permutalib/polyhedral: Advanced methods for lattice and group computations

Mathieu Dutour Sikirić

October 8, 2021

Introduction

- Over many years, I have worked on polyhedral computation using symmetries.
- The fields considered are lattice theory, optimization, topology, group theory, number theory.
- Most of the computations were done using **GAP**, but **GAP** has several limitations (slowness, large memory usage, threading limitation).
- Thus I decided to rewrite most of what I need in **C++**:
  - The code is completely available on github and I contribute daily to it.
  - It is Open Source and everyone can contribute.
  - Installations issues are addressed with **dockerfile** which allow easy installs.

`https://github.com/MathieuDutSik/permutalib`
`https://github.com/MathieuDutSik/polyhedral_common`
`https://hub.docker.com/r/mathieuds/polyhedralcpp`

Why **C++**?

C++ is well suited for high performance mathematical parallel computation

- The language has improved a lot since previous standards.
- Since **C++11**, there has been a strong push for making it simpler.
- Extremely fast computation are possible.
- Functional programming is partially possible with **std::function**.
- Many parallelization techniques are available (Threads, multiprocess, distributed, etc.)
- Templates allow to write abstract code and have tight abstrations, which is a very mathematical thing.

Interfacing with **C**, **Python**, **Rust**, **Julia**, **Java** is possible if the need arise.

I. Permutation Group Library

permutalib design goals

- For general groups, we have some algorithms but in general groups can be very wild. When algorithms exist, they invariably depend on some finite group subcalls.
- For finite groups, what we have is permutation groups and this is where we can effectively decide. **GAP** has implementation of most functionality we may need.
- Reimplementing all **GAP** permutation group algorithm is not needed.
- What we need for applications:
  - Computing the stabilizer of a set under a permutation group.
  - Testing if two sets are equivalent under a permutation group.
  - Finding the canonical form of the action of a group on sets.
  - Iterating over the all group elements.
- All of the above have been implemented in **C++** based on **GAP** code and the result is 10 to 100 times faster than **GAP**.
- There are other code by Christopher Jefferson (Vole) that provides functionality in Rust.

Orbit splitting and double cosets

- If we have an orbit xG for a group G, we often want to split the orbits for a subgroup H.
- We compute a double coset decomposition with G_x the stabilizer of x in G.

$$G = G_x g_1 H \cup \ldots \cup G_x g_p H \quad \text{and} \quad xG = \cup_j x g_j H$$

- So we need **DoubleCoset** decomposition:
  - A partial non-optimal solution is available by using the canonical form.
  - Implementing the **GAP** algorithm requires much more to be implemented: **Centre**, **Centralizer**, **Normalizer/Conjugator**, **RightCosets**, **AscendingChain**, etc.
  - There are heuristic questions to be considered. Does it make sense to store the double cosets? What about building the full orbit and then splitting it?

II. Graph algorithms and Canonical Form

Partition backtrack graph software

- Given a graph G the problem is to find the automorphism group of this graph.
- There are software based on Partition-Backtrack that allows to make this computation really fast.
- The field was started with **nauty** by Brendan McKay (around 10000 users) which is the first really fast program for such computation
- Better software is now **Traces** which is 10 times faster.
- We also have isomorphism check.
- The focus on graphs is because we can reduce almost all combinatorial structures to a graphical one (Edge weighted graphs, hypergraphs, cellular complexes, etc.)

Canonical form of graphs

- The partition backtrack programs such as **Traces** also compute a canonical form of the graph.
- In many cases (but not all), this allows to find a canonical form for the original object considered.
- This is useful for many reasons:
  - This allows to dispense from invariants.
  - Isomorphism tests becomes very easy.
  - We can compute the hash of an object.
  - From the hash we get easy partitioning of object class.
  - We can use standard data structures such as **std::unordered_set** which have very good performance guarantees.
- **Traces** does not provide isomorphism check but instead provides a canonical form computation that can then be compared for equality.

III. Canonical form of polytopes and lattices

Linear symmetry groups of a polyhedral cone

- Suppose C is a full-dimensional polyhedral cone generated by vectors $(v_1)_{1 \leq i \leq N}$ in $\mathbb{R}^n$.
- The linear symmetry group $Lin(C)$ is the group of transformations $\sigma \in \mathrm{Sym}(N)$ such that there exist $A \in \mathrm{GL}_n(\mathbb{R})$ with $Av_i = v_{\sigma(i)}$ (There are other groups related to polyhedral cones)
- Define the form

$$Q = \sum_{i=1}^{N} {}^t v_i v_i$$

- Define the edge colored graph $E(C)$ on N vertices with vertex and edge color

$$c_{ij} = v_i Q^{-1} {}^t v_j$$

- The automorphism group of the edge colored graph is $Lin(C)$.

Canonical form of a polyhedral cone, polytope

- By duplicating the colors of $E(C)$, we can reduce the vertex and edge colored graph to a classic graph $Cl(E(C))$ on which we can apply **Traces**.
- This allows to compute the automorphism group of the polyhedral cone and to compute the isomorphsim of polyhedral cones.
- For the canonical form one has to work harder:
  - First from the canonical form of the vertices of $G(E(C))$, we can get a canonical ordering of the vertices of $E(C)$.
  - From the canonical ordering of the vectors $(v_i)_{1 \leq i \leq N}$ we can get a standard basis $\mathcal{B}$.
  - We express the vectors v_i in the basis $\mathcal{B}$ and this gets us $Can((v_i)_{1 \leq i \leq N})$
- The form is canonical, that is $Can((v_i)_{1 \leq i \leq N}) = Can((Av_{\sigma(i)})_{1 \leq i \leq N})$.
- The strategy works effectively up to 30000 vertices.

Accelerating the computation

- There are 3 ways to accelerate the computation:
  - We can determine some subset of the vertices that will be preserved. Then compute the automorphism group of this subset and check if all symmetries will preserve the full polyhedral cone.
  - Another idea is to map the colors to another set of colors. Again check if it works.
  - Yet another idea is to compute for a subset and if the group is larger to compute the stabilizer of the subset of the remaining points.
- By using the first strategy, we can get to 100000 vertices.
- All such strategies allow to find a canonical form and compute the automorphism group.

Canonical form of lattices

- For a positive definite quadratic form A and $x \in \mathbb{Z}^n$, define $A[x] = xAx^T$. For $\lambda > 0$ we define

$$Min_\lambda(A) = \{x \in \mathbb{Z}^n \text{ s.t. } A[x] \leq \lambda\}$$

- Define λ_{min} the minimum λ such that Min_λ is full dimensional and span $\mathbb{Z}^n$.
- We define an edge weighted graph on $Min_{\lambda_{min}} = \{v_1, \ldots, v_N\}$ and $w_{ij} = v_i A v_j^T$.
- This allows to obtain an ordering of $Min_{\lambda_{min}}$.
- We apply the Hermite Normal Form in order to obtain a canonical form for $Min_{\lambda_{min}}$ and thus the matrix A.
- This is in many case faster than Minkowski reduction.

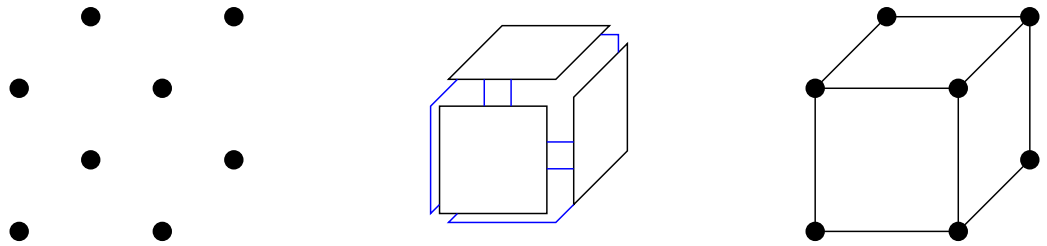
Integral linear groups

- Given a polytope P, we want to compute the integral symmetry group, that is $Lin_{\mathbb{Z}}(P)$.
- If P is spanned by $(v_i)_{1 \leq i \leq m}$ define $L' = \mathbb{Z}v_1 + \cdots + \mathbb{Z}v_m$. If $v_i \in \mathbb{Z}^n$ then $L' \subset L = \mathbb{Z}^n$.
- Define d the smallest integer such that $L' \subset L \subset \frac{1}{d}L'$.
- We compute $G = Lin(P)$. The group G becomes embedded in $\mathrm{GL}_n(\mathbb{Z}_d)$ and L corresponds to a subset of $(\mathbb{Z}_d)^n$.
- Thus the integral stabilizer becomes a problem of solving the set stabilizer.
- Similarly works for the isomorphism and canonical form (TODO).

IV. Dual description and face enumeration

Dual description problem

- Given a polytope P defined by vertices we want to find its facets



- The problem of going from the facets to the vertices is equivalent to this one by duality.
- The dual description problem is useful for many different kind of computations: Delaunay polytopes, perfect form, etc.
- Typically, the polytopes of interest are the one with a large symmetry group. We do not want the full set of facets, just representatives.
- n-dimensional Hypercube has $2n$ facets but 2^n vertices. Combinatorial explosion is to be expected in general.

The adjacency decomposition method

`Input`: The vertex-set of a polytope P and a group G acting on P.
`Output`: $\mathcal{O}$, the orbits of facets of P.

- Compute some initial facet F (by linear programming) and insert the corresponding orbit into $\mathcal{O}$ as `undone`.
- For every `undone` orbit O of facet:
  - Take a representative F of O.
  - Find the ridges contained in F, i.e. the facets of the facet F (this is a dual description computation).
  - For every ridge R, find the corresponding adjacent facet F' such that $R = F \cap F'$.
  - For every adjacent facet found test if the corresponding orbit is already present in $\mathcal{O}$. If no insert it as `undone`.
  - Mark the orbit O as `done`.
- Terminate when all orbits are `done`.

Reinvented many times (D. Jaquet 1993, T. Christof and G. Reinelt 1996).

The recursive adjacency method

In most cases the orbits of maximum incidence also have the highest symmetry and are the most difficult to compute.

- The computation of adjacent facets is a dual-description computation.
- So, the idea is to apply the Adjacency Decomposition method to those orbits as well.
- Based on information on the symmetry group, the incidence and the depth, we decide if we should spawn the adjacency method at a deeper level or apply directly cdd/ppl/lrs.

Issues:

- The number of cases to consider can grow dramatically.
- If one takes the stabilizer of a face, then the size of the groups involved may be too small to be efficient.

Program comparisons

Templatized code taking any coefficient type:

- lrs (*T_ring*): it iterates over all admissible basis in the simplex algorithm of linear programming
  - It is a tree search, no memory limitation, easy to parallelize.
  - Ideal if the polytope has a lot of vertices.
- cdd (*T_field*): it adds inequalities one after the other and maintain the double description throughout the computation
  - All vertices and facets are stored, memory limited, hard to parallelize.
  - Good performance if the polytope has degenerate vertices.

External program accessible from the code:

- pd (rational): We have a partial list of vertices, we compute the facets with lrs. If it does not coincide with $\mathcal{LF}$ then we can generate a missed vertex by linear programming.
  - It is a recommended method if there are less vertices than facets.
  - Bad performance for general polytopes.
- lrs/cdd/ppl (rational) Classical programs accessible from **polyhedral_common**.

Banking methods (or memoization)

- When one applies the Recursive Adjacency decomposition method, one needs to compute the dual description of faces.
- Some dual description may occur several times.
- The idea is to store the dual description of faces and when a dual description is needed to see if it has been already done. This is an ideal case for **std::unordered_map**. Still not without problem:
  - Need heuristic for when to decide to store or not.
  - Need heuristic when to check the sorage system or not.
  - Do we store for the full symmetry group or for the group that we had encountered?
  - We implement this either as a class or a client-server for multiprocessor cases.

Performance

- The cut polytope CUT_n is a classic polytope in combinatorial optimization. It has 2^n vertices, dimension $\frac{n(n-1)}{2}$ and $2^n n!$ symmetries.
- The last instance that can be computed easily on computer is CUT_8.
  - With the old **GAP** code: 2 days
  - With the **GAP** code using canonicalization: 90 minutes
  - With the newest **C++** code: 39 minutes.
  - On a recent laptop: 19 minutes.
- Most of the runtime is in the group library.
- For storing 2.10^8 facets the memory expense was 8G which is reasonable.

Face lattice computations

- Given a polytope of dimension d given by N vertices, we are interested in the faces of this polytope.
- The vertices are the faces of dimension 0 and the facets the faces of dimension $d-1$.
- For a d-dimensional simplex the number of faces of dimension i is $\binom{d+1}{i+1}$.
- Two different scenarios:
  - We are only interested in the faces of small dimension k (say $k = 1, 2, 3, \dots$). Then we can use linear programming and essentially we are limited only by the combinatorial explosion in those dimensions.
  - We want all dimensions. This includes the facets of course. Thus first step is to compute the facets and then to use the linear algebra (faster than linear programming) for deciding if a set is a facet. Much more intensive.
- The spanned faces are first canonicalized and then inserted into an unordered set.

V. Iso-edge domains in dimension six

Iso-edge domains

- The notion of C-types is a weakening of the notion of L-types in geometry of numbers where instead of taking into accounts all the faces of a Delaunay polytope, we use only the edges.
- The iso-edge domains form a tesselation of the cone $S^n_{>0}$ of positive definite quadratic forms.
- The iso-edge domain is encoded by a family of $2(2^n - 1)$ vectors.
- From this family of vectors, we can obtain the defining inequalities of the iso-edge domains.
- More details on
  - Mathieu Dutour Sikirić, Mario Kummer, *Iso Edge domains*, to appear in Expositiones Mathematicae **arxiv:2102.11139**

Enumeration in dimension six

- We need to enumerate configurations of 63 vectors in dimension six.
- We use the canonical form of such cones:
  - This allows to assign the iso-edge domains to one of the 20-processors canonically.
  - This allows to use hash tables.
- For the canonical form, instead of using the weight $w_{ij} = v_i A v_j^T$ we use $w_{ij} = \left| v_i A v_j^T \right|$. This reduce the graph to $2^n - 1$ vertices. We have to keep in mind that this may not work but we can handle failure cases gracefully.
- For the determination of possible switching to an adjacent domain, we use a nontrivial condition in dimension 3 to reduce the number of cases.
- Cdd was used for eliminating non-redundant inequality and we solved a bug in Clarkson method while doing so.
- It took 7 days on 20 processors and we got 55083358 types.

VI. Last points

Other functionalities

- The code for copositive programming is part of it **src_copos**.
- The implementation of Vinberg algorithm for computing the fundamental domain of hyperbolic Coxeter groups is **src_vinberg**
- The code for computing shortest vector configurations is in **src_short**
- The sparse solver using Approximate Message Passing is in **src_sparse_solver**
- There is also code for enumerating Delaunay polytopes in lattice as well as computing perfect forms.
- A lot of the work is motivated by the objective of computing the perfect forms in dimension 9, but this is still a work in progress.