GENERATING TUPLES OF INTEGERS MODULO THE ACTION OF A PERMUTATION GROUP AND APPLICATIONS

NICOLAS BORIE

Abstract. Originally motivated by algebraic invariant theory, we present an algorithm to enumerate integer vectors modulo the action of a permutation group. This problem generalizes the generation of unlabeled graph up to an isomorphism. In this paper, we present the full development of a generation engine by describing the related theory, establishing a mathematical and practical complexity, and exposing some benchmarks. We next show two applications to effective invariant theory and effective Galois theory.

Initialement motivé par la théorie algébrique des invariants, nous présentons une stratégie algorithmique pour énumérer les vecteurs d’entiers modulo l’action d’un groupe de permutations. Ce problème généralise le problème d’énumération des graphes non étiquetés. Dans cet article, nous développons un moteur complet d’énumération en expliquant la théorie sous-jacente, nous établissons des bornes de complexité pratiques et théoriques et exposons quelques bancs d’essais. Nous détaillons ensuite deux applications théoriques en théorie effective des invariants et en théorie de Galois effective.

Generation up to an Isomorphism, Enumerative Combinatorics, Computational Invariant Theory, Effective Galois Theory

1. Introduction

Let $G$ be a group of permutations, that is, a subgroup of some symmetric group $\mathfrak{S}_n$. Several problems in effective Galois theory (see [Gir87, Abd00]), computational commutative algebra (see [FR09, BTT11] [Bor11]) and generation of unlabeled with repetitions species of structures rely on the following computational building block.

Let $\mathbb{N}$ be the set of non-negative integers. An integer vector of length $n$ is an element of $\mathbb{N}^n$. The symmetric group $\mathfrak{S}_n$ acts on positions on integer vectors in $\mathbb{N}^n$: for $\sigma$ a permutation and $(v_1,\ldots,v_n)$ an integer vector,

$$\sigma.(v_1,\ldots,v_n) := (v_{\sigma^{-1}(1)},\ldots,v_{\sigma^{-1}(n)}).$$

This action coincides with the usual action of $\mathfrak{S}_n$ on monomials in the multivariate polynomial ring $K[x]$ with $K$ a field and $x := x_1,\ldots,x_n$ indeterminates.

Problem 1.1. Let $G \subseteq \mathfrak{S}_n$ be a permutation group. Enumerate the integer vectors of length $n$ modulo the action of $G$.

Note that there are infinitely many such vectors; in practice one usually wants to enumerate the vectors with a given sum or content.

For example, the Problem 1.1 contains the listing non-negative integer matrices with fixed sum up to the permutations of rows or columns appearing in the theory of multisymmetric functions [Ges87] [Mac04] and in the more recent investigations of multidiagonal coinvariant [Ber09] [BBT11].
Define the following equivalence relation over elements of $\mathbb{N}^n$: two vectors $\mathbf{u} := (a_1, \ldots, a_n)$ and $\mathbf{v} := (b_1, \ldots, b_n)$ are equivalent if there exists a permutation $\sigma \in G$ such that
\[ \sigma \cdot \mathbf{u} = (a_{\sigma^{-1}(1)}, \ldots, a_{\sigma^{-1}(n)}) = (b_1, \ldots, b_n) = \mathbf{v}. \]

Problem 1.1 consists in enumerating all $\mathbb{N}^n/G$ equivalence classes.

This problem is not well solved in the literature. Some applications present a greedy strategy searching and deleting all pairs of vectors such that the second part can be obtained from the first part. The most famous sub-problem is the unlabeled graph generation which consists in enumerate tuples over 0 and 1 of length $\binom{n}{2}$ enumerated up to the action of the symmetric groups acting on pair on nodes. This example has a very efficient implementation in Nauty which is able to enumerate all graphs over a small number of nodes.

The algorithms presented in this paper have been implemented, optimized, and intensively tested in Sage\cite{Sage}: most features are integrated in Sage since release 4.7 (2011-05-26, ticket \#6812, 1303 lines of code including documentation).

2. ORDERLY GENERATION AND TREE STRUCTURE OVER INTEGER VECTORS

The orderly strategy consists in setting a total order on objects before quotienting by the equivalence relation. This allows us to define a single representative by orbit. Using the lexicographic order on integer vectors, we will call a vector $\mathbf{v}$ canonical under the action of $G$ or just canonical if $\mathbf{v}$ is maximum in its orbit under $G$ for the lexicographic order:
\[ \mathbf{v} \text{ is canonical } \iff \mathbf{v} = \max_{\ell \in \mathbb{N}} \{ \sigma \cdot \mathbf{v} \mid \sigma \in G \}. \]

Now, the goal being to avoid to test systematically if vectors are canonical, we decided to use a tree structure on the objects in which we will get properties relaying the canonical vectors. Any result relating fathers, sons and the property of being canonical in the tree may allowed us to skip some canonical test.

2.1. Tree Structure over integer vectors. Let $\mathbf{r}$ be the vector $\mathbf{r} := (0, \ldots, 0)$ called root, we build a tree with the following function father.

**Definition 2.1.** Let $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ be a tuple of integers of length $n$ which is not the root. Let $1 \leq i \leq n$ be the position of the last non-zero entry of $\mathbf{a}$. We define the father of $\mathbf{a}$
\[ \text{father}(a_1, a_2, \ldots, a_i, 0, 0, \ldots, 0) := (a_1, a_2, \ldots, a_i - 1, 0, 0, \ldots, 0) \]

For any integer vector $\mathbf{v} = (a_1, \ldots, a_n)$, we can go back to the generation root $(0, \ldots, 0)$ by $\text{sum}(\mathbf{v}) := a_1 + \cdots + a_n$ steps. The corresponding application giving the children of an integer vector is thus:

**Definition 2.2.** Let $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ be a tuple of integers of length $n$. Let $1 \leq i \leq n$ be the position of the last non-zero entry of $\mathbf{a}$ ($i = 1$ if all entries are null). The set of children of $\mathbf{a}$ is obtained as:
\[ \text{children: } (a_1, a_2, \ldots, a_i, 0, 0, \ldots, 0) \mapsto \begin{cases} (a_1, a_2, \ldots, a_i + 1, 0, 0, \ldots, 0) \\ (a_1, a_2, \ldots, a_i, 1, 0, \ldots, 0) \\ (a_1, a_2, \ldots, a_i, 0, 1, \ldots, 0) \\ \vdots \\ (a_1, a_2, \ldots, a_i, 0, 0, \ldots, 1) \end{cases} \]
Proposition 2.3. For any permutation group $G \subset S_n$, for any integer vector $v$, if $v$ is not canonical under $G$, all children of $v$ are not canonical. Therefore, the canonicals form a "prefix tree" in the tree of all integer vectors.

**Sketch of proof:** When a father is not canonical, there exists a permutation such that the permuted vector is greater. Applying the same permutation on the children shows also it cannot be canonical.

![Enumeration tree of integer vectors modulo the action of $G = \langle (1, 2, 3) \rangle \subset S_3$, the cyclic group of degree 3.](image)

Figure 1. Enumeration tree of integer vectors modulo the action of $G = \langle (1, 2, 3) \rangle \subset S_3$, the cyclic group of degree 3.

2.2. Testing whether an integer vector is canonical. As we have seen, the fundamental operation for orderly generation is to test whether an integer vector is canonical; it is thus vital to optimize this operation. To this end, we use the work horse of computational group theory for permutation groups: stabilizer chains and strong generating sets.

Following the needs required by applications, we want to test massively if vectors are canonical or not. For this reason, we will use a strong generating system of the group $G$. We can compute this last item in almost linear time [Ser03] using GAP [GAP97].

Let $n$ a positive integer and $G$ a permutation group $G \subset S_n$. Recall that its stabilizer chain is $G_n = \{e\} \subset G_{n-1} \subset \cdots \subset G_1 \subset G_0 = G$, where

$$\forall i, 1 \leq i \leq n : G_i := \{g \in G| \forall j \leq i : g(j) = j\}.$$

From this chain, we build a strong generating system $T = \{T_1, T_2, \ldots, T_n\}$ where $T_i$ is a transversal of $G_{i-1}/G_i$. This set of strong generators is particularly adapted to the partial lexicographic order as stabilizers are defined with positions 1, 2, $\ldots$, $n$ from left to right.

Let $n$ and $i$ be two positive integers such that $1 \leq i \leq n$. For $v = (v_1, \ldots, v_n)$ and $w = (w_1, \ldots, w_n)$ two integer vectors of length $n$, let us define the following
binary relations

\[ v <_i w \iff (v_1, \ldots, v_i) <_{\text{lex}} (w_1, \ldots, w_i) \]

\[ v \leq_i w \iff (v_1, \ldots, v_i) \leq_{\text{lex}} (w_1, \ldots, w_i) \]

\[ v =_i w \iff \forall j, 1 \leq j \leq i : v_j = w_j \]

where \(<_{\text{lex}}\) and \(\leq_{\text{lex}}\) represent regular strict and large lexicographic comparison.

Algorithm 1 is a natural extension of McKay’s canonical graph labeling algorithm as it is explained in [HR09].

**Algorithm 1**

**Testing whether an integer vector is canonical**

**Arguments**

- \(v\): An integer vector of length \(n\);
- \(\text{sgs}(G)\): A strong generating set for \(G\), as a list \(\{T_1, \ldots, T_n\}\) of transversals.

```python
def is_canonical(v, sgs(G)):
    todo ← {v}
    for i ∈ \{1, 2, …, n\}:
        new_todo ← {}
        for w ∈ todo:
            children ← \{g · w | g ∈ T_i\}
            for child ∈ children:
                if v <_i child:
                    return False
                else:
                    if v =_i child and child / ∈ new_todo:
                        new_todo ← new_todo ∪ \{child\}
        todo ← new_todo
    return True
```

Algorithm 1 takes advantage of partial lexicographic orders and the strong generating system of the group \(G\). It tries to explore only a small part of the orbit of the vector \(v\); the worst case complexity of this step is bounded by the size of the orbit, and not by \(|G|\). In this sense, it does take into account the automorphism group of the vector \(v\).

**Proposition 2.4.** Let \(n\) be a positive integer and \(G\) a subgroup of \(\mathfrak{S}_n\). Let \(v\) be an integer vector of length \(n\). Algorithm 1 returns True if \(v\) is canonical under the action of \(G\) and returns False otherwise.

**Sketch of proof:** It is based on the properties of a strong generating system.

3. **Complexity**

3.1. **Theoretical complexity.**

3.1.1. **Efficiency of the tree structure.** Let \(n\) be a positive integer and \(G \subset \mathfrak{S}_n\) a permutation group. For any non negative integer \(d\), let \(C(d)\) (resp. \(\overline{C}(d)\)) be the number of canonical (resp. non canonical) integer vectors of degree \(d\). Based on the tree structure presented in Section 2.1, let \(T(n)\) (resp. \(\overline{T}(n)\)) the number of tested (resp. non tested) integer vectors.
Proposition 3.1. Generating all canonical integer vectors up to degree $d \geq 0$ using the generation strategy presented in Section 2 presents an absolute error bounded by $C(d)$. Equivalently, regarding the series, we have

$$\sum_{i=0}^{d} T(i) - \sum_{i=0}^{d} C(i) \leq C(d)$$

Sketch of proof: Using Lemma 2.3, we get this bound noticing two tested but non canonical vectors cannot have a paternity relation.

This absolute error is not very explicit (directly usable), but it can be used to get a relative error at the price of a rough approximation.

Corollary 3.2. Let $n$ and $b$ be two positive integers and $G \subset S_n$ a permutation group. Generating all canonical monomials under the action of $G$ up to degree $d$ using the generation strategy presented in Section 2 presents a relative error bounded by $\min\{\frac{n!|G|}{n+b}, n-1\}$.

Sketch of proof: We use the previous proposition with the fact that any integer vector has at least one child but no more than $n-1$ children (the generation root is the only one having $n$ children).

The bound is optimal for trivial groups ($\{e\} \subset S_n$), and seems to be better as the permutation group is of small cardinality. This relative error becomes better as we go up along the degree and tends to become optimal when the degree goes to infinity.

3.1.2. Complexity of testing if a vector is canonical. We now investigate the complexity of Algorithm [12]. We need first to select a reasonable statistic to collect, which will define the complexity of this algorithm.

The explosion appearing in the algorithm is conditioned by the size of the set $\text{new_todo}$. For $v$ an integer vector and $\{T_1, \ldots, T_n\}$ a strong generating system of a permutation $G$, when $i$ runs over $\{1, 2, \ldots, n\}$ in the main loop, the set $\text{new_todo}_i$ contains at step $i$:

$$\text{new_todo}_i = \{g_1 \cdots g_i \cdot v | g_1 \cdots g_i \cdot v =_{i} v, \forall j \leq i : g_j \in T_j\}$$

The right statistic to record is the size of the union of the $\text{new_todo}_i$ for all $i$ such that the algorithm is still running: that corresponds to the part of the orbit explored by the algorithm. This statistic appears to be very difficult to evaluate by a theoretical way. However, collecting it with a computer is a simple task.

3.1.3. Parallelization and memory complexity. Let us note that this generation engine is trivially amenable for parallelism: one can devote the study of each branch to a different processor. Our implementation uses a little framework SearchForest, co-developed by the author, for exploration trees and map-reduce operations on them. To get a parallel implementation, it is sufficient to use the drop-in parallel replacement for SearchForest under development by Jean-Baptiste Priez and Florent Hivert.

The memory complexity of the generation engine is reasonable, bounded by the size of the answer. Indeed, we keep in the cache only the Canonical vectors of degree $d-1$ when we search for those in degree $d$. In case one wants to only iterate through the elements of a given degree $d$, then this can be achieved with memory complexity $O(nd)$. 
3.2. **Benchmarks design.** To benchmark our implementation, we chose the following problem as test-case.

**Problem 3.3.** Let $n$ be a positive integer and $G \subseteq \mathfrak{S}_n$ a permutation group. Iterate through all the canonical integer vectors $v$ under the staircase (i.e. $v_i \leq n - i$).

A vector $v$ of length $n$ is said to be *under the staircase* when it satisfies $v \leq_{\text{lex}} (n-1, n-2, \ldots, 1, 0)$.

This problem contains essentially all difficulties that can appear. The family of $n!$ integer vectors under the staircase contains vectors with trivial automorphism group as well as vectors with a lot of symmetries. Applications also require to deal with this problem as the corresponding family of monomials plays a crucial role in algebra.

3.2.1. **Benchmarks for transitive permutation groups.** We now need a good family of permutation groups, representative of the practical use cases. We chose to use the database of all transitive groups of degree $\leq 30$ [Hul05] available in Sage through the system GAP [GAP97].

The benchmarks have been run on an off-the-shelf 2.40 GHz dual core Mac Book laptop running Ubuntu 12.4 and Sage version 5.3.

3.3. **Benchmarks.**

3.3.1. **Tree Structure over integer vectors.** This first benchmark investigates the efficiency of the tree structure presented in Section 2.1. As we don’t test children of non canonical integer vectors, one wants to take measures of the part of tested non canonical vectors (which corresponds to the useless part of computations). For that, we solve Problem 3.3 for each group of the database and we collect the following information as follows.

| Database Id. | $|G|$ | Index in $\mathfrak{S}_n$ | Canonicals | number of tests |
|--------------|-----|--------------------------|------------|----------------|
| 1            | 5   | 24                       | 71         | 81             |
| 2            | 10  | 12                       | 68         | 81             |
| 3            | 20  | 6                        | 46         | 67             |
| 4            | 60  | 2                        | 41         | 67             |
| 5            | 120 | 1                        | 41         | 67             |

This table displays the statistics for transitive groups of degree 5. *Database Id.* is the integer indexing the group, $|G|$ and *Index in $\mathfrak{S}_n$* are respectively the cardinality and the index of the group $G$ in the symmetric group $\mathfrak{S}_n$. *Canonicals* denotes the number of canonical vectors under the staircase and *number of tests* is the number of times the algorithm testing if an integer vector is canonical is called.

From this information, we set a quantity $\text{Err}$ defined as follows:

$$\text{Err} := \frac{\text{number of tests} - \text{Canonicals}}{\text{Canonicals}}.$$  

The following figure shows $\text{Err}$ depending on the index $\frac{n!}{|G|}$. The figure contains 166 crosses, one for each transitive group over at most 10 variables. We use a logarithmic scale on the x axis.
Empirical complexity of testing if a vector is canonical. Algorithm 1 needs to explore a part of the orbit of the tested integer vectors. The following table displays for each transitive group over 5 variables, the number of elements of all orbits of tested vectors solving Problem 3.3 compared to the total number of integer vectors explored.

Now we define Ratio to be the average size of the orbit needed to be explored to know if an integer vector is canonical:

$$\text{Ratio} := \frac{\text{total explored}}{\text{total orbits}}.$$  

The following figure plots Ratio in terms of $|G|$ for transitive groups on at most 9 variables.
3.3.3. Overall empirical complexity of the generation engine. We now evaluate the overall complexity by comparing the ratio between the computations and the size of the output. We define the measure Complexity as follows:

\[ \text{Complexity} := \frac{\text{total explored}}{\text{Canonicals}}. \]

The following graph displays Complexity in terms of the size of the group $|G|$ for transitive Groups on up to 9 variables (and excluding the alternate and symmetric group of degree 9).

The dashed line has as equation $y = 5\ln(|G|)$. Therefore, we get the following empirical overall complexity:

\[ \text{Computations} = O(\ln(|G|) \times \text{Output size}) \]
3.3.4. Tests around the unlabeled graph generation problem. Although the generation engine is not optimized for the unlabeled graph generation problem, we can apply our strategy on it.

Fix $n$, and consider the set $E$ of pairs of elements of $n$. The symmetric group $S_n$ acts on pairs by $\sigma \cdot (i, j) = (\sigma(i), \sigma(j))$ for $\sigma \in S_n$ and $(i, j) \in E$. Let $G$ be the induced group of permutations of $E$. A labeled graph can be identified with the integer vector with parts in $0, 1$. Then, two graphs are isomorphic if and only if the corresponding vectors are in the same $G$-orbit.

Now, one needs just to know which are these permutation groups acting on pairs of integers. In the following example, we retrieve the number of graphs on $n$ unlabeled nodes is, for small values of $n$ is given by: $1, 1, 2, 4, 11, 34, 156, 1044, 12346, 274668, 12005168, \ldots$

```
sage: L = [TransitiveGroup(1,1), TransitiveGroup(3,2), TransitiveGroup(6,6), TransitiveGroup(10,12), TransitiveGroup(15,28), TransitiveGroup(21,38), TransitiveGroup(28,502)]
sage: [IntegerVectorsModPermutationGroup(G,max_part=1).cardinality() for G in L]
[2, 4, 11, 34, 156, 1044, 12346]
```

Notice that our generation engine generalizes the graph generation problem in two directions. Removing the option `max_part`, one enumerates multigraphs (graphs with multiple edges between nodes). On the other hand, graphs correspond to special cases of permutation groups. From an algebraic point of view, we saw graphs as monomials whose exponents are 0 or 1, canonical for the action of the symmetric group on pairs of nodes.

4. Computing the invariants ring of a permutation group

Let us explain how the generation engine from Section 2 is plugged into effective invariant theory (see [DK02] and [Kin07]).

A well-known application to build an invariant polynomial under the action of a permutation group $G$ is the Reynolds operator $R$. From any polynomial $P$ in $n$ variables $x := x_1, x_2, \ldots, x_n$, the invariant is $$R(P) := \frac{1}{|G|} \sum_{\sigma \in G} \sigma \cdot P,$$
where $\sigma \cdot P$ is the polynomial built from $P$ for which $\sigma$ has permuted by position the tuple of variables $(x_1, x_2, \ldots, x_n)$. Formally, for any $\sigma \in G$
$$(\sigma \cdot P)(x_1, x_2, \ldots, x_n) := P(x_{\sigma^{-1}(1)}, x_{\sigma^{-1}(2)}, \ldots, x_{\sigma^{-1}(n)}).$$

For large groups, the Reynolds operator is not very convenient to build invariant polynomials. If $P$ is a monomial $x^a := x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}$ where $a = (a_1, a_2, \ldots, a_n)$, the minimal invariant one can build in number of terms is the orbit sum $\sum_{\text{Orb}(a)} (x^a)$ of $x$.

Let $\mathbb{K}$ a field, we denote by $\mathbb{K}[x]^G$ the ring formed by all polynomials invariant under the action of $G$.

$$\mathbb{K}[x]^G := \{ P \in \mathbb{K}[x] | \forall \sigma \in G : \sigma \cdot P = P \}.$$
For any subgroups $G$ of $S_n$ and $\mathbb{K}$ a field of characteristic 0, a result due to Hilbert and Noether state that the ring of invariant $\mathbb{K}[x]^G$ is a free module of rank $\frac{n!}{|G|}$ over the symmetric polynomials in the variable $x$. Computing the invariant ring $\mathbb{K}[x]^G$ consists essentially in building algorithmically an explicit family (called secondary invariant polynomials) of generators of this free module.

Searching the secondary invariant polynomials from orbit sum of monomials whose vector of exponents is canonical (instead of all monomials) produces a gain of complexity of $|G|$ if we assume that all orbits are of cardinality $|G|$. This assumption is obviously false; however, in practice, it seems to hold in average and up to a constant factor $[\text{Bor11}]$.

In [\text{BT11}], the authors calculate the secondary invariants of the 61st transitive group over 14 variables whose cardinality is 50803200. Using the canonicals monomials, they managed to build a family of 28 irreducible secondary invariants deploying a set of 1716 secondary invariants. This computation is unreachable by Gröbner basis techniques.

5. Computing primitive invariants for a permutation group

5.1. Introduction. We now apply our generation strategy to this problem concerning effective Galois theory.

Problem 5.1. Let $n$ a positive integer and $G$ a permutation group, subgroup of $S_n$. Let $\mathbb{K}$ be a field and $x := x_1, \ldots, x_n$ be $n$ formal variables. Find a polynomial $P \in \mathbb{K}[x_1, \ldots, x_n]$ such that

\[ \{ \sigma \in S_n | \sigma \cdot P = P \} = G. \]

A such polynomial is called a primitive invariant for $G$.

Problem 5.1 (exposed in [\text{Gir87}] and [\text{Abd00}]) consists in finding an invariant $P$ under the action of $G$ such that its stabilizer $\text{Stab}_{S_n}(P)$ in $S_n$ is equal to $G$ itself. Solving this problem becomes difficult when we want to construct a primitive invariant of degree minimal or a primitive invariant with a minimal number of terms.
Algorithm 2 Primitive invariant using stabilizer refinement

Prerequisites:
• `IntegerVectorsModPermgroup`: module to enumerate orbit representatives;
• `stabilizer_of_orbit_of(G, v)`: a function returning the permutation group which stabilizes the orbit of \( v \) under the action of the permutation group \( G \).

Arguments:
• \( G \): A permutation group, subgroup of \( \mathfrak{S}_n \).

```python
def minimal_primitive_invariant(G):
    cumulativeStab ← SymmetricGroup(degree(G))
    chain ← [[(0, 0, ..., 0), cumulativeStab, cumulativeStab]]
    if Cardinality(cumulativeStab) == Cardinality(G):
        return chain
    for v ∈ IntegerVectorsModPermgroup(G):
        AutV ← stabilizer_of_orbit_of(G, v)
        Intersect ← cumulativeStab ∩ AutV
        if Cardinality(Intersect) < Cardinality(cumulativeStab):
            chain ← chain ∪ [v, AutV, Intersect]
            cumulativeStab ← Intersect
        if Cardinality(cumulativeStab) == Cardinality(G):
            return chain
```

5.2. Primitive invariant of minimal degree.

5.3. Benchmarks. Algorithm 2 terminates in less than an hour for any subgroup of \( \mathfrak{S}_{10} \). Even, it can calculate some primitive invariants for a lot of subgroups with degree between 10 and 20 while the literature only provides examples up to degree 7 or 8. Using the same computer, this benchmark just collects the average time in seconds of execution of Algorithm 2 by executing systematically the algorithm on transitive groups of degree \( n \).

<table>
<thead>
<tr>
<th>Degree of Groups</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computations time</td>
<td>0.008</td>
<td>0.064</td>
<td>0.104</td>
<td>0.160</td>
<td>0.208</td>
<td>0.393</td>
<td>0.537</td>
<td>2.364</td>
<td>27.093</td>
</tr>
</tbody>
</table>

We would like to thanks Nicolas M. Thiéry, Simon A. King, Karl-Dieter Crisman and Dmitri V. Pasechnik for useful comments about implementation details, review of code and Cython optimizations.

This research was driven by computer exploration using the open-source mathematical software Sage [S+09]. In particular, we perused its algebraic combinatorics features developed by the Sage-Combinat community [SCc08], as well as its group theoretical features provided by GAP [GAP97].

References


Univ. Paris Est Marne-La-Vallée, Laboratoire d’Informatique Gaspard Monge, Cité Descartes, Bât Copernic – 5, bd Descartes Champs sur Marne 77454 Marne-la-Vallée Cedex 2, France