}\, F$ of a face F, which is the number of facets incident to F. The authors of [BDS09] have experienced that a relatively low incidence number may correspond to a relatively easy problem. They propose to order the subproblems of a decomposition method by incidence number ascendingly and solve as much of them as possible by a direct method. We can combine this ordering with a running time estimate or even time-outs to find the cut-off point between direct and decomposing conversion.

If used with the Adjacency Decomposition Method, this ordering also has the advantage that the problems which are suspected to be the most difficult may not even have to be solved. Balinski's theorem (cf. Corollary 3.11 on page 38) enables us to skip $d-2$ subproblems of a d-dimensional polyhedron, which we may choose to be the ones with the highest incidence number. This application of Balinski's theorem has proven to be useful for problems arising in the geometry of numbers (cf. [DSV07, BDS09]).

If we conclude that a problem cannot be solved directly, we can choose between the Incidence and the Adjacency Decomposition Method. A priori it may again not be clear which one will work best, but the IDM allows at least a rough estimation. In contrast to the ADM, all subproblems of the IDM can be determined before one of it is solved because they do not depend on each other. For a polyhedron P we can easily compute the number of G-inequivalent facets $F_1, \dots, F_l$. After solving a linear program to determine redundant rows, we also know the incidence number of each facet $\operatorname{inc} F_i$. The smaller $\sum_{i=1}^{l} \operatorname{inc} F_i$ relative to $\operatorname{inc} P$, the more promising an incidence decomposition is.

Recursive IDM was used in [DFMV03] to compute parts of the face lattice of metric polytopes. In [DSV07] recursive ADM was an important ingredient to classify perfect eight-dimensional forms. However, as [BDS09] observe, no combination of both techniques has been used so far. We will look at the potential of this strategy in Section 4.2.

3.4.3. Adjacency Decomposition Method

At the end of this section we look at a formalization of the ADM into which Balinski's theorem (Corollary 3.11) is integrated.

Input: Polyhedral cone $P = P(A, 0)$ of dimension d, permutation group G
Output: List of G-inequivalent rays R

```
1  r_0 ← one ray of P
2  R ← {r_0}
3  T ← {r_0}                                          // todo-list of rays
4  while T ≠ ∅ do
5  |  n_remaining ← Σ_{r∈T} |r^G|
6  |  if n_remaining ≤ (dim P) − 2 then
7  |  |  break                          // we may ignore d − 2 arbitrary rays
8  |  end
9  |  choose r ∈ T such that the incidence number of r is minimal
10 |  T ← T \ {r}
11 |  P' ← reduced support cone C̄(P, r) of r
12 |  H ← Stab(G, P')
13 |  compute set of rays R' of P' up to symmetry H
14 |  forall s ∈ R' do
15 |  |  compute the neighboring ray r' of r in direction of s
16 |  |  if r'^G ∩ R = ∅ then
17 |  |  |  R ← R ∪ {r'}
18 |  |  |  T ← T ∪ {r'}
19 |  |  end
20 |  end
21 end
```

Algorithm 3.5: Adjacency Decomposition Method for description conversion up to symmetry with Balinski's criterion

The main change compared to Algorithm 3.4 on page 36 is in line *4* and following and lines *11* and *12*. The latter change is that we compute the rays of P' up to $\operatorname{Stab}(G, P')$

symmetry. The former is the application of Balinski's theorem. Because T contains rays up to G-symmetry we have to count all elements in the G-orbits to properly apply the theorem. It may be worthwhile not to perform this computation explicitly but to use the orbit-stabilizer-theorem instead:

$$n_{\text{remaining}} \leftarrow \sum_{r \in T} |r^G| = \sum_{r \in T} |\operatorname{Stab}(G, r)|. \tag{3.9}$$

One advantages of this is that for every ray r we have the identity $\operatorname{Stab}(G, r) \cong \operatorname{Stab}(G, \bar{C}(P, r))$ as discussed in Section 3.4.1 on page 39. Because we have to compute this stabilizer for every ray of the todo-list anyway, we may save every stabilizer that we compute for (3.9) and re-use it later in line *11* of the so modified Algorithm 3.5.

3.5. Extensions

3.5.1. Adjacency graph

In his recent breakthrough paper [San10], Santos disproved the famous Hirsch conjecture from 1957. To state this conjecture we need one more term from graph theory. For a graph G the diameter $\delta(G)$ is the smallest number such that any two vertices can be connected by a path with at most $\delta(G)$ edges. The **Hirsch conjecture** is about a bound of the diameter of the graph $G(P)$ of a polytope: Hirsch conjectured that for a d-dimensional polytope P with n facets the following holds:

$$\delta(G(P)) \leq n - d. \tag{3.10}$$

This means that for every two vertices $u, v \in P$ there exists a path between u and v that has at most length $n - d$.

Santos gives an example of a polytope in dimension 43 with 86 facets that violates the bound (3.10). This polytope is constructed from a polytope P' in dimension 5 with 48 facets with $\delta(G(P')) = 6$. This P' is simple enough so that its adjacency graph up to symmetry can be computed and verified by hand and thus the diameter value can be proven.

In this section we briefly discuss the use of the Adjacency Decomposition Method, which we have looked at in Section 3.3.3, to compute the adjacency graph of a polyhedron up to symmetries. As the ADM already implicitly traverses the adjacency graph of a polyhedron we can simply record the adjacencies while the algorithm is running. We have to be careful not to use Balinski's criterion because this guarantees to find all rays but not all adjacencies between rays. Therefore Algorithm 3.6 is a modification of Algorithm 3.4 on page 36 to compute all rays and ray adjacencies of a polyhedral cone up to a given symmetry group.

Input: Polyhedral cone $P = P(A, 0)$, permutation group G
Output: List of G-inequivalent rays R, list R_A of adjacencies in R

```
1  r_0 ← one ray of P
2  R ← {r_0}
3  T ← {r_0}                                          // todo-list of rays
4  R_A ← ∅
5  while T ≠ ∅ do
6  |  choose r ∈ T
7  |  T ← T \ {r}
8  |  P' ← reduced support cone C̄(P, r) of r
9  |  compute set of rays R' of P'
10 |  forall s ∈ R' do
11 |  |  compute the neighboring ray r' of r in direction of s
12 |  |  if r'^G ∩ R = ∅ then
13 |  |  |  R ← R ∪ {r'}
14 |  |  |  T ← T ∪ {r'}
15 |  |  |  R_A ← R_A ∪ {{r, r'}}
16 |  |  end
17 |  end
18 end
```

Algorithm 3.6: Adjacency Decomposition Method for description conversion up to symmetry with adjacency graph

3.5.2. Face lattice

For some applications one may not only be interested in the vertices and rays of a polyhedron up to symmetries but also in other parts of the face lattice (up to symmetries). We can use the Incidence Decomposition Method to compute all faces of a polyhedron up to symmetries. As we have seen before, the IDM splits a polyhedron $P = P(A, b)$ into orbits of its facets. We can then recursively apply the IDM to the facets and obtain the facets of the facets of P, i.e. faces of codimension 2. Repeating this process until we reach the vertices of P yields the full face lattice up to symmetries.

Algorithm 3.7 formalizes this process. The variable i denotes the codimension of the faces in the todo-list T. All G-inequivalent facets of the faces in T are computed and stored in T', the todo-list for the next iteration. To compute the facets of a face F in line *9*, we can proceed as in the original IDM in Algorithm 3.2 on page 33. For every inequality $\langle a_i, x\rangle \leq 0$ that is inactive in F we consider $F' := F \cap \{x \ : \ \langle a_i, x\rangle = 0\}$. If $\dim F' = \dim F - 1$, then F' is a facet of F. In this manner we can compute all facets.

The authors of [DFMV03] have used the Incidence Decomposition Method recursively to compute faces of low codimension of the metric polytope. As they put it, "face lattices are fat". This means that the number of $\lfloor \frac{d}{2} \rfloor$-faces of a d-dimensional polytope may be much larger than the number of vertices or facets. Even reducing the considerations to orbit representatives of faces often leaves face lattices "fat". Thus it may not always be possible to compute the complete face lattice, but only the top layers with faces of low (co)dimension.

Input: Polyhedral cone $P = P(A, 0)$, permutation group G
Output: List $\mathcal{F}$ of all G-inequivalent faces

```
1  i ← 0
2  T ← {P}
3  T' ← ∅
4  F ← ∅
5  while i < dim P do
6  |  while T ≠ ∅ do
7  |  |  choose one element F from T
8  |  |  T ← T \ {F}
9  |  |  forall facets F' of F do
10 |  |  |  if F'^G ∩ F = ∅ then
11 |  |  |  |  F ← F ∪ {F'}
12 |  |  |  |  T' ← T' ∪ {F'}
13 |  |  |  end
14 |  |  end
15 |  end
16 |  T ← T'
17 |  T' ← ∅
18 |  i ← i + 1
19 end
```

Algorithm 3.7: Incidence Decomposition Method to compute the face lattice up to symmetry

4. Implementation

4.1. SymPol

The author has developed a software christened `SymPol`, which implements many of the concepts presented in this thesis. `SymPol` is written in C++ to avoid performance barriers. It uses the renowned C++ library [`Boost`] for various things, one of the most important being smart pointers to reduce the risk of memory leaks. All parts that require algorithmic treatment of groups are handled by the author's permutation group library [`PermLib`]. This includes computing matrix automorphisms, set stabilizers, orbits and the like.

Alternatively, the restricted symmetries of a polyhedron may also be computed by [`nauty`] and [`NTL`]. Although this combination is often faster than `PermLib` as we will see in the following section, it has not been included in `SymPol` by default. Because `NTL` has its own build system, it is not easy to integrate into other software. The software `nauty` is difficult to include because its software license is not compatible with the open source license GPL, which is used by `lrs` and `cdd` and therefore also for `SymPol`.

Description conversion up to symmetries is implemented in the three different forms that were presented in Section 3.3: direct method, Incidence Decomposition Method and Adjacency Decomposition Method. Unfortunately, the author is not aware of an implementation of double coset decomposition outside dedicated algebra software like [`GAP`] and [`Magma`]. So `SymPol` always uses only the stabilizer instead of all linear symmetries of subproblems. We will discuss below to what extent this is a performance limit.

For the core task of performing a description conversion without symmetries `SymPol` can choose between Avis' [`lrs`] and Fukuda's [`cdd`]. As both `lrs` and `cdd` use different techniques, choosing one or the other may have a huge impact on performance. All computations with polyhedra are performed with the help of the arbitrary precision library [`GMP`] to avoid rounding errors.

Internally, all polyhedra are stored in a simple $\mathcal{H}$-representation in matrix form $Ax \leq b$. All faces of polyhedra are identified with their facet incidences. This allows storing faces as bitsets where the elements are row indices of the defining matrix A. A face lattice automorphism then acts on a face simply by permuting these facet indices. If the dimension of a face has to be computed, we have to determine the rank of the corresponding sub-matrix of A. The rank computation is done with a simple Gaussian elimination (cf. [CLRS09, Ch. 28]).

The interested reader may find a more detailed description of `SymPol` and its features in Appendix B. An alternative to `SymPol` is the `GAP` package [`Polyh`] by Dutour Sikiric, which offers a different feature set. It provides the Adjacency Decomposition Method as only decomposition method, but it computes new symmetries for subproblems and can use double coset decomposition. This together with a "banking" feature, which stores the solution of all solved problem parts for further use, enables it to solve large

instances (cf. [DSV07]). Such a banking feature is especially useful if multiple recursion levels are necessary for a polyhedron P. In this case a face F' of a facet F_1 of P occurs also as a face of another facet F_2 of P. Thus F may have to be converted twice if the result of the first occurrence has not been saved.

4.2. Computational experiments

All experiments were conducted on a server of the Faculty of Mathematics with four dual-core AMD Opteron 2.8 GHz CPUs and 16 GB RAM, running an OpenSuSE 10.2. All binaries were compiled with the GNU g++ compiler in version 4.1.2. Because of occasional segmentation faults in the highest optimization setting `-O3` with this compiler version, the binaries were created with `-O2` optimization flags.

4.2.1. Polyhedra test set

To the best of the author's knowledge there is no existing set of benchmark instances for description conversion of polyhedra. To evaluate the performance of the algorithm implementations in `SymPol`, the author compiled an own set of problem instances from different areas. Most of these were already used and described by others exploiting symmetries of polyhedra.

One of the most simple and symmetric geometric object is the d-dimensional cube $C_d = [0,1]^d$. It has $2d$ facets and 2^d vertices. All its combinatorial symmetries are also affinely realizable and $|\mathrm{Aut}(C_d)| = d! \cdot 2^d$. Because all faces of C_d are also cubes and all facets are in the same orbit under symmetry, the cube is a prime example for the application of the Incidence Decomposition Method.

Furthermore, we look at polyhedra associated with the geometry of numbers. We define the **contact polytopes** of $C(E_7)$ and $C(E_8)$ as the convex hull of the root system of the exceptional simple Lie algebras E_7 and E_8. The root system of E_7 consists of 126 vectors in dimension 7, the one of E_8 has 240 vectors in dimension 8. These vectors also span the associated integer lattices E_7 and E_8. For these lattices we can compute its so called **Voronoi cone**, which, for instance, plays an important role in the classification of perfect quadratic forms. For more information about this topic the interested reader may consider [Sch09] and [DSV07]. In the latter work the description conversion up to symmetry of the Voronoi cone $V(E_8)$ of E_8 was necessary to classify all perfect quadratic forms in dimension 8. As the computations with $V(E_8)$ using the Adjacency Decomposition Method took over one year, this cone is not suitable for experiments in this thesis, but we may look at its smaller cousins, the Voronoi cones of E_6 and E_7. The cone $V(E_6)$ is spanned by 36 vectors in dimension 21, the cone $V(E_7)$ is generated by 63 vectors in dimension 28.

Besides these rather geometric instances, we look at some polytopes from combinatorics. For $n \in \mathbb{N}, n \geq 3$ the **metric polytope** met_n is given by $4\binom{n}{3}$ inequalities in dimension $\binom{n}{2}$. To define the cut polytope we need one term from graph theory. Let $G = (V, E)$ be a connected graph. A **cut** of G is a subset $S \subseteq E$ such that $(V, E \setminus S)$ contains exactly two disjoint subgraphs of G. Let K_n be the complete undirected graph with n vertices and $\binom{n}{2}$ edges. We can identify a cut S of K_n with its incidence vector $\chi(S) \in \{0,1\}^{\binom{n}{2}}$ where the component ij of $\chi(S)$ is 1 if and only if $ij \in S$. For $n \in \mathbb{N}, n \geq 3$ the **cut polytope**

cut_n is the convex hull of the incidence vectors of all the cuts of K_n. We also have that cut_n is $\binom{n}{2}$-dimensional. Moreover, the metric polytope is a relaxation of the cut polytope and we have $cut_n \subseteq met_n$. The symmetry of the cut and metric polytope was exploited in [DFPS01, DFMV03] to gain insights into its combinatorial structure, using the Incidence Decomposition Method. More polytopes from combinatorial optimization can be found on [SMA].

For the last class of polytopes in the test set, let $B \in \{0,1\}^{m\times n}$ be a 0/1-matrix. We say that B has **consecutive ones property (C1P)** if for every permutation of the columns of B the rows contain all 1-entries in consecutive order. We can regard every matrix in $\mathbb{R}^{m\times n}$ also naturally as a vector in $\mathbb{R}^{m\cdot n}$. This allows us to define the **consecutive ones polytope** $P^{m,n}_{\text{C1P}} \subset \mathbb{R}^{m\cdot n}$ as the convex hull of all $m \times n$ matrices with consecutive ones property. Note that the polytope $P^{m,n}_{\text{C1P}}$ has about 2^{mn} vertices. For some applications it is helpful to compute facets of consecutive ones polytopes (cf. [Osw01]).

In the next two sections we analyze experimental results for computing restricted symmetries and performing description conversion of these polyhedra.

4.2.2. Restricted symmetries

Based on our considerations from Section 2.3, we can compare three different configurations for computing restricted symmetries of polyhedra. First, using vertex-colored graph automorphisms and the well-known software `nauty` and `NTL`. Second, using matrix automorphisms and the author's own implementation based on `PermLib`. The third configuration via lattice automorphism requires further explanation.

The author is unaware of any implementation using the described lattice automorphism approach to compute symmetries of polyhedra. An own implementation would have gone beyond the scope of this thesis. Nevertheless, we may use the existing software [`AUTO`] for a proof of concept. The package `AUTO` computes lattice automorphisms only. To obtain symmetries of polyhedra from these, we have to compute a stabilizer in the lattice automorphism group, which is also a non-trivial task in general. For some polyhedra from the discussed test set, however, the stabilizer is easy to compute by taking the elementwise absolute value of the generating matrices. Before we discuss these cases, we compare `PermLib` and `nauty`/`NTL`.

Table 4.1 displays the running times for polyhedra from the test set. The fastest result is marked with a gray background. For all instances except the consecutive ones polytopes `nauty` is one magnitude faster than `PermLib`. A performance difference was to be expected and is likely to be caused by two things. First, the implementation is not very optimized for the problem but uses the generic partition backtrack framework of `PermLib`. Second, the algorithm used is very generic and does not make much use of the knowledge about the graph automorphism problem, which it still is.

Although there is a big relative difference between the running times of the two configurations, the absolute differences currently do not matter much. This is because the current implementation of `SymPol` computes symmetries only once at the beginning and this is still fast compared to the description conversion. When a double coset decomposition implementation is available, the situation has to be re-evaluated.

Table 4.1.: Running times in seconds for computing restricted symmetries with graph automorphisms

polytope	PermLib	NTL/nauty
C_{15}	0.08	0.02
C_{17}	0.14	0.03
$V(E_6)$	0.22	0.02
$V(E_7)$	1.10	0.06
$C(E_7)$	0.36	0.10
$C(E_8)$	1.37	0.19
met_5	0.06	0.00
met_6	0.45	0.03
met_7	2.74	0.12
cut_5	0.01	0.00
cut_6	0.10	0.01
cut_7	0.70	0.03
$P^{3,3}_{\text{C1P}}$	10.90	5.95
$P^{3,4}_{\text{C1P}}$	1874.08	8001.98
$P^{4,3}_{\text{C1P}}$	1921.34	21886.70

It remains unclear where the performance difference for $P^{3,4}_{\text{C1P}}$ and $P^{4,3}_{\text{C1P}}$ comes from. The conversion from edge-weighted to vertex-colored graph increases the number of vertices from about 3900 by a factor of 13 to about 50000. Virtually all the running time is spent in `nauty`, so probably the difference in the number of vertices causes the measurable performance difference.

From the absolute numbers we can see that computing $\mathrm{RAut}(P^{m,n}_{\text{C1P}})$ with a graph or matrix automorphism approach quickly gets infeasible because we have to invert a matrix with about 2^{2mn} entries and compute its automorphisms. Even for small parameter values like $m = 4$ and $n = 5$ the matrix is too large to fit into memory. Because $P^{m,n}_{\text{C1P}}$ admits an integer lattice representation, we can try to compute $\mathrm{RAut}(P^{m,n}_{\text{C1P}})$ with a lattice automorphism approach in dimension $m \cdot n$. As already stated above, not for all polyhedra from the test set the corresponding stabilizer in the lattice automorphism group can be computed easily. For the tested consecutive ones polytopes this stabilizer is trivially obtained. Thus we may compare the running times for these instances with the graph and matrix automorphism solution in Table 4.2.

From this table we can see that the lattice automorphism calculations based on `AUTO` are much faster than the other approaches discussed before. A detailed analysis of the running time also reveals that almost all the time is spent on computing the lattice gram matrix, i.e. summing all vectors up in $Q = \sum_{v \in V} vv^T$. The actual lattice automorphism computation takes still well below 50 milliseconds for $P^{4,5}_{\text{C1P}}$. This shows that there is at least one class of interesting polyhedra for which the lattice automorphism approach performs very well. An implementation beyond this ad hoc proof of concept may help to find symmetries in other relevant polyhedra with many vertices or facets in low dimension.

Table 4.2.: Running times in seconds for computing restricted symmetries with lattice automorphisms

polytope	$\|V\|$	dimension	PermLib	NTL/nauty	AUTO
$P^{3,3}_{\text{C1P}}$	506	9	10.90	5.95	0.01
$P^{3,4}_{\text{C1P}}$	3916	12	1874.08	8001.98	0.05
$P^{4,3}_{\text{C1P}}$	3940	12	1921.34	21886.70	0.06
$P^{3,5}_{\text{C1P}}$	29498	15	?[a]	?[a]	0.41
$P^{5,3}_{\text{C1P}}$	30218	15	?[a]	?[a]	0.42
$P^{4,5}_{\text{C1P}}$	747196	20	?[a]	?[a]	12.40

[a] not tested because graph/matrix is too big

4.2.3. Description conversion

In this section we compare several strategies for performing a description conversion up to symmetries on the polyhedra from the test set. As a reference all polyhedra were treated with the direct conversion method (cf. Algorithm 3.1) with both `lrs` and `cdd`. Because this thesis is not about a comparison of these two codes (for a case study see [ABS97]) always the code with the fastest direct results was used for decomposition strategies. As decomposition strategies we can think of three types: (recursive) ADM, (recursive) IDM and a combination of both techniques.

It turns out that most polyhedra from the test set are quite easy to solve so that recursive decomposition strategies produce more overhead than they help. Therefore Table 4.3 lists results for non-recursive ADM and IDM. Difficult instances with one recursion step (two levels of ADM in total) are marked separately. Besides these "pure" strategies the table also contains the running times for a mixed IDM/ADM strategy. For this, after one application of IDM one level of ADM was applied, so this is usually one recursion level more than the pure strategies use. A different one with ADM first and IDM second turned out to be always inferior on all tested instances, so it is not listed. Each test had a timeout of two days, which was hit for the larger instances with direct conversion. The timeout was not due to the final symmetry filtering part, which is usually fast, but occurred during the `lrs` or `cdd` run.

Table 4.3 shows the running times for the description conversion of all polyhedra from the test set with all discussed strategies. The best direct and best decomposition run is marked with a gray background, the absolute best run is printed in bold letters. Looking at the columns for the direct conversion times, we identify three rather difficult instances, $V(E_7), met_6, cut_7$. The rest could also be solved with `cdd` or `lrs` in less than five minutes.

For these easy instances the best results were almost always obtained with ADM. Applying this decomposition often improves the running time by one order of magnitude. A special case occurs with cubes where a combined IDM/ADM solution is best according to Table 4.3. However, the table entries are somewhat misleading as no recursive IDM has been included in the test setup. This imposes no restriction for most of the polyhedra from the test set because there are too many facet orbits on the first recursion level to yield an improvement by recursive IDM. For cubes this condition does not hold because for every cube face C_F there is only one orbit of facets of C_F up to symmetry. Thus a recursive IDM

Table 4.3.: Running times in seconds for description conversion

polytope	given as	direct (lrs)	direct (cdd)	ADM	IDM	IDM-ADM
C_{15}	$\mathcal{H}$	7.75	52.73	2.07	4.61	**1.73**
C_{17}	$\mathcal{H}$	39.90	866.52	9.93	23.42	**8.38**
$V(E_6)$	$\mathcal{V}$	40.97	150.43	**2.79**	23.71	5.16
$V(E_7)$	$\mathcal{V}$	?[a]	?[a]	**9891.55**[b]	?[a,c]	13574.20
$C(E_7)$	$\mathcal{V}$	2.52	1.36	**0.84**	2.06	1.98
$C(E_8)$	$\mathcal{V}$	56.20	52.64	2.82	3.80	**2.76**
met_5	$\mathcal{H}$	0.45	**0.10**	0.11	0.16	0.16
met_6	$\mathcal{H}$	772.92	3.57	**1.34**	1.56	1.53
met_7	$\mathcal{H}$	?[a]	?[a]	524.05[b]	12440.20	**284.69**
cut_5	$\mathcal{V}$	0.02	0.04	0.02	0.02	0.02
cut_6	$\mathcal{V}$	2.44	0.74	**0.36**	0.73	0.53
cut_7	$\mathcal{V}$	44600.00	30122.10	69.46[b]	10095.30	**43.37**
$P_{\text{C1P}}^{3,3}$	$\mathcal{V}$	253.05	12.64	**11.56**	91.97	17.23
$P_{\text{C1P}}^{3,4}$	$\mathcal{V}$	?[a]	317.73	**50.46**	32098.60	1191.67

[a] aborted after 172800 seconds (2 days)
[b] with two levels of ADM
[c] with two levels of IDM

should be particularly suitable for cubes. With $k-1$ recursion levels of IDM for C_k the running times for C_{15} and C_{17} can be lowered to 1.21 and 6.01 seconds, respectively. This bound cannot be decreased substantially in `SymPol` because of necessary group calculations.

For the three difficult instances $V(E_7), met_7, cut_7$ using a decomposition method affects the running time tremendously. Again we distinguish two cases: $V(E_7)$ on the one hand and met_7, cut_7 on the other. The Voronoi cone $V(E_7)$ could not be solved directly within two days. With one recursive ADM application, two levels in total, this problem can be solved in less than three hours. A combination of IDM and ADM was significantly slower and using IDM alone rendered the problem again unsolvable within the time limit. This happens because there are too many and too difficult facet orbits of $V(E_7)$ and its first recursion level.

The polyhedra met_7 and cut_7 also profit enormously from adjacency decomposition. But the conversion of these polyhedra can be performed even faster by combining IDM and ADM. The facet structure of met_7 and $(cut_7)^\Delta$ is such that there is only one facet up to symmetry (cf. the metric cone from Section 3.3.2). Thus an initial application of the IDM is able to improve performance.

Except the very easy polyhedra met_5 and cut_5 all polyhedra from the test set got a significant performance improvement by the Adjacency Decomposition Method. Successfully using the Incidence Decomposition Method needs more care as its running time depends on the number of facet orbits. For some polyhedra one (metric and cut polytopes) or more (cubes) facet "layers" can be cut off with the IDM because there is only one facet up to symmetry. After this reduction ADM may be employed to further reduce the running time. Other combinations were not useful for the polyhedra from the test set. This

is partly because the instances are rather easy and most them require only one level of decomposition. The other reason is that IDM is only helpful if there are only very few orbits of facets, which happens rarely during recursion. At least the last factor might me mitigated to some extent by exploiting more symmetries of subproblems, which the next section explains in more detail.

4.2.4. Symmetries of subproblems

When an Incidence or Adjacency Decomposition Method is applied to a polyhedron P, `SymPol` does not compute the restricted symmetries $\mathrm{RAut}(P')$ for a subproblem P'. Instead, it uses only $\mathrm{Stab}(G, P')$ as the symmetry group for P' where G is a known part of the symmetry group of P. If a subproblem P' is to be solved directly and without decomposition, then it does not matter whether its symmetry group used is $\mathrm{Stab}(G, P')$ or something bigger. For the direct method symmetry filtering is performed after a description conversion without symmetries so we cannot expect any improvements by larger groups. Thus the `SymPol` implementation does not suffer as long as zero or one recursion levels are used.

Since the available test set of polyhedra contains only one instance which needs a considerable amount of time with two recursion levels, an implementation of sub-problem symmetry computation was not attempted for `SymPol`. But we may evaluate the potential of using larger symmetry groups in an ad-hoc manner at the polyhedron $V(E_7)$.

The cone $V(E_7)$ took about 10000 seconds and two levels of ADM to be solved and thus is by a wide margin the most difficult polyhedron of the test set (cf. Table 4.3 on page 50). The restricted symmetry group $G := \mathrm{RAut}(V(E_7))$ has order $|G| = 1451520$. In the following we consider the 157 sub-problems which arise from the first ADM level. The symmetry group used for them is the only one to have a performance impact because the second level polyhedra are solved with the direct method.

For every first level sub-problem P' we can compare the order of the stabilizer $\mathrm{Stab}(G, P')$ with the order of the symmetries $\mathrm{RAut}(P')$. The stabilizers are often very small with an average of about 360 over all 157 sub-cones. More than two third have of these have order less than or equal to 4. The symmetry groups are several magnitudes larger with an average order of about 10^{18}.

Most of these cones are rather trivial but there is one particular sub-cone P_1 which is responsible for about two thirds of the total running time and is used to find almost all rays of $V(E_7)$. Its stabilizer $\mathrm{Stab}(G, P_1)$ has order 1920 and $\mathrm{RAut}(P_1)$ has order 61440. This difference is not as huge as the average difference. Nevertheless, using $\mathrm{RAut}(P_1)$ as a symmetry group the time needed for P_1 is reduced from 6560 to 250 seconds.

Although there currently is no way to exploit these larger symmetry group in `SymPol` to solve $V(E_7)$, we may get an estimate of the performance gain. An improved implementation for $V(E_7)$ would still need to solve the same 157 sub-cones. By solving these separately and adding up their running time we know roughly how fast an integrated solution could be. All 157 sub-cones were solved in 370 seconds. Transforming the sub-cone results from $\mathrm{RAut}(P')$- to $\mathrm{Stab}(G, P')$-symmetry was very fast when performed on random samples with the help of double coset composition in `GAP`. So an improved version of `SymPol` solving the original cone should not be much slower. This means that the total running time for $V(E_7)$ can be reduced further from about 10000 seconds with ADM to

about 400 seconds by computing and exploiting restricted symmetries of sub-problems. This result is in line with the experience of the authors of [DSV07] who state that computing symmetries of sub-problems was necessary to succeed a description conversion of $V(E_8)$.

5. Conclusion

5.1. Summary

In this thesis we have discussed different aspects of performing a description conversion of polyhedra using symmetries. We started with a look at the hierarchy of geometrical and combinatorial symmetries that we can look for at polyhedra. Because the task of a description conversion implies that only partial knowledge of a polyhedron is available, we could not expect to compute the full combinatorial symmetry group, which is the maximal symmetry group. Thus we had to settle for less and discussed restricted symmetries, which can be computed if either all rays and vertices or all facets are known.

We looked at two methods to compute restricted symmetries of polyhedra. The first, well-known method uses a transformation to a graph problem to compute polyhedral symmetries as graph automorphisms. We considered a reduction of the graph to a matrix automorphism problem, fitted into the partition backtracking framework described in the author's thesis [Reh10] and implementation [`PermLib`]. The second, novel method regards polyhedral symmetries as lattice automorphisms. This does not work in all cases, but we have seen problem classes which are infeasible for the graph-based approach and work well with the lattice-based approach in practice.

Knowing a subgroup of the combinatorial subgroup, we analyzed different algorithms to exploit these symmetries in a description conversion process. We discussed two decomposition schemes, the Adjacency Decomposition Method (ADM) and the Incidence Decomposition Method (IDM), which split a polyhedron into sub-polyhedra, leading to a recursive solution.

Although both methods have been used separately for some years, a combination of them seems not to have been attempted yet. The author's C++ implementation `SymPol` accompanying this thesis implements both techniques and allows to mix them. Furthermore, `SymPol` allows to compute restricted symmetries of the polyhedra with the presented graph/matrix-based approach.

We looked at several experiments with `SymPol` that compared the performance of different algorithms for computing symmetries and description conversion. The comparison between the author's graph automorphisms code (which is actually stated as a matrix automorphism code) in `PermLib` and the dedicated graph automorphism solution `nauty` revealed that the latter is on most instances quite a bit faster but curiously not on all. The proposed algorithm based on lattice automorphisms could not be tested on all instances because no complete implementation is available. Despite that, we saw a problem class, consecutive ones polytopes, which is infeasible for the graph-based algorithm and on which the new method proves its potential.

In the description conversion part, recursive application of either ADM or IDM exploiting symmetries improved the performance measurably. For some instances using symme-

tries decreased the running times by one or more orders of magnitude, otherwise hitting the time limit with conventional description conversion techniques. Whether ADM or IDM is more suitable depends on the polyhedron. The ADM generally performs very well, the IDM needs special conditions to work fast. In some cases a combination of using IDM on the first computational level and ADM on the second further increased performance over a pure ADM recursion strategy.

5.2. Outlook

`SymPol` has already proven to be effective for real-world computations. For instance, it helped the author to prove an open conjecture about the symmetry of permutation polytopes (cf. Appendix A). Moreover, Kumar uses it in his upcoming paper [Kum] to shorten an otherwise lengthy description conversion process. To make it an even more useful tool and to extend the possible application areas there is room for improvements in at least two directions.

As the computational experiments have shown, using the dedicated graph automorphism software `nauty` is often faster than the author's own implementation currently used in `SymPol`. The software `nauty` is not included in `SymPol` due to software license incompatibilities. If obtaining symmetries turns out to be a performance bottleneck in some instances, one could also try other graph automorphisms implementations with open source licenses like [`bliss`]. Other improvements that have not been implemented in `SymPol` yet include computing symmetries of sub-polyhedra and double coset decomposition to make use of them, and a banking feature to store difficult solved sub-problems. These features have already proven to be useful for huge problems (cf. [DSV07]).

Another line of further research and implementation is the lattice automorphism approach to compute symmetries of instances where they cannot be computed with a graph transformation. An implementation would need an extension of the author's `PermLib` for matrix groups and new code to cope with lattices. This effort seems worthwhile especially since an ad hoc construction in this thesis has been successful.

It would also be of interest to implement and analyze concepts from invariant theory. We have seen that the invariant ring has very nice theoretical properties, which perhaps could also lead to improvements in practice.

For other application areas it may be useful to investigate whether and how symmetries can be found on slightly inaccurate input. In computational biology, for instance, large polyhedra occur whose input are rounded floating point values (cf. [LP10]). Exploiting symmetries in those kind of polyhedra may also boost the range of feasibly computable instances.

A. Permutation Polytopes of Finite Abelian Permutation Groups

A.1. Permutation polytopes

Consider a finite permutation group $G \leq S_m$ and the canonical representation $r : G \to \mathrm{GL}(\mathbb{R}, m)$. That means $r(\sigma)$ for $\sigma \in G$ is an $m \times m$ matrix with exactly one 1 in each row and column: We set $r(\sigma)_{i,j} = 1$ if $\sigma(i) = j$ and $r(\sigma)_{i,j} = 0$ otherwise. We can regard these $m \times m$ matrices also as vectors in $\mathbb{R}^{m^2}$. Thus we define in accordance with [BHNP09] the **permutation polytope** $P(G)$ of G as

$$P(G) := \mathrm{conv}\{r(\sigma) \ : \ \sigma \in G\} \subset \mathbb{R}^{m^2}.$$

As an example we consider the cyclic group $C_m := \langle\sigma\rangle \leq S_m$ where $\sigma := (1\,2\,3\,\ldots\,m)$. For this group we have $P(C_m) = \mathrm{conv}\{r(\sigma^0), r(\sigma^1), \ldots, r(\sigma^{m-1})\}$. This polytope in dimension m^2 is actually a polytope in dimension m. To see this we note that the $m-1$ last rows of $r(\sigma^i)$ are linearly dependent on the first row due to the cyclic group structure. Denoting the first row of $r(\sigma^i)$ by $(x_1, x_2, x_3, \ldots, x_m)$, the k-th row is given by

$$(x_{m-k+2}, x_{m-k+3}, \ldots, x_m, x_1, x_2, x_3, \ldots, x_{m-k+1})$$

for $k \geq 2$. This holds because $\sigma(k) - \sigma(1) \equiv k - 1 \pmod m$ by definition of σ.

Let $\pi : \mathbb{R}^{m^2} \to \mathbb{R}^m$ be the projection onto the first row. We set $P'_m := \pi(P(C_m)) = \mathrm{conv}\{\pi(r(\sigma^0)), \ldots, \pi(r(\sigma^{m-1}))\} \subset \mathbb{R}^m$. Because of the linear dependence we can find a linear map $T : \mathbb{R}^m \to \mathbb{R}^{m^2}$ such that $T(P'_m) = P(C_m)$. Since T is an injective linear function, P'_m and $P(C_m)$ must have the same combinatorial structure. Because $\pi(r(\sigma^i))$ is the $(i+1)$-th unit normal vector in $\mathbb{R}^m$, the polytope P'_m is an $(m-1)$-simplex T_{m-1} in dimension m. Thus also $P(C_m) \cong T_{m-1}$.

As an extension we consider $\langle\tau\rangle \leq S_m$ for an arbitrary permutation τ. Let $\Omega := \{1, 2, \ldots, m\}$. We write $\mathrm{supp}(\tau) := \{x \in \Omega \ : \ \tau(x) \neq x\}$ for the **support** of $\tau \in S_m$. We can decompose τ into disjoint cycles $\kappa_1, \kappa_2, \ldots, \kappa_l$. Suppose that $\mathrm{supp}(\kappa_i)$ is element-wise smaller than $\mathrm{supp}(\kappa_j)$ for $i < j$ and $|\kappa_i| \geq 2$ for all i. Then $r(\tau)$ has a block-diagonal structure $r(\tau) = \mathrm{diag}(A_1, A_2, \ldots, A_l)$ with blocks $A_i \in \mathbb{R}^{|\kappa_i|}$. For our considerations of $P(\langle\tau\rangle)$ we can eliminate by projection all components of $r(\tau^i)$ off the diagonal, which are always 0. In each block we can project onto the first row as the remaining rows of each block are linearly dependent on it. By these projections we obtain a polytope P' in dimension $\sum_{i=1}^{l} |\kappa_i|$ that is combinatorically equivalent to $P(\langle\tau\rangle)$.

A.2. Abelian permutation groups

The main tool to work with finite Abelian groups is the well-known classification theorem, which is given in Theorem A.1. Proofs can be found in many algebra books, for instance, [Ros09, Ch. 7.2].

Theorem A.1 (Basis Theorem for Finite Abelian Groups). Every finite Abelian group G is isomorphic to a direct product of cyclic groups.

With the help of this classification result, we can prove the following theorem, which gives a positive answer to Conjecture 5.4 of [BHNP09, p. 450]. Remark A.3 gives a hint for why the symmetries of permutation groups are less obvious than one might think.

Theorem A.2. Let G be a finite Abelian permutation group with $|G| > 2$. Then the order of the affine automorphism group $\mathrm{Aut}(P(G))$ of $P(G)$ is larger than $|G|$.

Proof. We prove this theorem in several steps. First we check whether $|G| = 2^d$ for some d. In this case Lemma A.7 shows that $|\mathrm{Aut}(P(G))| > |G|$ and we are done.

Otherwise, we note that $|G|$ has a prime factor $p \geq 3$. We proceed by decomposing G into a direct product of cyclic groups using Lemma A.6. We can combine the bounds of the automorphism groups of the permutation polytopes of these factors to get a bound on $|\mathrm{Aut}(P(G))|$ by Lemma A.5.

At least one of these cyclic group factors H has order divisible by $p \geq 3$. For this H, Theorem A.4 implies $|\mathrm{Aut}(P(H))| > |H|$ because $p! > p$. Thus we have shown that $|\mathrm{Aut}(P(G))| > |G|$. □

Remark A.3. A permutation polytope depends on the representation of the group. We have $P(C_6) \cong T_5 \ncong T_1 \times T_2 \cong P(C_2 \times C_3)$ although $C_6 \cong C_2 \times C_3$ as groups.

Theorem A.4. Let $G \cong C_n$ be a finite cyclic permutation group of order n. Consider a factorization $n = \prod_{i=1}^{l} k_i$ where each k_i is a power of a prime p_i and $p_i \neq p_j$ for $i \neq j$. Then the order of the affine automorphism group $\mathrm{Aut}(P(G))$ of $P(G)$ is at least $\prod_{i=1}^{l} k_i!$.

Lemma A.5. Let $G = H_1 \times H_2$ be the direct product of two permutation groups H_1, H_2. Then $|\mathrm{Aut}(P(G))| \geq |\mathrm{Aut}(P(H_1))| \cdot |\mathrm{Aut}(P(H_2))|$.

Proof. For $G = H_1 \times H_2$ the canonical representation r of G decomposes into a direct sum $r = r_1 \oplus r_2$ of canonical representations r_1, r_2 of H_1, H_2. Let A_1 be an affine symmetry of $P(H_1)$ and A_2 be an affine symmetry of $P(H_2)$. Then $A_1 \oplus A_2$ is an affine symmetry of $P(G)$ and the bound on the automorphism group order follows. □

Lemma A.6. Let G be a finite Abelian group. Let $a \in G$ be an element with maximal order. Then $G = \langle a \rangle \times H$ for a subgroup $H \leq G$.

Proof. This follows from Theorem A.1. See, for instance, [Ros09, Ch. 7]. □

Lemma A.7. Let $G \cong (C_2)^d$ for some $d \in \mathbb{N}$. Then $P(G)$ is affinely isomorphic to a d-dimensional cube and $|\mathrm{Aut}(P(G))| = d!\, 2^d$.

Proof. [BHNP09, Cor. 3.6] shows that $P(G)$ is affinely isomorphic to a d-cube if $|G| = 2^d$. Thus also $|\mathrm{Aut}(P(G))| = d!\, 2^d$. □

A.3. Proof of Theorem A.4

A.3.1. Preliminaries

Before we prove Theorem A.4, we study permutation polytopes of conjugated groups. Let $G \leq S_m$ be a finite permutation group and $\tau \in S_m$ be a permutation. We consider $G^\tau := \{\tau^- \sigma \tau \ : \ \sigma \in G\}$, the conjugate of G under τ. We have $P(G^\tau) = \operatorname{conv}\{r(\tau^- \sigma \tau) \ : \ \sigma \in G\} = \operatorname{conv}\{r(\tau^-) r(\sigma) r(\tau) \ : \ \sigma \in G\}$. Thus for a fixed τ there exist bijective linear maps $S_\tau, T_\tau : \mathbb{R}^{m^2} \to \mathbb{R}^{m^2}$, corresponding to the matrix multiplication in $\mathrm{GL}(\mathbb{R}, m)$, with $P(G^\tau) = S_\tau P(G) T_\tau$. Hence $\operatorname{Aut}(P(G))$ and $\operatorname{Aut}(P(G^\tau))$ are conjugated as well.

This allows us to choose an appropriate setting for the proof. For the rest of the section let $G \cong C_n$ of order n, embedded in a symmetric group S_m for some m. Let $\sigma \in G$ be a generator of G of order n. We consider the decomposition of σ into non-trivial disjoint cycles $\kappa_1, \kappa_2, \ldots, \kappa_q$. Let $c_i := |\kappa_i|$ be the order of κ_i, which is the length of the cycle.

We now construct a permutation $\tau \in S_m$ such that the conjugate $\sigma^\tau := \tau^- \sigma \tau$ is in a canonical form. First, we find a τ such that $\operatorname{supp}(\sigma^\tau) = \{1, 2, \ldots, \sum_{i=1}^q c_i\}$. Second, we modify τ so that $\operatorname{supp}(\kappa_j^\tau) = \sum_{i=1}^{j-1} c_i + \{1, 2, \ldots, c_j\} =: K_j$, where we build the sum of the scalar and the set element-wise. Third, we order each cycle and modify τ such that

$$\kappa_j^\tau(k) = \begin{cases} \sum_{i=1}^{j-1} c_i + (1 + \lfloor k+1 \rfloor_{c_j}) & \text{if } k \in K_j \\ k & \text{otherwise} \end{cases}. \tag{A.1}$$

Here we write $\lfloor a \rfloor_m$ short for $a \bmod m$, the unique solution of $a \equiv x \pmod m$ for $x \in \{0, 1, \ldots, m-1\}$. Equation (A.1) is nothing but a formal way to state that each κ_j under conjugation with τ is a translate of the canonical cycle generator $(1\,2\,3\,\ldots\,c_{j-1}\,c_j)$.

For our considerations of the order of $\operatorname{Aut}(P(G))$ we may thus assume that for $G = \langle \sigma \rangle$ the generator σ consists of cycles κ_j with property (A.1) without loss of generality. As we have seen in the first section, it is enough to work with a projection of the full $P(G) \subset \mathbb{R}^{m^2}$ onto $\mathbb{R}^M$ with $M := \sum_i c_i$. In the following we thus only work with $P'(G) := \operatorname{conv}\{v_0, v_1, \ldots, v_{n-1}\} \subset \mathbb{R}^M$ where $v_i := s(\sigma^i)$ and s is a combination of the canonical representation r and a suitable projection.

By our assumption these vectors have a block structure. The first c_1 components of each v_i contain exactly one 1. After this first block, the next c_2 components of each v_i contain exactly one 1, and so on. For instance, for the cyclic group $\langle (1\,2\,3)(4\,5\,6\,7) \rangle \leq S_7$, which is isomorphic to C_{12}, we obtain $c_1 = 3$, $c_2 = 4$, and

$$\begin{aligned}
v_0 &= (\ 1\ 0\ 0\ \ 1\ 0\ 0\ 0\ )^T \\
v_1 &= (\ 0\ 1\ 0\ \ 0\ 1\ 0\ 0\ )^T \\
v_2 &= (\ 0\ 0\ 1\ \ 0\ 0\ 1\ 0\ )^T \\
v_3 &= (\ 1\ 0\ 0\ \ 0\ 0\ 0\ 1\ )^T \\
v_4 &= (\ 0\ 1\ 0\ \ 1\ 0\ 0\ 0\ )^T \\
v_5 &= (\ 0\ 0\ 1\ \ 0\ 1\ 0\ 0\ )^T \\
&\ \ \vdots \\
v_{10} &= (\ 0\ 1\ 0\ \ 0\ 0\ 1\ 0\ )^T. \\
v_{11} &= (\ 0\ 0\ 1\ \ 0\ 0\ 0\ 1\ )^T.
\end{aligned}$$

In the remainder of this section we will explicitly construct symmetries of $P'(G)$ for arbitrary n.

A.3.2. Finding symmetries

Our goal is to find symmetries of the permutation polytope of a cyclic group $G = \langle\sigma\rangle$ with $|G| = n$. Let $n = \prod_{i=1}^{l} k_i$ be a factorization where each k_i is a power of a prime p_i and $p_i \neq p_j$ for $i \neq j$. The reason to look at this decomposition is that $G \cong C_{k_1} \times C_{k_2} \times \cdots \times C_{k_l}$ by Theorem A.1 and further factoring k_i leads to another, non-isomorphic group.

Lemma A.8. For the cycles of length $c_1, c_2, \ldots, c_q$ of which σ consists, we note that we must always have $\operatorname{lcm}(c_1, c_2, \ldots, c_q) = n$. Especially, $c_1, \ldots, c_q$ consist of the prime factors $p_1, \ldots, p_l$ that make up $k_1, \ldots, k_l$.

Proof. We know that $\operatorname{lcm}(c_1, c_2, \ldots, c_q) = |\sigma| = n$ because the cycles $\kappa_1, \ldots, \kappa_q$ are disjoint by our assumption. □

We choose an arbitrary $b \in \{1, \ldots, l\}$ and $t \in \{2, \ldots, k_b\}$. For these we define the permutation π of the set $\{0, 1, \ldots, n-1\}$ as follows:

$$\pi_{b,t}(r) = \begin{cases} \lfloor r + s \rfloor_n & \text{if } r \equiv 0 \pmod{k_b}, \\ \lfloor r - s \rfloor_n & \text{if } r \equiv t - 1 \pmod{k_b}, \\ r & \text{otherwise,} \end{cases} \tag{A.2}$$

where $s \in \{0, 1, \ldots, n-1\}$ is the unique solution of

$$\begin{aligned} s &\equiv t - 1 \pmod{k_b} \\ s &\equiv 0 \pmod{k_i} \quad \text{for } i \neq b. \end{aligned} \tag{A.3}$$

This unique solution s exists due to the well-known Chinese remainder theorem because the k_i have greatest common divisor 1 (cf. [CLRS09, Sec. 31.5]).

Lemma A.9. $\pi_{b,t}$ is a well-defined permutation of $\{0, 1, \ldots, n-1\}$.

Proof. To show that $\pi_{b,t}$ is bijective it suffices to show that it is injective, We observe that $r \equiv 0 \pmod{k_b}$ implies $\lfloor r + s \rfloor_n \equiv t - 1 \pmod{k_b}$ by construction of s in (A.3). Likewise $r \equiv t - 1 \pmod{k_b}$ implies $\lfloor r - s \rfloor_n \equiv 0 \pmod{k_b}$. We also know that $f_c(x) := \lfloor x + c \rfloor_n$ is bijective for $x \in \{0, \ldots, n-1\}$ Because either $r \equiv 0 \pmod{k_b}$ or $r \equiv t - 1 \pmod{k_b}$, the function $\pi_{b,t}$ is injective and thus a well-defined permutation of $\{0, 1, \ldots, n-1\}$. □

We can think of the $\pi_{b,t}$ as permutations of the set of vectors $v_0, \ldots, v_{n-1}$. In the following we will construct linear functions $A_{b,t}$ such that $A_{b,t} v_i = v_{\pi_{b,t}(i)}$ for every i. We will accomplish this by giving a coordinate permutation $\rho_{b,t}$ so that the j-th component of $v_{\pi_{b,t}(i)}$ is the $\rho_{b,t}(j)$-th component of v_i for every i and j. Such a permutation can naturally be realized by a linear function.

It is easier to construct $\rho_{b,t}$ block-wise. For every $1 \leq i \leq q$ we define $\rho_{b,t}^{(i)}$ by

$$\rho_{b,t}^{(i)}(j) = \begin{cases} \lfloor j + s \rfloor_{c_i} & \text{if } j \in R_0, \\ \lfloor j - s \rfloor_{c_i} & \text{if } j \in R_{t-1}, \\ j & \text{otherwise,} \end{cases} \tag{A.4}$$

where we use $R_x := \{\lfloor r \rfloor_{c_i} \ : \ r \in \{0, \dots, n-1\} \text{ and } r \equiv x \pmod{k_b}\}$. To make sense of this definition, we observe that v_r has a 1 in its i-th block only at position $\lfloor r \rfloor_{c_i}$. Hence R_x contains all possible positions of a 1 at rows with indices congruent x modulo k_b. The permutation $\rho_{b,t}^{(i)}$ shifts this position by s or $-s$ where necessary. We have to show two things: first, that $\rho_{b,t}^{(i)}$ is well-defined, and second, that $\rho_{b,t}^{(i)}$ actually realizes $\pi_{b,t}$.

Lemma A.10. $\rho_{b,t}^{(i)}$ as given in (A.4) is a well-defined permutation of $\{0, 1, \dots, c_i - 1\}$.

Proof. If $s \equiv 0 \pmod{c_i}$, we have nothing to show as $\rho_{b,t}$ is the identity. So let $\lfloor s \rfloor_{c_i} > 0$. It again suffices to show that $\rho_{b,t}^{(i)}$ is injective. We observe that $j \in R_0$ implies $\lfloor j + s \rfloor_{c_i} = \lfloor r + s \rfloor_{c_i}$ for some $r \equiv 0 \pmod{k_b}$. Thus $r + s \equiv t - 1 \pmod{k_b}$ and $\lfloor j + s \rfloor_{c_i} \in R_{t-1}$. Analogously, we see that $j \in R_{t-1}$ implies $\lfloor j - s \rfloor_{c_i} \in R_0$.

It remains to show that R_0 and R_{t-1} are disjoint for $t > 1$. If $j \in R_0 \cap R_{t-1}$, then there exist r, r' such that $\lfloor r' - r \rfloor_{c_i} = 0$ and $r' - r \equiv t - 1 \pmod{k_b}$. We have $p_b \mid c_i$ because otherwise $\lfloor s \rfloor_{c_i} = 0$ by construction of s in (A.3) and Lemma A.8, which states that c_i consists of some of the primes $p_1, \dots, p_l$. The first relation yields $p_b \mid r' - r$ and thus $t - 1$ has to be a multiple of p_b. This implies $s \equiv t - 1 \equiv 0 \pmod{k_b}$ because k_b is a power of p_b. This can only be true for $t = 1$. Hence R_0 and R_{t-1} are disjoint for $t > 1$ and $\rho_{b,t}^{(i)}$ is well-defined and injective. □

Lemma A.11. The permutation $\rho_{b,t}^{(i)}$ defined in (A.4) realizes $\pi_{b,t}$. That is,

$$\lfloor \pi_{b,t}(r) \rfloor_{c_i} = \rho_{b,t}^{(i)}(\lfloor r \rfloor_{c_i}) \tag{A.5}$$

for all rows $r \in \{0, \dots, n-1\}$ and all blocks $1 \leq i \leq q$.

Proof. The only cases we have to evaluate in detail are $r \equiv 0 \pmod{k_b}$ and $r \equiv t - 1 \pmod{k_b}$. In all other cases $\rho_{b,t}^{(i)}$ and $\pi_{b,t}$ are the identity and we have nothing to show.

First, let $r \equiv 0 \pmod{k_b}$. Then $\pi_{b,t}(r) = \lfloor r + s \rfloor_n$ and $\lfloor \pi_{b,t}(r) \rfloor_{c_i} = \lfloor r + s \rfloor_{c_i}$ because $c_i \mid n$. We also have $\rho_{b,t}^{(i)}(\lfloor r \rfloor_{c_i}) = \lfloor r + s \rfloor_{c_i}$, thus for this case (A.5) holds.

Second, let $r \equiv t - 1 \pmod{k_b}$. Analogously, we obtain $\rho_{b,t}^{(i)}(\lfloor r \rfloor_{c_i}) = \lfloor r - s \rfloor_{c_i} = \lfloor \pi_{b,t}(r) \rfloor_{c_i}$. Thus for all r and i equation (A.5) is satisfied. □

Corollary A.12. Let $\pi_{b,t}$ be defined as in (A.2). Every element of the group $\langle \{\pi_{b,t} \ : \ b \in \{1, \dots, l\}, t \in \{1, \dots, k_b\}\} \rangle$ corresponds to an affine automorphism of $P(G)$.

Proof. Lemma A.11 states in other words the following: Let $r \in \{0, \dots, n-1\}$ be an arbitrary row. The position of the 1 in each block i, given by $\lfloor r \rfloor_{c_i}$, is moved by $\rho_{b,t}^{(i)}$ on the same place as in row $\pi_{b,t}(r)$. Thus every permutation of the vectors $v_0, v_1, \dots, v_{n-1}$ by $\pi_{b,t}(r)$ can be realized as a linear transformation $A_{b,t}$. This $A_{b,t}$ permutes the components of its input according to $\rho_{b,t}^{(1)}, \dots, \rho_{b,t}^{(q)}$. Hence also every concatenation of $\pi_{b,t}(r)$ for different values of b and t is induced by a linear transformation. □

To finish the proof of Theorem A.4 we have to count the number of distinct automorphism we get from combinations of some $\pi_{b,t}$.

A.3.3. Counting symmetries

Lemma A.13. For fixed b the group $\langle \pi_{b,2}, \pi_{b,3}, \ldots, \pi_{b,k_b} \rangle$ is isomorphic to S_{k_b}.

Proof. Because $\pi_{b,t}(r) = \pi_{b,t}(\lfloor r \rfloor_{k_b})$ modulo k_b, it is enough to study $\lfloor \pi_{b,t}(r) \rfloor_{k_b}$ for $r \in \{0, 1, \ldots, k_b - 1\}$. We observe that $\lfloor \pi_{b,t}(r-1) \rfloor_{k_b} + 1$ is the transposition $(1\,t)$ on $r \in \{1, \ldots, k_b\}$. Thus $\langle \pi_{b,2}, \pi_{b,3}, \ldots, \pi_{b,k_b} \rangle \cong \langle (1\,2), (1\,3), \ldots, (1\,k_b) \rangle = S_{k_b}$. □

Lemma A.14. Let $H_b := \langle \pi_{b,2}, \pi_{b,3}, \ldots, \pi_{b,k_b} \rangle$. Then $|H_1 H_2 \cdots H_l| = \prod_{i=1}^{l} |H_i| = \prod_{i=1}^{l} k_i!$.

Proof. We show that $\pi_i, \pi_i' \in H_i$ and $\pi_1 \pi_2 \cdots \pi_l = \pi_1' \pi_2' \cdots \pi_l'$ implies $\pi_i = \pi_i'$ for every i. Together with Lemma A.13 this yields the stated size of $H_1 H_2 \cdots H_l$.

So let q be the first index such that $\pi_q \neq \pi_q'$. For $i \in \{1, \ldots, l\}$ let $s_i \in \{0, \ldots, n-1\}$ be the solution of

$$\begin{aligned} s_i &\equiv 1 \pmod{k_i} \\ s_i &\equiv 0 \pmod{k_j} \quad \text{for } j \neq i. \end{aligned} \tag{A.6}$$

Then every π_i moves each point r by $\lfloor \lambda_i(r) s_i \rfloor_n$ for some $\lambda_i(r) \in \mathbb{Z}$. Similarly, π_i' moves each point r by $\lfloor \mu_i(r) s_i \rfloor_n$ for some $\mu_i(r) \in \mathbb{Z}$. Thus for each r we have

$$\sum_{i=1}^{l} \lambda_i(r) s_i \equiv \sum_{i=1}^{l} \mu_i(r) s_i \pmod{n}. \tag{A.7}$$

By our assumption there exists one r such that not all $\lambda_i(r), \mu_i(r)$ are zero. We write $\lambda_i := \lambda_i(r)$ and $\mu_i := \mu_i(r)$ for this r.

Because of (A.7) we know that $k_q \mid \sum_{i=1}^{l} (\lambda_i - \mu_i) s_i$. We observe that $k_q \mid s_i$ for all $i \neq q$ and $k_q \nmid s_q$ by construction of s_q in (A.6). Hence $k_q \mid (\lambda_q - \mu_q)$. Consider the permutation $\pi_q \pi_q'^{-} \in H_q$. It moves r by $\lfloor (\lambda_q - \mu_q) s_q \rfloor_n$. Because $(\lambda_q - \mu_q) s_q$ is a multiple of $n = k_1 k_2 \ldots k_l$, it does not move r at all. Thus $\pi_q \pi_q'^{-}$ is the identity, which contradicts our assumption that $\pi_q \neq \pi_q'$. Hence we must have $\pi_i \neq \pi_i'$ for all i. □

Lemma A.14 together with Corollary A.12 yields the desired lower bound on the number of affine automorphisms of $P(G)$.

A.4. Conclusion

In this chapter we have proven a result on the symmetries of a special class of permutation polytopes. For permutation polytopes $P(G)$ of finite Abelian groups G we could show in Theorem A.2 that the group of affine automorphisms is larger than G for $|G| > 2$. We used the classification theorem for finite Abelian groups and a new, explicit lower bound on the number of affine automorphisms for $P(G)$ where G is cyclic (cf. Theorem A.4). This gives a positive answer to an open conjecture in [BHNP09]. Computations with `SymPol` suggest that the lower bound given in Theorem A.4 is probably sharp with a candidate being the direct product of C_{k_i} for pairwise coprime k_i.

B. SymPol Manual

B.1. Overview

SymPol computes restricted automorphisms of polyhedra and performs polyhedral description conversion up to a given or computed symmetry group.

This document will give you a step-by-step introduction to the usage of SymPol. You can find a very compact quick start guide in Section B.8 on page 68.

B.2. Compile and install

B.2.1. Software requirements

SymPol comes already bundled with patched versions of [cdd] and [lrs] to actually perform polyhedral representation conversion. It also contains a copy of [PermLib] for computations with permutations. To use and compile SymPol the following external software is required:

Parts of the [Boost] library are required by both PermLib and SymPol. Boost has to be installed in version 1.34.1 or higher with its program_options library. cdd, lrs and SymPol also make use of the arbitrary precision arithmetics library [GMP], both in its C and C++ version. Building SymPol (see also the next section) is most easily accomplished with the [CMake] configuration system.

So on a Debian/Ubuntu-based computer you would install the following packages:

- libboost-dev
- libboost-program-options-dev
- libgmp3-dev
- libgmpxx4ldbl (GMP C++ interface)
- cmake

B.2.2. Building SymPol

Compiling and installing

If CMake, Boost and GMP are installed, SymPol can be compiled as follows:

```
~/sympol$ mkdir build && cd build
~/sympol/build$ cmake -DCMAKE_BUILD_TYPE=Release  ..
~/sympol/build$ make
```

You can then use SymPol directly from the build directory and, for instance, print the command line help with

```
~/sympol/build$ ./sympol/sympol -h
SymPol v0.1 and PermLib 0.2 with lrs 4.2c and cddlib 0.94f
Allowed options:
  -h [ --help ]                         produce help message
 [...]
```

You can also build SymPol from any other directory by calling

```
cmake -DCMAKE_BUILD_TYPE=Release /path/to/sympol-source
```

If you want to install SymPol you can call `make install` as root. This will install SymPol into `/usr/local`. To choose a different prefix add an additional argument `-DCMAKE_INSTALL_PREFIX=/your/prefix/here` after the `-DCMAKE_BUILD_TYPE=Release`. On Linux systems you may have to call `ldconfig` as root afterwards so that the system knows about the new shared libraries (`cddgmp` and `lrsgmp`).

Assuming that the default installation directory is in your `$PATH`, you can check that SymPol is correctly installed and view the command line help by issuing

```
$ sympol -h
```

Optional libraries

SymPol can use [nauty] and [NTL] for a possibly faster computation of polyhedral symmetries, but for several reasons these are not bundled with SymPol but have to be compiled and installed separately (see section B.4). Assuming that you have downloaded and compiled both nauty and NTL, the following is required for use with SymPol.

- Copy `libntl.a` from the `NTL-src`-directory and the complete `include`-directory into `external/NTL` of SymPol.
- Please note that nauty is not open source and imposes usage restrictions. First you have to create a static library by `ar rcs libnauty.a naututil.o nauty.o nautil.o nautinv.o naugraph.o rng.o`. Then copy `libnauty.a` and `nauty.h` into `external/nauty` of SymPol.

If both NTL and nauty libraries are installed like this, run `cmake` from the build directory as described above.

B.3. Input format

SymPol mainly builds on the .ine/.ext file format as established by [cdd] and [lrs]. However, SymPol imposes one restriction and offers one extension to the format.

Regardless of the file format, the input filename has to be specified by the command-line argument `-i <filename>`. In most cases the `-i` can be omitted and ending the command with a filename suffices.

B.3.1. H-representation

A polyhedron in H-representation, i.e. given by m inequalities in $n-1$ variables, can be represented in an .ine file with the following format

```
H-representation
begin
m n rational
 { list of inequalities }
end
```

Every inequality is expected to be in the form $a_0+a_1x_1+a_2x_2+\cdots+a_{n-1}x_{n-1} \geq 0$. Such an inequality then is denoted in the .ine file as line $a_0\ a_1\ a_2\ \ldots\ a_{n-1}$. So, for instance, a two-dimensional triangle defined by

$$\begin{array}{rcrcl} x_1 & & & \geq & 0 \\ & & x_2 & \geq & 0 \\ x_1 & + & x_2 & \leq & 1 \end{array}$$

can be represented by the file

```
* 2-dim triangle
H-representation
begin
3 3 rational
 0  1  0
 0  0  1
 1 -1 -1
end
```

Here, the first line denotes a comment, introduced by a starting '*' character.

B.3.2. V-representation

Similarly, if a polytope is given by m rays and vertices in dimension $n-1$, it can be represented by an .ext file according to

```
V-representation
begin
m n rational
 { list of vertices }
 { list of rays }
end
```

Every vertex v gets a line $1\ v_1\ v_2\ \ldots\ v_{n-1}$ and every ray r a line $0\ r_1\ r_2\ \ldots\ r_{n-1}$.

B.3.3. Differences to cdd and lrs

As extension to this basic format, `SymPol` allows to include the automorphism group of the polyhedron, or parts of it, into the file. The user may specify after the `end` of the H- or V-representation a permutation group section as follows:

```
...
end
permutation group
p
 { list of #p group generators }
q
 { #q base points separated by whitespace }
```

The p group generators are to be given in cycle form, where commas separate cycles. The value q may be set to zero if no base of the group is known. A definition of a group base and what it is good for is explained in [Ser03], [HEO05] or [Reh10]. So to denote a group $G = \langle (1\,3)(4\,5), (2\,6\,5) \rangle$ with two generators and no base one would write:

```
...
end
permutation group
2
 1 3,4 5
 2 6 5
0
```

If you want to use $1, 2, 4, 5$ as a (potentially partial) base the section should look like

```
...
end
permutation group
2
 1 3,4 5
 2 6 5
4
 1 2 4 5
```

In contrast to `cdd` and `lrs`, in `SymPol` every inequality (H-representation case) or vertex and ray (V-representation case) has to be in exactly one line. At least `lrs` tolerates parts of an inequality or vertex spread over multiple lines, which `SymPol` currently does not support. However, `SymPol` comes with a Perl script that converts an .ine or .ext file into a suitable format (see Section B.9).

B.4. Computing restricted automorphisms

If no or only a few automorphisms are known a priori, the user has the possibility to compute *restricted automorphisms* of a polyhedron. These automorphisms may not be the full (combinatorial) symmetry group of the polyhedron, but it can be computed without full knowledge of both descriptions. We refer to [BDS09] for further details. `SymPol` offers two different implementations. One is a straight-forward implementation using a standard matrix inversion algorithm (cf. [CLRS09, Ch. 28]) and `PermLib` to compute matrix automorphisms (cf. [Reh10] and Section 2.3 of this thesis).

The other one relies on [`NTL`] for matrix inversion over integers and [`nauty`] for graph automorphisms. This approach is usually a bit faster, but is included only as an compile-time option for two reasons: `nauty` is not released under a proper open source license, so it cannot be distributed with `SymPol`. The `NTL` library is GPL licensed but not easy to integrate into `SymPol` automatically. Section B.2.2 contains a description for the advanced user of how to activate this implementation.

To just compute and print the restricted automorphism group of a polyhedron use the command-line switch `-automorphisms-only`.

B.5. Description conversion

In order to perform a description conversion of a polyhedron `SymPol` offers several algorithms. One of the following has to be chosen at the command-line as an automatic strategy selection currently is work in progress.

B.5.1. Direct conversion

The most straightforward way is to compute the complete complementary description and filter up to symmetries afterwards. The user can choose between [`lrs`] and [`cdd`] to perform the description conversion task. One may also estimate the difficulty of a problem by using the estimation feature of `lrs`.

The command-line switch `-e` enables the estimation mode. In this mode only a difficulty estimation is made by `lrs` and the program exits. To perform a polyhedral representation conversion up to symmetry for “easy” polyhedra you can use the `-d` switch for direct conversion. What “easy” means is hard to specify but experiments so far suggest that polyhedra with an estimation of 40 or below are good for the direct conversion technique. More difficult problems should be treated with one of the recursive methods shown below. Note that the absolute value of the estimation depends on the speed of the computer on which the estimation is performed, so these numbers are not necessarily comparable between different machines.

You also can choose between `cdd` and `lrs` for the core polyhedral computations (difficulty estimation so far only by `lrs`). Both programs may behave quite differently on the same input so it may be worthwhile to try both options for difficult problems. By default `lrs` is used. To replace it with `cdd` use the `-cdd` command-line switch. The interface to `cdd` is still a bit experimental and may not work in all cases.

B.5.2. Recursive methods

More sophisticated algorithms are recursive Adjacency Decomposition Method (ADM) and Incidence Decomposition Method (IDM), described in [BDS09] and Section 3.3 of this thesis. If the estimation of a problem exceeds 40 one of the following methods should be used.

The ADM is selected by the command-line argument `-a <threshold>`. For problems and arising subproblems whose estimation is below the given threshold the representation conversion is performed directly. Problems whose difficulty exceeds the threshold are broken into smaller sub-problems with the Adjacency Decomposition Method.

For some polyhedra it seems to be advantageous to also use the Incidence Decomposition Method in combination with ADM. The `-idm-adm <thresholdIDM> <thresholdADM>` command-line argument selects this strategy. This works like the pure ADM strategy before with an ADM threshold with one addition: Before a problem is estimated by `lrs` a heuristic IDM metric is computed.

Problems with an IDM metric value below the given IDM threshold are treated with IDM. If they exceed this threshold `lrs` computes an estimation. Then based on the ADM threshold either ADM or direct conversion is applied. The IDM metric is still being developed but for the current implementation an IDM threshold of 10 is recommended. As mentioned above, for the ADM threshold a value of 40 seems reasonable.

Alternatively, the strategy can be made depending on the recursion level with the `-idm-adm-level <levelIDM> <levelADM>` command-line argument. This will use IDM for the first **levelIDM** levels (may be 0 to turn off IDM) and ADM up to level **levelADM**, if IDM is not used. After **levelADM** direct computation will be used. This strategy helps to avoid expensive difficulty estimations.

B.5.3. Examples

Direct conversion

```
sympol -d -i input-file
```

ADM

```
sympol -a 40 -i input-file
```

IDM and ADM combined

```
sympol --idm-adm 10 40 -i input-file

# use ADM for the first two recursion levels
sympol --idm-adm-level 0 2 -i input-file

# use IDM for the first, ADM for the second level
sympol --idm-adm-level 1 2 -i input-file
```

B.5.4. Memory usage and other parameters

The memory usage of SymPol is dominated by the number of orbit elements it is allowed to store in RAM. Storing orbits in RAM allows to decide fast whether a new vertex/ray is equivalent under symmetry to one computed before. If not all orbits can be stored a quite expensive calculation is started to check for equivalence [Reh10, Ch. 3]. Thus the memory limit for orbits, specified by the command-line argument `-conf-compute-orbit-limit <number>`, should be chosen as large as possible. The default value 1024 means that SymPol will pre-compute orbits as long as it occupies less than 1024 megabytes of RAM. If this memory limit is exceeded vertex/ray equivalence will be computed by other means.

B.5.5. Combinatorial features

SymPol can compute the adjacency graph of the description conversion result, up to the used symmetries. Construction of the adjacency graph requires the use of the ADM at least at the first recursion level. In this case the command-line option `-adjacencies` activates the adjacency graph computation. The adjacency graph is printed in a format suitable for visualization with [Graphv]. The vertex numbers correspond to the position of the vertex/facet in the output list above.

Example

```
sympol --adjacencies --idm-adm-level 0 1 -i input-file
```

If you copy the adjacency part of the output into a textfile `adjacencies.dot` and have [Graphv] installed you can generate, for instance, a PNG graphic `adjacencies.png` of the graph by

```
neato -Tpng -o adjacencies.png adjacencies.dot
```

B.6. Output format

The output format follows the .ine/.ext format. The only difference is that the data section contains only rays/inequalities up to symmetry. The computed automorphisms group and the base used are printed, following the description in Section B.3.3.

B.7. Other program options

By default SymPol prints usage statistics about used processor time and RAM and warnings and errors. If you want a more verbose output you can specify the `-v` parameter, followed by a number which represents the logging level: `INFO` (1), `DEBUG` (2), `DEBUG2` through `DEBUG5` (3–6).

The command-line switch `-t` enables time measurement and prints the CPU time used at the end of the computation.

B.8. Quick Start Guide

Install If [CMake], [Boost] and [GMP] are installed:

```
~/sympol$ mkdir build && cd build
~/sympol/build$ cmake -DCMAKE_BUILD_TYPE=Release  ..
~/sympol/build$ make
# as root or in su/sudo shell
~/sympol/build$ make install
```

Then the SymPol binary will be installed in /usr/local/bin. On Linux systems you may have to call ldconfig as root afterwards so that the system knows about the new shared libraries (cddgmp and lrsgmp).

Compute the restricted automorphism group

```
sympol --automorphisms-only input-file
```

Estimate the difficulty of a representation conversion

```
sympol -e input-file
```

Do a representation conversion For "easy" input (see estimation, probably estimation below 40), try:

```
sympol -d input-file
```

For "difficult" input, try one or more layers of ADM

```
sympol --idm-adm-level 0 1 -i input-file
```

Compute the adjacency graph after conversion

```
sympol --idm-adm-level 0 1 --adjacencies -i input-file
```

If you copy the adjacency part of the output into a textfile adjacencies.dot and have [Graphv] installed you can generate, for instance, a PNG graphic adjacencies.png of the graph by

```
neato -Tpng -o adjacencies.png adjacencies.dot
```

B.9. Directory overview

After you extract a SymPol distribution package, you will find the following directories:

- contrib contains scripts to manipulate .ine/.ext files.
- data contains various example polyhedra in .ine/.ext files.
- external contains all third-party software used by SymPol: [cdd], [lrs], [PermLib].
- sympol contains the source code of the SymPol application

B.10. License

As both `cdd` and `lrs` are published under GPL2, `SymPol` is also available under GPL2. The complete text of the license is available from `http://www.gnu.org/licenses/gpl-2.0.html`.

Parts of the source code come with a more liberal license: The author's PermLib is BSD licensed.

Nomenclature

$\operatorname{aff} V$ affine hull of V, page 2

$\operatorname{cone} V$ conical hull of V, page 2

$\operatorname{conv} V$ convex hull of V, page 2

$\lfloor a \rfloor_m$ the rest of a modulo m, short for $a \bmod m$, page 57

$\operatorname{homog}(P)$ homogenization of a polytope P, page 3

g^- inverse of $g \in G$, page 5

$|G|$ order of group G, page 5

$\operatorname{OP}(\Omega)$ set of all ordered partitions of Ω, page 10

$\Pi \wedge \Sigma$ intersection of two partitions Π, Σ, page 11

$\operatorname{RAut}(P)$ group of restricted symmetries of a polyhedron P, page 13

$\operatorname{Stab}(G, P)$ setwise stabilizer of a set P in the group G, page 39

$\operatorname{supp} \tau$ support of the permutation τ, page 55

G^τ conjugate of the group G with permutation τ, page 57

$G^{[i]}$ pointwise stabilizer of the $i-1$ first base elements, page 9

P^Δ polar of the polytope P, page 4

x^G orbit of $x \in \Omega$ under $g \in G$, page 5

References

Bibliography

[ABCC06] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The traveling salesman problem*. Princeton Series in Applied Mathematics. Princeton University Press, Princeton, NJ, 2006. A computational study.

[ABS97] David Avis, David Bremner, and Raimund Seidel. How good are convex hull algorithms? *Computational Geometry*, 7:265–301, Apr. 1997.

[AD00] David Avis and Luc Devroye. Estimating the number of vertices of a polyhedron. *Information Processing Letters*, 73(3–4):137–143, 2000.

[AF92] David Avis and Komei Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete and Computational Geometry*, 8(1):295–313, Dec 1992.

[Avi00] David Avis. lrs: A revised implementation of the reverse search vertex enumeration algorithm. In Gil Kalai and Günter M. Ziegler, editors, *Polytopes – Combinatorics and Computation*, volume 29 of *DMV Seminar*, pages 177–198. Birkhäuser Verlag, 2000.

[Bal61] Michael L. Balinski. On the graph structure of convex polyhedra in n-space. *Pacific Journal of Mathematics*, 11(2):431–434, 1961.

[BDS09] 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.

[BEK84] Jürgen Bokowski, Günter Ewald, and Peter Kleinschmidt. On combinatorial and affine automorphisms of polytopes. *Israel Journal of Mathematics*, 47(2–3):123–130, Dec 1984.

[BFM98] David Bremner, Komei Fukuda, and Ambros Marzetta. Primal-dual methods for vertex and facet enumeration. *Discrete and Computational Geometry*, 20(3):333–357, 1998.

[BHNP09] Barbara Baumeister, Christian Haase, Benjamin Nill, and Andreas Paffenholz. On permutation polytopes. *Adv. Math.*, 222(2):431–452, 2009.

[BL98] Michael R. Bussieck and Marco E. Lübbecke. The vertex set of a 0/1-polytope is strongly P-enumerable. *Computational Geometry*, 11(2):103–109, 1998.

[Bre99] David Bremner. Incremental convex hull algorithms are not output sensitive. *Discrete and Computational Geometry*, 21(1):57–68, 1999.

[BW98] Thomas Becker and Volker Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Graduate Texts in Mathematics. Springer, 1998.

[CK70] Donald R. Chand and Sham S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 17(1):78–86, 1970.

[CLO08] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms*. Undergraduate Texts in Mathematics. Springer, third edition, 2008.

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

[Cox73] Harold S. M. Coxeter. *Regular Polytopes*. Dover Publications, third edition, 1973.

[CR01] Thomas Christof and Gerhard Reinelt. Decomposition and parallelization techniques for enumerating the facets of combinatorial polytopes. *International Journal of Computational Geometry and Applications*, 11(4):423–437, 2001.

[DFMV03] Antoine Deza, Komei Fukuda, Tomohiko Mizutani, and Cong Vo. On the face lattice of the metric polytope. In *Discrete and Computational Geometry*, volume 2866 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[DFPS01] Antoine Deza, Komei Fukuda, Dmitrii V. Pasechnik, and Masanori Sato. On the skeleton of the metric polytope. In *Discrete and Computational Geometry*, volume 2098 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 2001.

[DK02] Harm Derksen and Gregor Kemper. *Computational Invariant Theory*. Encyclopaedia of Mathematical Sciences. Springer, 2002.

[DSV07] Mathieu Dutour Sikiric, Achill Schürmann, and Frank Vallentin. Classification of eight-dimensional perfect forms. *Electronic Research Announcements of the American Mathematical Society*, 13:21–32, 2007.

[Dye83] Martin E. Dyer. The Complexity of Vertex Enumeration Methods. *Mathematics of Operations Research*, 8(3):381–402, 1983.

[Fio01] Samuel Fiorini. Determining the automorphism group of the linear ordering polytope. *Discrete Applied Mathematics*, 112(1–3):121–128, 2001.

[FLM97] Komei Fukuda, Thomas M. Liebling, and François Margot. Analysis of backtrack algorithms for listing all vertices and all faces of a convex polyhedron. *Computational Geometry*, 8(1):1–12, June 1997.

[FP85] U. Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):463–471, 1985.

[FP96] Komei Fukuda and Alain Prodon. Double description method revisited. In *Selected papers from the 8th Franco-Japanese and 4th Franco-Chinese Conference on Combinatorics and Computer Science*, volume 1120 of *Lecture Notes In Computer Science*, pages 91–111, London, UK, 1996. Springer.

[GLS93] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, second corrected edition edition, 1993.

[GO04] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. Discrete Mathematics and Applications. Chap-

man & Hall/CRC, second edition, 2004.

[Gri92] Viatcheslav P. Grishukhin. Computing extreme rays of the metric cone for seven points. *European Journal of Combinatorics*, 13(3):153–165, 1992.

[Had40] Hugo Hadwiger. Über ausgezeichnete Vektorsterne und reguläre Polytope. *Commentarii Mathematici Helvetici*, 13:90–107, 1940.

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

[KBB+06] Leonid Khachiyan, Endre Boros, Konrad Borys, Khaled Elbassioni, and Vladimir Gurvich. Generating all vertices of a polyhedron is hard. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 758–765, New York, NY, USA, 2006. ACM.

[Kem99] Gregor Kemper. An algorithm to calculate optimal homogeneous systems of parameters. *Journal of Symbolic Computation*, 27:171–184, 1999.

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

[KS03] Volker Kaibel and Alexander Schwartz. On the complexity of polytope isomorphism problems. *Graphs and Combinatorics*, 19(2):215–230, 2003.

[Kum] Abhinav Kumar. Elliptic fibrations on a generic jacobian kummer surface. In preparation.

[KW10] Volker Kaibel and Arnold Waßmer. Automorphism groups of cyclic polytopes. In Frank H. Lutz, editor, *Triangulated Manifolds*, chapter 8. Springer, 2010. To appear.

[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.

[Lin04] Steve Linton. Finding the smallest image of a set. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 229–234. ACM, 2004.

[LP10] Francisco Llaneras and Jesás Picó. Which metabolic pathways generate and characterize the flux space? A comparison among elementary modes, extreme pathways and minimal generators. *Journal of Biomedicine and Biotechnology*, 2010:753904, 2010.

[MRTT53] Theodore S. Motzkin, Howard Raiffa, Gerald L. Thompson, and Robert M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games, Vol. II*, volume 28 of *Annals of Mathematical Studies*, pages 81–103. Princeton University Press, 1953.

[MS02] Peter McMullen and Egon Schulte. *Abstract regular polytopes*, volume 92 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2002.

[Neu07] Mara D. Neusel. *Invariant theory*. Student Mathematical Library. AMS, Rhode Island, NJ, 2007.

[Osw01] Marcus Oswald. *Weighted Consecutive Ones Problems*. PhD thesis, Universität Heidelberg, 2001.

[PS97] Wilhelm Plesken and Bernd Souvignier. Computing isometries of lattices. *Journal of Symbolic Computation*, 24:327–334, 1997.

[Reh10] Thomas Rehn. Fundamental Permutation Group Algorithms for Symmetry Computation. Diploma thesis (computer science), Otto von Guericke University Magdeburg, February 2010.

[Rob84] Stewart A. Robertson. *Polytopes and Symmetry*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1984.

[Ros09] Harvey E. Rose. *A Course on Finite Groups*. Universitext. Springer, 2009.

[San10] Francisco Santos. A counterexample to the hirsch conjecture. Preprint available at `http://arxiv.org/abs/1006.2814`, Jun 2010.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 1998.

[Sch09] Achill Schürmann. *Computational Geometry of Positive Definite Quadratic Forms*, volume 48 of *University Lecture Series*. AMS, 2009.

[Sei04] Raimund Seidel. *Convex hull computations*, chapter 22. In Goodman and O'Rourke [GO04], second edition, 2004.

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

[SMA] SMAPO, a library of linear descriptions of low-dimensional 0/1-polytopes. [Online; accessed October 14, 2010], `http://comopt.ifi.uni-heidelberg.de/software/SMAPO/`.

[Stu08] Bernd Sturmfels. *Algorithms in Invariant Theory*. Springer, second edition, 2008.

[Thi01] Nicolas M. Thiéry. Computing minimal generating sets of invariant rings of permutation groups with sagbi-gröbner basis. In *Discrete Mathematics and Theoretical Computer Science Proceedings*, pages 315–328, 2001.

[Zie95] Günter M. Ziegler. *Lectures on Polytopes*. Graduate Texts in Mathematics. Springer, 1995.

Software

[AUTO] The automorphism program AUTO by B. Souvignier.

[bliss] bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs by T. Junttila and P. Kaski. `http://www.tcs.hut.fi/Software/bliss/`.

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

[cdd] cdd, cddplus and cddlib by K. Fukuda. `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html`.

[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/.

[GMP] GMP, The GNU Multiple Precision Arithmetic Library. http://gmplib.org/.

[Graphv] Graphviz – Graph Visualization Software. http://www.graphviz.org/.

[lrs] lrs by D. Avis. http://cgm.cs.mcgill.ca/~avis/C/lrs.html.

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

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

[NTL] NTL: A Library for doing Number Theory by V. Shoup. http://www.shoup.net/ntl/.

[PermLib] PermLib, a C++ library for permutation computations, by T. Rehn. http://www.mathematik.uni-rostock.de/lehrstuehle/geometrie/software/.

[Polyh] Polyhedral: A package for handling polytopes and lattices by M. Dutour Sikiric. http://www.liga.ens.fr/~dutour/Polyhedral/.

Index