Discrete & Computational Geometry (2022) 67:1147–1173
https://doi.org/10.1007/s00454-021-00344-x

A New Face Iterator for Polyhedra and for More General Finite Locally Branched Lattices

Jonathan Kliem[1] · **Christian Stump**[2]

Received: 6 May 2020 / Revised: 6 April 2021 / Accepted: 16 April 2021 /
Published online: 18 March 2022
© The Author(s) 2022

Abstract

We discuss a new memory-efficient depth-first algorithm and its implementation that iterates over all elements of a finite locally branched lattice. This algorithm can be applied to face lattices of polyhedra and to various generalizations such as finite polyhedral complexes and subdivisions of manifolds, extended tight spans and closed sets of matroids. Its practical implementation is very fast compared to state-of-the-art implementations of previously considered algorithms. Based on recent work of Bruns, García-Sánchez, O'Neill, and Wilburne, we apply this algorithm to prove *Wilf's conjecture* for all numerical semigroups of multiplicity 19 by iterating through the faces of the *Kunz cone* and identifying the possible *bad faces* and then checking that these do not yield counterexamples to Wilf's conjecture.

Keywords Wilf's conjecture · Numerical Semigroup · Polyhedron · Face lattice · Face iterator · Enumeration · f-Vector

Mathematics Subject Classification 52B05 · 06A07 · 68Q25 · 05A15 · 05B35

Editor in Charge: Kenneth Clarkson

J.K. receives funding by the Deutsche Forschungsgemeinschaft DFG under Germany's Excellence Strategy - The Berlin Mathematics Research Center MATH+ (EXC-2046/1, project ID: 390685689). C.S. is supported by the DFG Heisenberg grant STU 563/{4-6}-1 "Noncrossing phenomena in Algebra and Geometry".

Jonathan Kliem
jonathan.kliem@fu-berlin.de

Christian Stump
christian.stump@rub.de

[1] Institut für Mathematik, Freie Universität Berlin, Arnimallee 2, 14195 Berlin, Germany

[2] Fakultät für Mathematik, Ruhr-Universität Bochum, Universitätsstraße 150, 44801 Bochum, Germany

1 Introduction

We call a finite lattice $(\mathscr{P}, \leq)$ *locally branched* if all intervals of length two contain at least four elements. We show that such lattices are atomic and coatomic and refer to Sect. 2 for details.

This paper describes a depth-first algorithm to iterate through the elements in a finite locally branched lattice given its coatoms, see Sect. 3. It moreover describes variants of this algorithm allowing the iteration over slightly more general posets. Examples of such locally branched lattices (or its mild generalizations) include face posets of

- polytopes and unbounded polyhedra,
- finite polytopal or polyhedral complexes,
- finite polyhedral subdivisions of manifolds,
- extended tight spans, and
- closed sets of matroids.

One may in addition compute all cover relations as discussed in Sect. 4.1. The provided theoretical runtime (without variants) is the same as of the algorithm discussed by Kaibel and Pfetsch in [7], see Sect. 4.

In practice the chosen data structures and implementation details make the implementation[1] very fast for the iteration and still fast for cover relations in the graded case compared to state-of-the-art implementations of previously considered algorithms, see Sect. 5.

In Sect. 6, we apply the presented algorithm to affirmatively settle *Wilf's conjecture* for all numerical semigroups of multiplicity 19 by iterating, up to a certain symmetry of order 18, through all faces of the *Kunz cone* (which is a certain unbounded polyhedron), identifying the *bad faces* which possibly yield counterexamples to Wilf's conjecture, and then checking that these do indeed not yield such counterexamples. This is based on recent work of Brunscet et al. [2] who developed this approach to the conjecture and were able to settle it up to multiplicity 18.

In Appendix A, we finally collect detailed runtime comparisons between the implementation of the presented algorithm with the state-of-the-art implementations in `polymake` and in `normaliz`.

2 Formal Framework

Let $(\mathscr{P}, \leq)$ be a finite poset and denote by $\prec$ its cover relation[2]. We usually write $\mathscr{P}$ for $(\mathscr{P}, \leq)$ and write $\mathscr{P}^{\mathrm{op}}$ for the opposite poset $(\mathscr{P}^{\mathrm{op}}, \leq_{\mathrm{op}})$ with $b \leq_{\mathrm{op}} a$ if $a \leq b$. For $a, b \in \mathscr{P}$ with $a \leq b$ we denote the interval as $[a, b] = \{p \in \mathscr{P} \mid a \leq p \leq b\}$. If $\mathscr{P}$ has a lower bound $\hat{0}$, its *atoms* are the upper covers of the lower bound,

$$\mathrm{Atoms}\, \mathscr{P} = \{p \in \mathscr{P} \mid \hat{0} \prec p\}$$

[1] See https://trac.sagemath.org/ticket/26887, merged into `SageMath` version `sage-8.9`.

[2] $a \prec b$ whenever $a < b$ and there does not exist c satisfying $a < c < b$

and, for $p \in \mathscr{P}$, we write Atoms $p = \{a \in \text{Atoms}\, \mathscr{P} \mid p \geq a\}$ for the atoms below p. Analogously, if $\mathscr{P}$ has an upper bound $\hat{1}$, its *coatoms* are the lower covers of the upper bound, coAtoms $\mathscr{P} = \{p \in \mathscr{P} \mid p \prec \hat{1}\}$. $\mathscr{P}$ is called *graded* if it admits a rank function $r\colon \mathscr{P} \to \mathbb{Z}$ with $p \prec q \;\Rightarrow\; r(p) + 1 = r(q)$.

Definition 2.1 $\mathscr{P}$ is *locally branched* if for every chain $a \prec b \prec c$ there exists an element $d \neq b$ with $a < d < c$. If this element is unique, then $\mathscr{P}$ is said to have the *diamond property*.

The diamond property is a well-known property of face lattices of polytopes, see [10, Thm. 2.7 (iii)]. The property of being locally branched has also appeared in the literature in contexts different from the present one, under the name 2-*thick lattices*, see for example [1] and the references therein.

An obvious example of a locally branched lattice is the Boolean lattice B_n given by all subsets of $\{1, \ldots, n\}$ ordered by containment. We will later see that all locally branched lattices with n atoms are isomorphic to meet semi-sublattices of B_n.

In the following, we assume $\mathscr{P}$ to be a finite lattice with meet operation $\wedge$, join operation $\vee$, lower bound $\hat{0}$, and upper bound $\hat{1}$. We say that

- $\mathscr{P}$ is *atomic* if all elements are joins of atoms,
- $\mathscr{P}$ is *coatomic* if all elements are meets of coatoms,
- $p \in \mathscr{P}$ is *join-irreducible* if p has a unique lower cover $q \prec p$,
- $p \in \mathscr{P}$ is *meet-irreducible* if p has a unique upper cover $p \prec q$.

Atoms are join-irreducible and coatoms are meet-irreducible. The following classification of atomic and coatomic lattices is a well-known folklore.

Lemma 2.2 *We have that*

(i) *$\mathscr{P}$ is atomic if and only if the only join-irreducible elements are the atoms,*
(ii) *$\mathscr{P}$ is coatomic if and only if the only meet-irreducible elements are the coatoms.*

Proof First observe that for all $p, q \in \mathscr{P}$ we have $p \geq q \;\Rightarrow\;$ Atoms $p \supseteq$ Atoms q and $p \geq \bigvee \text{Atoms}\, p$. Moreover, $\mathscr{P}$ is atomic if and only if $p = \bigvee \text{Atoms}\, p$ for all $p \in \mathscr{P}$.

Assume that $\mathscr{P}$ is atomic and let $q \in \mathscr{P}$ be join-irreducible and $p \prec q$. Because we have Atoms $p \neq$ Atoms q, it follows that $p = \hat{0}$. Next assume that $\mathscr{P}$ is not atomic and let $p \in \mathscr{P}$ be minimal such that $p > \bigvee \text{Atoms}\, p$. If $q < p$ then, by minimality, $q = \bigvee \text{Atoms}\, q$. It follows that $q \leq \bigvee \text{Atoms}\, p$ and p is join-irreducible. The second equivalence is the first applied to $\mathscr{P}^{\text{op}}$. □

Example 2.3 The face lattice of a polytope has the diamond property, it is atomic and coatomic, and every interval is again the face lattice of a polytope. The face lattice of an (unbounded) polyhedron might neither be atomic nor coatomic as witnessed by the face lattice of the nonnegative orthant in $\mathbb{R}^2$ with five faces. Example 2.11 will explain how to deal with this.

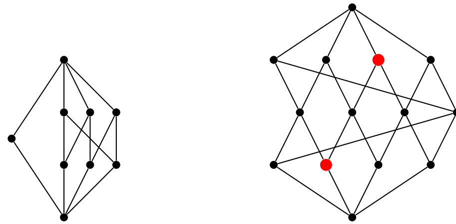

Fig. 1 A non-graded locally branched lattice and an atomic, coatomic, not locally branched lattice

The reason to introduce locally branched posets is the following relation to atomic and coatomic lattices, which has, to the best of our knowledge, not appeared in the literature.

Proposition 2.4 *The following statements are equivalent:*

(i) $\mathscr{P}$ *is locally branched,*
(ii) *every interval of* $\mathscr{P}$ *is atomic,*
(iii) *every interval of* $\mathscr{P}$ *is coatomic.*

Proof $\mathscr{P}$ is locally branched if and only if $\mathscr{P}^{\mathrm{op}}$ is locally branched. Also, $\mathscr{P}$ is atomic if and only if $\mathscr{P}^{\mathrm{op}}$ is coatomic. Hence, it suffices to show (i) $\Leftrightarrow$ (ii). Suppose $\mathscr{P}$ is not locally branched. Then, there exist $p \prec x \prec q$ such that the interval $[p, q]$ contains exactly those three elements. Clearly, $[p, q]$ is not atomic. Now suppose $[p, q] \subseteq \mathscr{P}$ is not atomic. Lemma 2.2 implies that there is join-irreducible x with unique lower cover y with $p < y \prec x$. There exists $z \in [p, q]$ with $z \prec y$ and the interval $[z, x]$ contains exactly those three elements. □

Example 2.5 Figure 1 left gives an example of a non-graded locally branched lattice. On the right it gives an example of an atomic, coatomic lattice, which is not locally branched as the interval between the two larger red elements contains only three elements.

Let $\mathscr{P}$ be a finite locally branched poset with atoms $\{1, \dots, n\}$. We have seen that $\mathscr{P}$ is atomic and thus $p = \bigvee \operatorname{Atoms} p$ for all $p \in \mathscr{P}$. The following proposition underlines the importance of subset checks and of computing intersections to understanding finite locally branched lattices.

Proposition 2.6 *In a finite locally branched lattice it holds that*

(i) $p \le q \;\Leftrightarrow\; \operatorname{Atoms} p \subseteq \operatorname{Atoms} q$;
(ii) $p \wedge q = \bigvee(\operatorname{Atoms} p \cap \operatorname{Atoms} q)$.

Proof (i) If $p \le q$ then clearly $\operatorname{Atoms} p \subseteq \operatorname{Atoms} q$. On the other hand, if $\operatorname{Atoms} p \subseteq \operatorname{Atoms} q$, then $p = \bigvee \operatorname{Atoms} p \le \bigvee \operatorname{Atoms} q = q$, as $\bigvee \operatorname{Atoms} q$ is in particular an upper bound for $\operatorname{Atoms} p$.

(ii) By (i) it holds that $\bigvee(\operatorname{Atoms} p \cap \operatorname{Atoms} q)$ is a lower bound of p and q. Also, $\operatorname{Atoms}(p \wedge q) \subseteq \operatorname{Atoms} p, \operatorname{Atoms} q$ and we obtain

$$\operatorname{Atoms}(p \wedge q) \subseteq \operatorname{Atoms} p \cap \operatorname{Atoms} q. \qquad \square$$

This proposition provides the following meet semi-lattice[3] embedding[4] of any finite locally branched lattice into a Boolean lattice.

Corollary 2.7 *Let $\mathscr{P}$ be a finite locally branched lattice with* Atoms $\mathscr{P} = \{1, \ldots, n\}$. *The map $p \mapsto$* Atoms p *is a meet semi-lattice embedding of $\mathscr{P}$ into the Boolean lattice B_n.*

Example 2.8 The above embedding does not need to be a join semi-sublattice embedding as witnessed by the face lattice of a square in $\mathbb{R}^2$.

Remark 2.9 Proposition 2.6 shows that checking whether the relation $p \leq q$ holds in $\mathscr{P}$ is algorithmically a subset check Atoms $p \subseteq$ Atoms q, while computing the meet is given by computing the intersection Atoms $p \cap$ Atoms q.

Justified by Corollary 2.7, we restrict our attention in this paper to meet semi-sublattices of the Boolean lattice.

2.1 Variants of this Framework and Examples

Before presenting in Sect. 3 the algorithm to iterate over the elements of a finite locally branched lattice together with variants to avoid any element above certain atoms and to avoid any element below certain coatoms (or other elements of B_n), we give the following main use cases for such an iterator.

Example 2.10 (*polytope*) The face lattice of a polytope P has the diamond property and is thus locally branched.

Example 2.11 (*polyhedron*) A polyhedron P can be projected onto the orthogonal complement of its linear subspace. The face lattices of those polyhedra are canonically isomorphic. Thus, we can assume that P does not contain an affine line. It is well known (see e.g. [10, Exer. 2.19]) that we may add an extra facet $\overline{F}$ to obtain a polytope $\overline{P}$. The faces of P are exactly the faces of $\overline{P}$ not contained in $\overline{F}$ (together with the empty face). Thus, the iterator visits all non-empty faces of P by visiting all faces of $\overline{P}$ not contained in $\overline{F}$.

Example 2.12 (*polytopal subdivision of manifold*) The face poset of a finite polytopal subdivision of a closed manifold (compact manifold without boundary). Adding an artificial upper bound $\hat{1}$, this is a finite locally branched lattice.

Example 2.13 (*extended tight spans*) We consider extended tight spans as defined in [6, Sect. 3] as follows: Let $P \subset \mathbb{R}^d$ be a finite point configuration, and let Σ be a polytopal complex with vertices P, which covers the convex hull of P. We call the maximal cells of Σ facets. We can embed Σ into a closed d-manifold M: We can add a vertex at infinity and for each face F on the boundary of Σ a face $F \cup \{\infty\}$. In many cases, just adding one facet containing all vertices on the boundary will work as well.

[3] A poset with meet operation.

[4] A poset-embedding preserving meets.

Given a collection Γ of boundary faces of Σ, we can iterate over all elements of Σ, which are not contained in Γ: Iterate over all faces of M, which are not contained in $\Gamma \cup (M \setminus \Sigma)$.

In the case when Γ is the collection of all boundary faces and Σ is therefore the tight span of the polytopal subdivision, and if Σ permits to add a single facet F to obtain a closed d-manifold M, we can just iterate over all faces of M not contained in F.

Example 2.14 (*closed sets of a matroid*) The MacLane–Steinitz exchange property (see e.g. [9, Lem. 1.4.2]) ensures that the closed sets of a matroid form a locally branched finite lattice.

Example 2.15 (*locally branched lattices with non-trivial intersection*) Let $\mathscr{P}_1, \dots, \mathscr{P}_k$ be finite locally branched meet semi-sublattices of B_n such that for $p \in \mathscr{P}_i$ and $q \in \mathscr{P}_j$ with $\operatorname{Atoms} p \subseteq \operatorname{Atoms} q$ it follows that $p \in \mathscr{P}_j$. Then the iterator may iterate through all elements of their union by first iterating through $\mathscr{P}_1$, then through all elements in $\mathscr{P}_2$ not contained in $\mathscr{P}_1$ and so on.

Example 2.16 (*polyhedral complexes*) Using the iteration as in the previous example allows to iterate through polytopal or polyhedral complexes.

3 The Algorithm

Let $\mathscr{P}$ be a finite locally branched lattice given as a meet semi-sublattice of the Boolean lattice B_n. We assume $\operatorname{Atoms} \mathscr{P} = \{1, \dots, n\}$ and we may identify an element p with $\operatorname{Atoms} p$. The following algorithm is a recursively defined depth-first iterator through the elements of $\mathscr{P}$. Given $p \in \mathscr{P}$ and its lower covers $x_1, \dots, x_k$, the iterator yields p and then computes, one after the other, the lower covers of $x_1, \dots, x_k$, taking into account those to be ignored, and then recursively proceeds. Being an *iterator* means that the algorithm starts with only assigning the input to the respective variables and then waits in its current state. Whenever an output is requested, it starts from its current state and runs to the point **OUTPUT**, outputs the given output, and again waits.

```
1  DECLARATIONS
2   • c, r, v, x            – sets of integers
3   • C, C_new, V, V_new    – duplicate-free lists (of sets of integers)
4   • D, E                  – lists (of sets of integers), possibly with duplicates
5
6  Algorithm FaceIterator
7
8  INPUT
9   • C    – all coatoms of 𝒫 not contained in any of V
10  • V    – list of subsets of {1, ..., n}
11  • r    – subset of {1, ..., n}
12
13 OUTPUT: Each c ∈ 𝒫 with c ≠ 1̂ ∧ c ∩ r = ∅ ∧ ∀v ∈ V : c ⊈ v.
14
15 PROCEDURE
16 if C ≠ []:
17   c := an_element(C)
```

```
18  if c ∩ r = ∅:
19      OUTPUT c    # continue from here
20
21   D = [c ∩ x : x ∈ C, x ≠ c]
22   E = [x ∈ D : ∀ v ∈ V : x ⊈ v]
23   C_new = inclusion_maximals(E)
24   V_new = [v : v ∈ V]
25
26  # Apply algorithm for sublattice [0, c]
27  FaceIterator(C_new, V_new, r)
28
29   V = V + [c ∪ r]           # append V
30   C = [x ∈ C : x ⊈ c ∪ r]  # update C accordingly
31  FaceIterator(C, V, r)
```

The recursive function calls in lines 27 and 31 can be executed in parallel: r can be declared constant. The lists C and V will be modified, but not their elements.

One should think of V as a list of inclusion maximal elements of those already visited.

The algorithm does not visit $\hat{1}$. However, we will still assume that this is the case whenever suitable. This would have to be done, before calling the algorithm.

For polyhedra, a technically elaborated version of this algorithm is implemented in `SageMath`[1]. Before proving the correctness of the algorithm, we provide several detailed examples. In the examples, we do not ignore any atoms and set $r = \emptyset$. Also V will be empty if not specified.

Example 3.1 We apply the algorithm to visit faces of a square.

- **INPUT**: $C = [\{1, 2\}, \{1, 4\}, \{2, 3\}, \{3, 4\}]$
- $c = \{1, 2\}$, **OUTPUT**: $\{1, 2\}$
- $C_{\text{new}} = [\{1\}, \{2\}]$
- Apply **FaceIterator** to sublattice $[\hat{0}, \{1, 2\}]$
  - **INPUT**: $C = [\{1\}, \{2\}]$
  - $c = \{1\}$, **OUTPUT**: $\{1\}$
  - $C_{\text{new}} = [\emptyset]$
  - Apply **FaceIterator** to sublattice $[\hat{0}, \{1\}]$
    - **INPUT**: $C = [\emptyset]$
    - $c = \emptyset$, **OUTPUT**: $\emptyset$
    - (C_{new} is empty)
    - Apply **FaceIterator** to sublattice $[\hat{0}, \hat{0}]$ without output
    - Add $\emptyset$ to V (to the copy in this call of **FaceIterator**)
    - Reapply **FaceIterator** to sublattice $[\hat{0}, \{1\}]$
    - **INPUT**: $C = []$, $V = [\emptyset]$
  - $V = [\{1\}]$
  - Reapply **FaceIterator** to sublattice $[\hat{0}, \{1, 2\}]$
  - **INPUT**: $C = [\{2\}]$, $V = [\{1\}]$
  - $c = \{2\}$, **OUTPUT**: $\{2\}$
  - Apply **FaceIterator** to sublattice $[\hat{0}, \{2\}]$

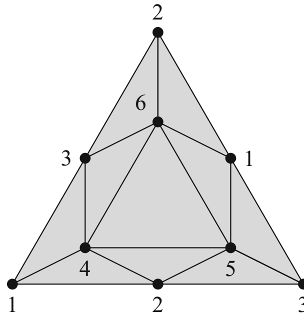

Fig. 2 Minimal triangulation of $\mathbb{RP}^2$ with vertices $1, \dots, 6$

- **INPUT**: $C = []$, $V = [\{1\}]$
  - $V = [\{1\}, \{2\}]$
  - Reapply **FaceIterator** to sublattice $[\hat{0}, \{1, 2\}]$
  - **INPUT**: $C = []$, $V = [\{1\}, \{2\}]$
- $V = [\{1, 2\}]$
- Reapply **FaceIterator** to entire lattice
- **INPUT**: $C = [\{1, 4\}, \{2, 3\}, \{3, 4\}]$, $V = [\{1, 2\}]$
- $c = \{1, 4\}$, **OUTPUT**: $\{1, 4\}$
- Apply **FaceIterator** to sublattice $[\hat{0}, \{1, 4\}]$
  - **INPUT**: $C = [\{4\}]$, $V = [\{1, 2\}]$
  - $c = \{4\}$, **OUTPUT**: $\{4\}$
  - Apply **FaceIterator** to sublattice $[\hat{0}, \{4\}]$ without output
- $V = [\{1, 2\}, \{1, 4\}]$
- ... further outputs: $\{2, 3\}, \{3\}, \{3, 4\}$

Example 3.2 We apply the algorithm to the minimal triangulation of $\mathbb{RP}^2$ given in Fig. 2.

- **INPUT**: $C = [\{1, 2, 4\}, \dots, \{4, 5, 6\}]$
- $c = \{1, 2, 4\}$, **OUTPUT**: $\{1, 2, 4\}, \{1, 2\}, \{1\}, \emptyset, \{2\}, \{1, 4\}, \{4\}, \{2, 4\}$
- $c = \{1, 2, 6\}$, **OUTPUT**: $\{1, 2, 6\}, \{1, 6\}, \{6\}, \{2, 6\}$
- $c = \{1, 3, 4\}$, **OUTPUT**: $\{1, 3, 4\}, \{1, 3\}, \{3\}, \{3, 4\}$
- $c = \{1, 3, 5\}$, **OUTPUT**: $\{1, 3, 5\}, \{1, 5\}, \{5\}, \{3, 5\}$
- $c = \{1, 5, 6\}$, **OUTPUT**: $\{1, 5, 6\}, \{5, 6\}$
- $c = \{2, 3, 5\}$, **OUTPUT**: $\{2, 3, 5\}, \{2, 3\}, \{2, 5\}$
- $c = \{2, 3, 6\}$, **OUTPUT**: $\{2, 3, 6\}, \{3, 6\}$
- $c = \{2, 4, 5\}$, **OUTPUT**: $\{2, 4, 5\}, \{4, 5\}$
- $c = \{3, 4, 6\}$, **OUTPUT**: $\{3, 4, 6\}, \{4, 6\}$
- $c = \{4, 5, 6\}$, **OUTPUT**: $\{4, 5, 6\}$

Example 3.3 We apply the algorithm to the tight span given in Fig. 3.

- **INPUT**: $C = [\{1, 2, 3, 4\}, \{1, 2, 5, 6\}, \{1, 3, 6\}, \{2, 4, 5\}]$, $V = [\{3, 4, 5, 6\}]$

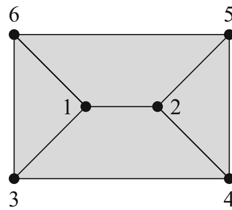

Fig. 3 Tight span on vertices $1, \ldots, 6$ with interior vertices 1 and 2

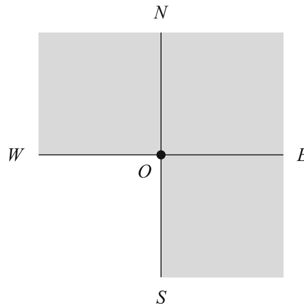

Fig. 4 Polyhedra complex consisting of three quadrants of the plane with south-west quadrant removed

- $c = \{1, 2, 3, 4\}$, **OUTPUT**: $\{1, 2, 3, 4\}, \{1, 2\}, \{1\}, \{2\}, \{1, 3\}, \{2, 4\}$
- $c = \{1, 2, 5, 6\}$, **OUTPUT**: $\{1, 2, 5, 6\}, \{1, 6\}, \{2, 5\}$
- $c = \{1, 3, 6\}$, **OUTPUT**: $\{1, 3, 6\}$
- $c = \{2, 4, 5\}$, **OUTPUT**: $\{2, 4, 5\}$

Example 3.4 Visit all faces of the polyhedral complex given in Fig. 4.

- *Incorrect application* by applying to the polyhedra as if they were facets.
  - **INPUT**: $C = [\{W, N, 0\}, \{N, E, 0\}, \{S, E, 0\}]$
  - $c = \{W, N, 0\}$, **OUTPUT**: $\{W, N, 0\}$
  - $C_{\text{new}} = [\{N, 0\}]$, **OUTPUT**: $\{N, 0\}$
  - $V = [\{W, N, 0\}]$
  - $c = \{N, E, 0\}$, **OUTPUT**: $\{N, E, 0\}$
  - $C_{\text{new}} = [\{E, 0\}]$, **OUTPUT**: $\{E, 0\}$
  - $V = [\{W, N, 0\}, \{N, E, 0\}]$
  - $c = \{S, E, 0\}$, **OUTPUT**: $\{S, E, 0\}$
  - $C_{\text{new}} = []$
- *Correct application* by applying successively to all faces of all polyhedra:
  - Before applying **FaceIterator** to $\{W, N, 0\}$: **OUTPUT**: $\{W, N, 0\}$
  - Apply algorithm to $\{W, N, 0\}$:
    - **INPUT**: $C = [\{W, 0\}, \{N, 0\}]$, $V = [\{W, N\}]$
    - **OUTPUT**: $\{W, 0\}, \{0\}, \{N, 0\}$
  - Before applying **FaceIterator** to $\{N, E, 0\}$: **OUTPUT**: $\{N, E, 0\}$

- Apply algorithm to $\{N, E, 0\}$:
  - **INPUT**: $C = [\{E, 0\}]$, $V = [\{W, N, 0\}, \{N, E\}]$
  - **OUTPUT**: $\{E, 0\}$
- Before applying **FaceIterator** to $\{S, E, 0\}$: **OUTPUT**: $\{S, E, 0\}$
- Apply algorithm to $\{S, E, 0\}$:
  - **INPUT**: $C = [\{S, 0\}]$, $V = [\{W, N, 0\}, \{N, E, 0\}, \{S, E\}]$
  - **OUTPUT**: $\{S, 0\}$

3.1 Correctness of the Algorithm

As assumed, let $\mathscr{P}$ be a locally branched meet semi-sublattice of the Boolean lattice B_n. In the following, we see that the algorithm visits each element $p \in \mathscr{P}$ not contained in any of V and not containing any of r exactly once. We remark that we could relax the condition on B_n: It suffices for the interval $[p, \hat{1}]$ to be locally branched for p to be visited exactly once under those conditions.

Proposition 3.5 *The algorithm* **FaceIterator** *is well defined in the following sense: Let C be the list of coatoms of $\mathscr{P}$ that are not contained in any of V.*

(i) *Then the call of* ***FaceIterator*** *in line* 27 *calls the algorithm for the sublattice* $[\hat{0}, c]$ *with C_{new} being the list of coatoms of* $[\hat{0}, c]$ *that are not contained in any of V.*
(ii) *The call of* ***FaceIterator*** *in line* 31 *calls the algorithm for $\mathscr{P}$, but with $c \cup r$ appended to V. The updated C contains all coatoms of $\mathscr{P}$ that are not contained in any of V.*

Proof (i) C_{new} is a sublist of E, which is a sublist of D. By construction all elements in D and thus in C_{new} are strictly below c. Now, let $d \prec c \prec \hat{1}$ in $\mathscr{P}$ and let d not be contained in any of V. Since $\mathscr{P}$ is locally branched there is an element $x \neq c$ with $d < x \prec \hat{1}$, implying $d = c \cap x$. If d is not contained in any of V, then the same must hold for x as $d < x$. This implies that x is in C and thus d is contained in D.

Assume that d is contained in D. It is contained in E exactly if it is not contained in any of V by construction of E in line 22. It remains to show that d in E is contained in C_{new} exactly if $d \prec c$. As any element in E is strictly below c, $d \prec c$ implies that d is inclusion maximal. On the other hand, if d is not inclusion maximal, it lies below a coatom of $[\hat{0}, c]$. As d is in E, it cannot be contained in any of V and the same holds for this coatom. Thus, d is not inclusion maximal in E.

(ii) Line 30 removes exactly those elements in C that are contained in $c \cup r$. □

Theorem 3.6 *The algorithm* **FaceIterator** *iterates exactly once over all elements in $\mathscr{P}$ that are not contained in any of V and do not contain any element in r.*

Proof We argue by induction on the cardinality of C. First note that the cardinalities of C_{new} and C in the two subsequent calls of **FaceIterator** in lines 27 and 31 are both strictly smaller than the cardinality of C. If $C = []$, then all elements of $\mathscr{P} \setminus \hat{1}$ are contained in elements of V, and the algorithm correctly does not output any element. Suppose that C is not empty and let c be the element assigned in line 17. Let $p \in \mathscr{P}$. If p is contained in an element of V, then it is not contained in the initial C. By

Proposition 3.5 it will never be contained in C in any recursive call and thus cannot be output. On the other hand, if p contains an element in r, then it cannot be output by line 18. Otherwise,

- if $p = c$, then the algorithm outputs p correctly in line 19,
- if $p < c$, then p is contained in $[\hat{0}, c]$ and is output by **FaceIterator** in line 27 by induction,
- if $p \nleq c$, then $p \nleq c \cup r$ and p is output in the call of **FaceIterator** in line 31 by induction, as it is not contained in any of $V + [c \cup r]$. □

3.2 Variants of the Algorithm

We finish this section with a dualization property followed by explicitly stating the result when applying the algorithm for the variants discussed in Sect. 2.1.

Let $\mathscr{P}$ be a locally branched lattice and V be a list of coatoms, and r be a list of atoms. Instead of directly applying Theorem 3.6 one can consider $\mathscr{P}^{\mathrm{op}}$, V^{op}, and r^{op}. V^{op} is now a list of atoms of $\mathscr{P}^{\mathrm{op}}$ (given as indices). r^{op} is a list of coatoms of $\mathscr{P}^{\mathrm{op}}$ (each given as list of atoms of $\mathscr{P}^{\mathrm{op}}$).

Corollary 3.7 *The algorithm can be applied to visit all elements of $\mathscr{P}^{\mathrm{op}}$, which are not contained in any of r^{op}, and do not contain any element in V^{op}. This is the same as visiting all elements of $\mathscr{P}$ that are not contained in any of V and do not contain any element in r, but that each element is now given as coatom-incidences instead of atom-incidences.*

We later see in Theorem 4.1 that considering $\mathscr{P}^{\mathrm{op}}$ instead of $\mathscr{P}$ might be faster as the runtime depends on the number of coatoms. For example, in Example 3.2 one could apply the algorithm to $\mathscr{P}^{\mathrm{op}}$ to improve runtime as there are ten facets but only six vertices.

Corollary 3.8 (i) *Let P be a polytope and let $\mathscr{P}$ be its face lattice with coatoms C given as vertex/atom incidences. The algorithm then outputs every face of P as a list of vertices it contains.*

(ii) *Let P be an unbounded polyhedron with trivial linear subspace and let $\overline{P}$ be a projectively equivalent polytope with marked face. Provided V, a list containing just the marked face of $\overline{P}$, and C, the remaining facets, all are given as vertex incidences. The algorithm then outputs all faces of P as vertex/ray incidences.*

(iii) *Let P be a finite polytopal subdivision of a closed manifold. Let C be the maximal faces given as vertex incidences. The algorithm then outputs the faces of P as vertex incidences.*

(iv) *Let Σ be an extended tight span in $\mathbb{R}^d$ as described in Example 2.13. Let Γ be a subset of boundary faces of Σ. As explained in Example 2.13 we can embed Σ into a (triangulated) manifold M. Given the maximal faces of $M \setminus \Sigma$ and Γ as V and the remaining maximal faces as C all as vertex incidences, the algorithm outputs the faces of Σ not contained in any of Γ as vertex incidences.*

(v) *Let P be a polyhedral complex. Given the atom incidences of the facets of each maximal face. The algorithm can be iteratively applied to output all faces of P: Let F be a maximal face. Given the atom incidences of the facets of F (and*

possibly the marked far face). As described in (i) *and* (ii)*, the algorithm outputs all faces of F. Let $F_1, \ldots, F_n$ be some other maximal faces. Append $F_1, \ldots, F_n$ (as atom/ray incidences) to V and remove all elements of C contained in any of $F_1, \ldots, F_n$. Then, the algorithm outputs all faces of F not contained in any of $F_1, \ldots, F_n$.*

4 Data Structures, Memory Usage, and Theoretical Runtime

The operations used in the algorithm are `intersection`, `is_subset`, and `union`. It will turn out that the crucial operation for the runtime is the subset check.

For the theoretical runtime we consider representation as (sparse) *sorted-lists-of-atoms*. However, in the implementation we use (dense) *atom-incidence-bit-vectors*. This is theoretically slightly slower, but the crucial operations can all be done using bit-wise operations. The improved implementation only considers the significant chunks, which has optimal theoretic runtime again. A chunk contains 64/128/256 bits depending on the architecture. We store for each set, which chunk has set bits. To check whether A is a subset of B, it suffices to loop through the significant chunks of A. Experiments suggest that for many atoms `RoaringBitmap` described in [8] performs even better.[5]

Observe that a *sorted-lists-of-atoms* needs as much memory as there are incidences. Consider two sets A and B (of integers) of lengths a and b, respectively, and a (possibly unsorted) list C of m sets $C_1, \ldots, C_m$ with $\alpha = |C_1| + \cdots + |C_m|$. Using standard implementations, we assume in the runtime analysis that

- intersection $A \cap B$ and union $A \cup B$ have runtime in $\mathcal{O}(a+b) = \mathcal{O}(\max(a,b))$ and the results can be guaranteed to be sorted,
- a subset check $A \subseteq B$ or $A \subsetneq B$ has runtime in $\mathcal{O}(b)$, and
- to check whether A is a subset of any element in C has runtime in $\mathcal{O}(\alpha)$.

Let $d+1$ be the number of elements in a longest chain in $\mathscr{P}$, let $m = |C|$, $n = |\operatorname{Atoms} \mathscr{P}|$, and let

$$\alpha = \sum_{a \in C \cup V} |a \cup r|.$$

(In the case that V and r are both empty, the sum of cardinalities of $C_1, \ldots, C_m$ is α. Otherwise it is bounded by α). Let φ be the number of elements in $\mathscr{P}$ that are not contained in any of V. If r is empty, this is the cardinality of the output.

Theorem 4.1 *The algorithm has memory consumption $\mathcal{O}(\alpha \cdot d)$ and runtime $\mathcal{O}(\alpha \cdot m \cdot \varphi)$.*

Remark 4.2 We assume constant size of integers as in [7]. To drop this assumption, one needs to multiply our runtime and memory usage by $\log(\max(n, m))$ and likewise for [7].

[5] `RoaringBitmap` performs better for computing the f-vector of the d-dimensional associahedron for $d \geq 11$. See discussion on https://github.com/Ezibenroc/PyRoaringBitMap/pull/59.

Proof We will assume that recursive calls are not made, when C resp. C_{new} are empty. To check whether a list is empty can be performed in constant time. Then, the number of recursive calls is bounded by φ: Any element assigned to c is an element from $\mathscr{P}$. Any element in $\mathscr{P}$ is assigned at most once. This follows from the proof of Theorem 3.6. (It follows directly, if r is empty as then every element assigned to c is also output.) Note that for each recursive call of **FaceIterator** the number of elements in C is bounded by m. The sum of the cardinalities of C, D, E, and V is bounded by α. So is the cardinality of r.

To prove the claimed runtime, it suffices to show that each call of **FaceIterator** not considering recursive calls has runtime in $\mathscr{O}(m \cdot \alpha)$. With above assumptions, this follows: The check preceding the output in line 18 can be performed in $\mathscr{O}(\alpha)$. Obtaining D in line 21 can be done in $\mathscr{O}(n \cdot m)$ (each size is bounded by n and there are at most m intersections to perform). $n \leq \alpha$ as every atom must be contained at least once in a coatom of C, in an element of V or in r. To check, whether an element is contained in any of V can be done in $\mathscr{O}(\alpha)$. Again there are at most m elements, so the claim holds for line 22. To check whether an element is contained in any of E can be done in $\mathscr{O}(\alpha)$ and the claim holds for 23. Note that we can perform a strict subset check for larger indices and a non-strict subset check for smaller indices to remove duplicates as well. Clearly, we can copy V to V_{new} in this time and append V in line 29. The individual subset check for each of at most m sets in line 30 is done in $\mathscr{O}(\alpha)$. This proves the claimed runtime.

A single call of **FaceIterator** has memory usage at most $c \cdot \alpha$ for a global constant c, not taking into account the recursive calls. The call in line 31 does not need extra memory as all old variables can be discarded. The longest chain of the lattice $[0, c]$ is at most of length $d - 1$. By induction the call of **FaceIterator** in line 27 has total memory consumption at most $(d - 1) \cdot c \cdot \alpha$. The claimed bound follows. □

Remark 4.3 When searching elements with certain properties, we might observe from c that all of $[\hat{0}, c]$ is not of interest. After assigning c we can skip everything until line 27. This will result in not visiting any further element of $[\hat{0}, c]$ (some might have been visited earlier).

If r is empty and the check whether to skip $[\hat{0}, c]$ can be performed in time $\mathscr{O}(m \cdot |c|)$, the runtime will reduce linear to the number of elements output: Appending V in line 29 and updating C in line 30 can both be performed in time $\mathscr{O}(m \cdot |c|)$. This runtime can be accounted for by an upper cover of c, which we must have visited: The sum of the cardinalities of the lower covers of an element is bounded by α. Thus the runtime of skipping elements accounts for runtime in $\mathscr{O}(m \cdot \alpha)$ per element visited. If we skip some of the $[\hat{0}, c]$ in this way, the runtime will therefore be in $\mathscr{O}(\alpha \cdot m \cdot \psi)$, where ψ is the cardinality of the output.

4.1 Computing All Cover Relations

Applying the algorithm to a graded locally branched meet semi-sublattice of B_n while keeping track of the recursion depth allows an a posteriori sorting of the output by the level sets of the grading. The recursion depth is the number of iterative calls using

line 27. We obtain the same bound for generating all cover relations as Kaibel and Pfetsch [7]. For a list L of (sorted) subsets of $\{1, \ldots, n\}$ we additionally assume that

- two sets of cardinality a and b resp. can be lexicographically compared in time $\mathcal{O}(\min(a, b))$,
- L can be sorted in time $\mathcal{O}(n \cdot |L| \cdot \log |L|)$, and
- if L is sorted, we can look up, whether L contains some set of cardinality a in time $\mathcal{O}(a \cdot \log |L|)$.

Proposition 4.4 *Let $\mathscr{P}$ be a graded meet semi-sublattice of $\mathscr{B}_n$. Assume each level set of $\mathscr{P}$ to be given as* ***sorted-lists-of-atoms****, one can generate all cover relations in time $\mathcal{O}(\alpha \cdot \min(m, n) \cdot \varphi)$ with quantities as defined above using the above algorithm.*

Observe that in the situation of this proposition, V and r are both empty and in particular α is the total length of the coatoms. As before, φ is the number of elements in $\mathscr{P}$. The level sets are not assumed to be sorted.

Proof First, we sort all level-sets. As each element in $\mathscr{P}$ appears exactly once in each level set, all level sets can be sorted in time $\mathcal{O}(n \cdot \varphi \cdot \log \varphi)$. Then, we intersect each element with each coatom, obtaining its lower covers and possibly other elements. We look up each intersection to determine the lower covers. All such intersections are obtained in time $\mathcal{O}(\varphi \cdot m \cdot n)$. For a fixed element the total length of its intersections with all coatoms is bounded by α. Hence, all lookups are done in time $\mathcal{O}(\varphi \cdot \alpha \cdot \log \varphi)$. Finally, we note that $m, n \leq \alpha$ and that $\log \varphi \leq \min(m, n)$. ◻

In the ungraded case, one first sorts all elements in $\mathscr{P}$, and then intersects each element p with all coatoms. The inclusion maximal elements among those strictly below p are lower covers of p. They can be looked up in the list of sorted elements to obtain an index. Observe that all this is done time $\mathcal{O}(\alpha \cdot m \cdot \varphi)$.

4.2 Theoretic Comparison

We first compare our approach with the one from Kaibel and Pfetsch [7]. They have written their algorithm in terms of closure operators starting from the vertices. Using the terminology of our paper (applying their algorithm to the dual case), there are some differences:

(i) To translate from *coatom representation* to *atom representation* the corresponding coatoms are intersected. Likewise they translate from *atom representation* to *coatom representation*.
(ii) They store the *coatom representation* of c.
(iii) As a first step, they obtain the *atom representation* of c.
(iv) To obtain a list containing all lower covers, they intersect c with all coatoms not containing c.
(v) For checking which of the sets is inclusion maximal, they transform them back to *coatom representation*. The subset check is then trivial.
(vi) They do not store visited faces.

Our runtime is the same as in their approach. They require memory in $\mathcal{O}(\varphi \cdot m)$. In [7, Sect. 3.3] however, they mention that one could use lexicographic ordering to avoid storing all the faces and achieve similar memory usage as our approach. To our knowledge, this memory efficient approach has not been implemented.

Advantages of our algorithm to the lexicographic approach are:

(i) The order of output is somewhat flexible. We are free to choose any element c from C (in any recursion step) in line 17 of the algorithm.
(ii) Our order relates to the lattice: By Remark 4.3 we could skip some $[\hat{0}, c]$ and effectively reduce runtime. E.g. we could use the iterator to only visit faces of a polytope that are not a simplex, in runtime linear to the output.
(iii) Let G be the automorphism group of $\mathscr{P}$. If we sort the elements lexicographically by their *coatom representation*, any first element representative of an orbit, is contained in a first representative of a coatom-orbit. To visit all orbits, it suffices to visit only the first representatives of the facet-orbits and then append each facet in the orbit to V. This will efficiently reduce runtime.

The other algorithm we compare our approach to was described by Bruns et al. in [2] and was independently developed to our algorithm. It also stores each element in atom representation. In each step of the algorithm, the atom representation is computed. Then, each element c is intersected with all coatoms and the inclusion maximal elements are computed just like in our approach via subset checks. Finally the inclusion maximal elements are transformed to atom representation and, after a lookup, the new ones are stored. Although there is no theoretic analysis of runtime and memory consumption, it appears that the runtime agrees with [7] (although the implementation by dense *bit-vectors* does not achieve this) and the memory usage is $\mathcal{O}(\varphi \cdot n)$. They introduced usage of the automorphism group of the atom-coatom-incidences and have first developed a variation that visits the first representative.

5 Performance of the Algorithm Implemented in `SageMath`

We present running times for several computations. An implementation is available through `sage-8.9` and later. This uses dense *bit-vectors* and has runtime $\mathcal{O}(n \cdot m \cdot \varphi)$.

The presented algorithm can be parallelized easily as the recursive calls in lines 27 and 31 do not depend on each other. The implementation using bitwise operations allows to use advanced CPU-instructions such as *Advanced Vector Extensions*. Furthermore, *bit-vectors* can be enhanced to account for sparse vectors, obtaining asymptotically optimal runtime. All these improvements are available in `sage-9.4`.

The benchmarks are performed on an Intel® Core™ i7-7700 CPU @ 3.60GHz x86_64-processor with four cores and 30 GB of RAM. The computations are done either using

- `polymake 3.3` [5], or
- `normaliz 3.7.2` [3], or
- the presented algorithm in `sage-8.9`, or
- the presented algorithm in `sage-8.9` with additional parallelization, intrinsics, and subsequent improvements as explained above.

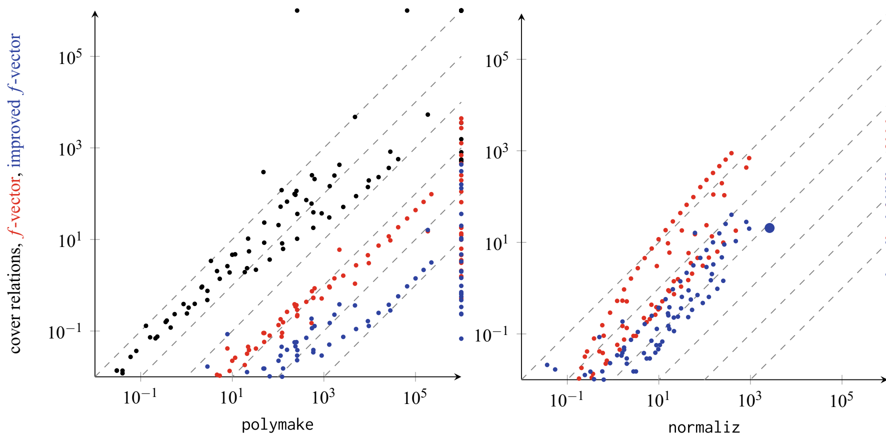

Fig. 5 Runtime comparison. Every dot represents one best-of-five computation, and every shifted diagonal is a factor-10 faster runtime. Dots on the boundary represent memory overflows. The left diagram compares polymake to three implementations: SageMath computing all cover relations (black), SageMath computing the f-vector (red) and SageMath with aforementioned improvements (blue). The right diagram compares normaliz to SageMath (without and with improvements) computing the f-vector. e.g. the fat blue dot in the right diagram has coordinates slightly bigger than $(10^3, 10)$ and represents computing the f-vector of the Kunz cone with parameter $m = 15$. It took 2622 seconds with normaliz and 21 seconds with SageMath with improvements

Remark 5.1 It appears that there is no difference in performance regarding the f-vector for polymake 3.3 and polymake 4.1. Likewise for normaliz 3.7.2 and normaliz 3.8.9 (a computation goal DualFVector was added, but we already applied normaliz to the dual problem, whenever suitable).

We computed:

(1) cover relations and f-vector in polymake (x-axis in the left diagram of Fig. 5),
(2) f-vector in normaliz *with* parallelization, (x-axis in the right diagram of Fig. 5),
(3) all cover relations with the presented implementation in SageMath,
(4) f-vector with the presented implementation in SageMath,
(5) f-vector with the presented implementation in SageMath with parallelization, intrinsics, and additional improvements.

Remark 5.2
– The computation of the f-vector in (1) also calculates all cover relations.
– polymake also provides a different algorithm to compute the f-vector from the h-vector for simplicial/simple polytopes (providing this additional information sometimes improves the performance in polymake).
– normaliz does not provide an algorithm to compute the cover relations.

For every algorithm we record the best-of-five computation[6] on

[6] Benchmarks were taken on a desktop computer. Best-of-five was chosen to account for other processes causing temporary slowdown. All implementations are completely deterministic without randomness.

- the simplex of dimension n,
- several instances of the cyclic polytope of dimension 10 and 20,
- the associahedron of dimension n,
- the permutahedron of dimension n embedded in dimension $n + 1$,
- a 20-dimensional counterexample to the Hirsch conjecture,
- the cross-polytope of dimension n,
- the Birkhoff polytope of dimension $(n - 1)^2$,
- joins of such polytopes with their duals,
- Lawrence polytopes of such polytopes,
- Kunz cone in dimension $n - 1$ defined in Definition 6.3.

Figure 5 confirms that the implementations behave about the same asymptotically. For computing the cover relations, the implementation in `sage-8.9` is as fast or up to 100 times faster than the implementation in `polymake`. For computing only the f-vector, the implementation in `sage-8.9` is about 1000 times faster than `polymake` and a bit faster than `normaliz`. However, `normaliz` used four physical cores (eight threads) for those results, while `sage-8.9` only needed one. With parallelization and other improvements one can gain a factor of about 10 using four physical cores.

5.1 Possible Reasons for the Performance Difference

In [2, Rem. 5.5] it was mentioned that for one example about 6 % of the computation time is needed for converting from *coatom representation* to *atom representation* and back and for computing the intersections. Another 4 % are required for computing which elements in C are inclusion-maximal. 40 % are observed by checking which elements were seen before. The rest are other operations such as system operations.

Contrary to this, in `sage-8.9` other operations are almost negligible. About 90 % of the time is spent doing subset checks. About 10 % of the time is spent computing the intersections. Note, that these times may vary depending on the application. There is no need to do the expensive translation from *coatom representation* to *atom representation* and back. It seems that our algorithm allowed to avoid those 90 % that `normaliz` spends with lookups and other operations.

As parallelization is trivial, there is very little overhead even with as much as 40 threads: The overhead is due to the fact that we perform a depth-first search. When the function is called, we can almost immediately dispatch the call at line 31. In this way, there are trivially independent jobs (one per coatom) that can be parallelized without overhead. However, the workload is shared badly. In the extreme example of the Boolean lattice, half of the elements visited will be subject to the first job in that way and we should expect this to take half of the time.

Our approach is to have one job per coatom of the coatoms (parallelization at codimension 2). The jobs are assigned monotonic dynamically. Each thread has independent data structure and recomputes the first C_{new} if necessary. This still has almost no overhead. However, when computing the bad orbits of the Kunz cone with 40 threads, one of them took about a day longer to finish than the others. Experiments suggest that parallelizing at codimension 3 has still reasonable overhead and will pay

off with enough threads (depending on the lattice). At level 4 the overhead seems unreasonable.

As for `polymake` the comparison is of course unfair. Their implementation tries to compute all cover relations in decent time and asymptotically optimal. Of course, one can be much faster, when not storing all cover relations. The implementation in `sage-8.9` to compute the cover relations is usually faster. We refer to Appendix A for detailed runtimes, which were plotted in Fig. 5.

6 Application of the Algorithm to Wilf's Conjecture

Bruns et al. provided an algorithm that verifies Wilf's conjecture for a given fixed multiplicity [2]. We give a brief overview of their approach:

Definition 6.1 A *numerical semigroup* is a set $S \subset \mathbb{Z}_{\geq 0}$ containing 0 that is closed under addition and has finite complement.

- Its *conductor* $c(S)$ is the smallest integer c such that $c + \mathbb{Z}_{\geq 0} \subseteq S$.
- Its *sporadic elements* are the elements $a \in S$ with $a < c(S)$ and let $n(S)$ be the number of sporadic elements.
- The *embedding dimension* $e(S) = |S \setminus (S + S)|$ is the number of elements that cannot be written as sum of two elements.
- The *multiplicity* $m(S)$ is the minimal nonzero element in S.

Conjecture 6.2 (Wilf) *For any numerical semigroup S,*

$$c(S) \leq e(S)n(S).$$

For fixed multiplicity m one can analyze certain polyhedra to verify this conjecture.

Definition 6.3 [2, Defn. 3.3] Fix an integer $m \geq 3$. The *relaxed Kunz polyhedron* is the set P'_m of rational points $(x_1, \ldots, x_{m-1}) \in \mathbb{R}^{m-1}$ satisfying

$$\begin{aligned} x_i + x_j &\geq x_{i+j}, \qquad 1 \leq i \leq j \leq m-1, \quad i + j < m, \\ x_i + x_j + 1 &\geq x_{i+j}, \qquad 1 \leq i \leq j \leq m-1, \quad i + j > m, \end{aligned}$$

The *Kunz cone* is the set C_m of points $(x_1, \ldots, x_{m-1}) \in \mathbb{R}^{m-1}$ satisfying

$$x_i + x_j \geq x_{i+j}, \qquad 1 \leq i \leq j \leq m-1, \quad i + j \neq m.$$

(All indices in this definition are taken modulo m.)

Remark 6.4 Every numerical semigroup S of multiplicity m corresponds to a lattice point in the *relaxed Kunz polyhedron* (not vice versa, thus *relaxed*): x_i is the smallest integer such that $i + mx_i \in S$. The inequalities correspond to $j + mx_j \in S$ implying that $i + j + m(x_i + x_j) \in S$.

Definition 6.5 Let F be a face of P'_m or C_m. Denote by $e(F)-1$ and $t(F)$ the number of variables not appearing on the right and left hand sides resp. of any defining equations of F.

The Kunz cone is a translation of the relaxed Kunz polyhedron. $e(F)$ and $t(F)$ are invariants of this translation.

Every numerical semigroup S of multiplicity m corresponds to an (all-)positive lattice point in P'_m. If the point corresponding to S lies in the relative interior of some face $F \subseteq P'_m$, then $e(F) = e(S)$ and $t(F) = t(S)$, see [2, Thm. 3.10 & Cor. 3.11]. The following proposition summarizes the approach by which we can check for bad faces:

Proposition 6.6 [2] *There exists a numerical semigroup S with multiplicity m that violates Wilf's conjecture if and only if there exists a face F of P'_m with positive integer point $(x_1, \dots, x_{m-1}) \in F^\circ$ and $f \in [1, m-1]$ such that*

$$mx_i + i \le mx_f + f \quad \text{for every } i \ne f, \text{ and}$$
$$mx_f + f - m + 1 > e(F) \cdot (mx_f + f - m - (x_1 + \dots + x_{m-1}) + 1).$$

A face F of P'_m is *Wilf* if no interior point corresponds to a violation of Wilf's conjecture. A face F of C_m is *Wilf*, if the corresponding face in P'_m is Wilf.

Proposition 6.7 [2, p. 9] *Let F be a face of P'_m or C_m.*

- *If $e(F) > t(F)$, then F is Wilf.*
- *If $2e(F) \ge m$, then F is Wilf.*

Checking Wilf's conjecture for fixed multiplicity m can be done as follows:

1. For each face F in C_m check if Proposition 6.7 holds.
2. If Proposition 6.7 does not hold, check with Proposition 6.6 if the translated face in P'_m contains a point corresponding to a counterexample of Wilf's conjecture.

We say that a face F of C_m is *bad* if Proposition 6.7 does not hold. The group of units $(\mathbb{Z}/m\mathbb{Z})^\times$ acts on $\mathbb{R}^{m-1}$ by multiplying indices. The advantage of the Kunz cone over the (relaxed) Kunz polyhedron is that it is symmetric with respect to this action. Even more, $e(F)$ and $t(F)$ are invariant under this action. Thus in order to determine the bad faces, it suffices to determine for one representative of its orbit, if it is bad. We say that an orbit is *bad*, if all its faces are bad.

While [2] uses a modified algorithm of `normaliz` to determine all bad orbits, we replace this by the presented algorithm. We can also apply the symmetry of C_m. As described in Sect. 4.2, we can sort the elements by lexicographic comparison by the *coatom representation*. It suffices to visit the first facet of each orbit to see the first element of each orbit (and possibly more). The concrete implementation is available as a branch of `SageMath`[7]. This implementation worked well enough to apply the presented algorithm to Wilf's conjecture. In Table 1 we compare the runtimes of computing the bad orbits.

[7] See https://git.sagemath.org/sage.git/tree/?h=u/gh-kliem/KunzConeWriteBadFaces.

Table 1 Runtime comparison for determining the bad orbits

m	# Bad orbits	`normaliz`	`SageMath`
15	180,464	3:33 m	7 s
16	399,380	54:39 m	1:14 m
17	3,186,147	19:35 h	16:55 m
18	17,345,725	27:13 d	16:22 h
19	100,904,233		14:22 d

Table 2 Number of bad orbits to check and time it took to verify Wilf's conjecture for them

m	# Orbits	Time
15	193	1 s
16	5669	11 s
17	7316	31 s
18	17,233	1:54 m
19	285,684	2:22 h

These are performed on an Intel® Xeon™ CPU E7-4830 @ 2.20 GHz with a total of 1 TB of RAM and 40 cores. We used 40 threads and about 200 GB of RAM. The timings in [2] used only 32 threads and a slightly slower machine.

While testing all bad faces takes a significant amount of time, a recent work by Eliahou has simplified this task.

Theorem 6.8 [4, Thm. 1.1] *Let S be a numerical semigroup with multiplicity m. If $3e(S) \geq m$ then S satisfies Wilf's conjecture.*

Checking the remaining orbits can be done quickly (we used an Intel® Core™ i7-7700 CPU @ 3.60 GHz x86_64-processor with four cores). For each of the orbits with $3e < m$, we have checked whether the corresponding region is empty analogously to the computation in [2]. See Table 2 for the runtimes. This computation yields the following proposition.

Proposition 6.9 *Wilf's conjecture holds for $m = 19$.*

Acknowledgements We thank Winfried Bruns and Michael Joswig for valuable discussions and for providing multiple relevant references. We further thank Jean-Philippe Labbé for pointing [2] to us and all participants of the trac ticket in `SageMath`[1] for stimulating discussions.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix A: Detailed Runtimes

We give, for each of the five computations, an example of how it is executed for the 2-simplex.

1 Compute cover relations and f-vector in `polymake` from vertex-facet incidences. To our knowledge, this applies the algorithm in [7].

```
polytope> new Polytope(VERTICES_IN_FACETS=>
                        [[0,1],[0,2],[1,2]])->F_VECTOR;
```

2 Compute the f-vector with `normaliz` (via `pynormaliz`, optional package of `SageMath`). This is the algorithm described in [2].

```
sage: P = polytopes.simplex(2, backend='normaliz')
sage: P._nmz_result(P._normaliz_cone, 'FVector')
```

3 Compute cover relations in `SageMath`. This is the algorithm **FaceIterator** with Proposition 4.4.

```
sage: C = CombinatorialPolyhedron([[0,1],[0,2],[1,2]])
sage: C._compute_face_lattice_incidences() # non-public
```

4 & 5 Compute f-vector in `SageMath` using **FaceIterator**.

```
sage: C = CombinatorialPolyhedron([[0,1],[0,2],[1,2]])
sage: C.f_vector()
```

For displaying the runtimes, we use the following notations:

- Δ_d for the d-dimensional simplex,
- $\mathscr{C}_{d,n}$ for the d-dimensional cyclic polytope with n vertices,
- $\mathscr{A}_d$ for the d-dimensional associahedron,
- $\mathscr{P}_d$ for the d-dimensional permutahedron,
- $\mathcal{H}$ for the 20-dimensional counterexample to the Hirsch conjecture,
- $\square_d$ for the d-cube,
- $\mathscr{B}_n$ for the $(n-1)^2$-dimensional Birkhoff polytope,
- P^{op} for the polar dual of a polytope P,
- $L(P)$ for the Lawrence polytope of P,
- K_n the Kunz cone in ambient dimension $n-1$, treated as a inhomogenous polyhedron of dimension $n-2$.

The runtimes of the five best-of-five computations on the various examples are as given in Table 3.

Table 3 Runtime comparison for obtaining the f-vector in `polymake` (1) and `normaliz` (2) to runtime in `SageMath`for all cover relations (3), f-vector (4), and improved algorithm for the f-vector (5). Table consists of best-of-five timings rounded to whole seconds and the number of atoms and coatoms, the dimension and the number of elements. “MOF” indicates that the process was killed due to memory overflow, and a dash indicates a runtime of less than half a second

	Time in s								
	(1)	(2)	(3)	(4)	(5)	$\min(n, m)$	$\max(n, m)$	d	$\lvert\varphi\rvert$
$\square_5 \star \square_5^{op}$	3	–	3	–	–	42	42	11	59,536
$\square_6 \star \square_6^{op}$	119	2	120	–	–	76	76	13	532,900
$\square_7 \star \square_7^{op}$	4771	31	4,744	1	–	142	142	15	4,787,344
$\square_8 \star \square_8^{op}$	MOF	478	MOF	18	11	272	272	17	43,059,844
$\mathscr{A}_4 \star \mathscr{A}_4^{op}$	3	—	1	–	–	56	56	9	39,204
$\mathscr{A}_5 \star \mathscr{A}_5^{op}$	537	2	61	–	–	152	152	11	817,216
$\mathscr{A}_6 \star \mathscr{A}_6^{op}$	186,000	61	5335	15	16	456	456	13	18,318,400
$\mathscr{A}_7$	1	–	–	–	–	35	1430	7	20,794
$\mathscr{A}_8$	21	–	2	–	–	44	4862	8	103,050
$\mathscr{A}_9$	589	2	39	1	–	54	16,796	9	518,860
$\mathscr{A}_{10}$	28,226	16	826	10	–	65	58,786	10	2,646,724
$\mathscr{A}_{11}$	MOF	237	MOF	196	5	77	208,012	11	13,648,870
$\mathscr{B}_5$	228	9	99	–	–	25	120	16	6,092,722
$\mathscr{B}_6$	MOF	MOF	MOF	2710	310	36	720	25	19,989,171,034
$\square_{10}$	3	–	–	–	–	20	1024	10	59,050
$\square_{11}$	19	–	2	–	–	22	2048	11	177,148
$\square_{12}$	122	1	8	–	–	24	4096	12	531,442
$\square_{13}$	893	2	37	1	–	26	8192	13	1,594,324

Table 3 continued

	Time in s								
$\square_{14}$	9558	9	195	3	–	28	16,384	14	4,782,970
$\square_{15}$	MOF	32	MOF	18	1	30	32,768	15	14,348,908
$\square_{16}$	MOF	154	MOF	111	3	32	65,536	16	43,046,722
$\square_{17}$	MOF	938	MOF	693	20	34	131,072	17	129,140,164
$\square_{18}$	MOF	MOF	MOF	4410	132	36	262,144	18	387,420,490
$\mathscr{C}_{10,20}$	33	–	2	–	–	20	4004	10	171,650
$\mathscr{C}_{10,21}$	65	–	4	–	–	21	5733	10	238,912
$\mathscr{C}_{10,22}$	128	1	6	–	–	22	8008	10	325,954
$\mathscr{C}_{10,23}$	260	1	10	–	–	23	10,948	10	436,864
$\mathscr{C}_{10,24}$	586	1	18	1	–	24	14,688	10	576,258
$\mathscr{C}_{10,25}$	1310	2	30	1	–	25	19,380	10	749,312
$\mathscr{C}_{10,26}$	2647	2	51	2	–	26	25,194	10	961,794
$\mathscr{C}_{10,27}$	5039	3	88	3	–	27	32,319	10	1,220,096
$\mathscr{C}_{10,28}$	9128	5	140	5	–	28	40,964	10	1,531,266
$\mathscr{C}_{10,29}$	15,624	7	230	7	–	29	51,359	10	1,903,040
$\mathscr{C}_{10,30}$	26,123	10	365	12	–	30	63,756	10	2,343,874
$\mathscr{C}_{10,31}$	41,903	14	568	19	1	31	78,430	10	2,862,976
$\mathscr{C}_{10,32}$	65,655	20	MOF	29	1	32	95,680	10	3,470,338
$\mathscr{C}_{10,33}$	100,307	29		44	1	33	115,830	10	4,176,768
$\mathscr{C}_{10,34}$	150,241	40		66	2	34	139,230	10	4,993,922
$\mathscr{C}_{10,35}$	222,340	57		98	3	35	166,257	10	5,934,336
$\mathscr{C}_{10,36}$		80		161	5	36	197,316	10	7,011,458
$\mathscr{C}_{10,37}$		110		231	7	37	232,841	10	8,239,680

Table 3 continued

	Time in s								
$\mathscr{C}_{10,38}$		152		329	10	38	273,296	10	9,634,370
$\mathscr{C}_{10,39}$		209		463	16	39	319,176	10	11,211,904
$\mathscr{C}_{10,30}$		287		642	25	40	371,008	10	12,989,698
$\mathscr{C}_{10,41}$		385		889	40	41	429,352	10	14,986,240
$\mathscr{C}_{10,42}$		MOF		1216	67	42	494,802	10	17,221,122
$\mathcal{H}$	MOF		MOF	453		40	36,425	20	353,731,266
$\mathscr{P}_6$	8	1	3	–	–	62	720	5	4,684
$\mathscr{P}_7$	2179	17	428	6	–	126	5040	6	47,294
Δ_{16}	2	–	1	–	–	17	17	16	131,072
Δ_{17}	5	–	2	–	–	18	18	17	262,144
Δ_{18}	10	1	5	–	–	19	19	18	524,288
Δ_{19}	22	1	10	–	–	20	20	19	1,048,576
Δ_{20}	50	3	23	–	–	21	21	20	2,097,152
Δ_{21}	113	5	52	–	–	22	22	21	4,194,304
Δ_{22}	252	11	115	–	–	23	23	22	8,388,608
Δ_{23}	548	21	253	1	–	24	24	23	16,777,216
Δ_{24}	MOF	44	553	2	–	25	25	24	33,554,432
Δ_{25}	MOF	91		3	–	26	26	25	67,108,864
Δ_{26}	MOF	189		6	1	27	27	26	134,217,728
Δ_{27}	MOF	MOF		13	2	28	28	27	268,435,456
	(1)	(2)	(3)	(4)	(5)	$\min(n,m)$	$\max(n,m)$	d	φ
$\mathscr{C}_{20,21}$	48	3	297	–	–	21	21	20	2,097,152

Table 3 continued

	Time in s								
$\mathscr{C}_{20,22}$	260	6	MOF	–	–	22	121	20	4,190,210
$\mathscr{C}_{20,23}$	MOF	12	MOF	1	–	23	506	20	8,341,504
$\mathscr{C}_{20,24}$	MOF	26	MOF	2	–	24	1716	20	16,474,114
$\mathscr{C}_{20,25}$		44		6	1	25	5005	20	32,120,832
$\mathscr{C}_{20,26}$		110		24	3	26	13,013	20	61,554,690
$\mathscr{C}_{20,27}$		269		107	9	27	30,888	20	115,546,112
$\mathscr{C}_{20,28}$		811		431	28	28	68,068	20	212,004,866
$\mathscr{C}_{20,29}$		MOF		1790	98	29	140,998	20	379,838,464
$L(\mathscr{A}_3)$	625	17	209	1	–	28	416	17	10,435,664
$L(\mathscr{C}_{4,8})$	1	–	–	–	–	16	148	12	50,746
$L(\mathscr{C}_{4,9})$	5	–	1	–	–	18	261	13	167,098
$L(\mathscr{C}_{4,10})$	23	1	6	–	–	20	430	14	524,800
$L(\mathscr{C}_{4,11})$	102	2	21	–	–	22	671	15	1,582,332
$L(\mathscr{C}_{4,12})$	412	7	73	–	–	24	1002	16	4,606,876
$L(\mathscr{C}_{4,13})$	1696	21	249	1	–	26	1443	17	13,015,500
$L(\mathscr{C}_{4,14})$	MOF	70	803	5	1	28	2016	18	35,829,622
$L(\mathscr{C}_{5,8})$	2	–	–	–	–	16	120	13	61,456
$L(\mathscr{C}_{5,9})$	9	–	2	–	–	18	261	14	224,330
$L(\mathscr{C}_{5,10})$	50	1	8	–	–	20	514	15	782,596
$L(\mathscr{C}_{5,11})$	263	5	36	–	–	22	935	16	2,614,020
$L(\mathscr{C}_{5,12})$	1339	16	146	1	–	24	1596	17	8,390,656
$L(\mathscr{C}_{5,13})$	MOF	50	554	4	1	26	2587	18	25,990,044

Table 3 continued

	Time in s								
$L(\mathscr{C}_{5,14})$	MOF	144	MOF	14	2	28	4018	19	77,999,464
$L(\mathscr{C}_{4,6}^{\mathrm{op}})$	2	–	1	–	–	18	57	13	121,894
$L(\mathscr{C}_{4,7}^{\mathrm{op}})$	MOF	53	510	2	–	28	672	18	24,233,912
$L(\mathscr{C}_{4,8}^{\mathrm{op}})$	MOF	MOF	MOF	1270	159	40	4208	24	7,188,015,356
$L(\mathscr{C}_{5,7}^{\mathrm{op}})$	156	8	67	–	–	24	110	17	4,577,866
$L(\mathscr{C}_{5,8}^{\mathrm{op}})$	MOF	MOF	MOF	3563	434	40	5,928	25	17,364,262,196
$L(\square_5^{\mathrm{op}})$	12	1	5	–	–	20	84	15	479,566
$L(\square_6^{\mathrm{op}})$	239	10	94	–	–	24	152	18	5,909,086
$L(\square_7^{\mathrm{op}})$	MOF	124	1552	5	1	28	284	21	72,097,678
$L(\square_8^{\mathrm{op}})$	MOF	MOF	MOF	65	10	32	544	24	873,869,950
$L(\square_4)$	MOF	255	3525	10	1	32	296	20	130,851,046
$\mathscr{K}_{12}$		2			–	60	1864	10	669,794
$\mathscr{K}_{13}$		16			–	72	7005	11	4,389,234
$\mathscr{K}_{14}$		137			1	84	15,585	12	21,038,016
$\mathscr{K}_{15}$		2,622			21	98	67,262	13	137,672,474
$\mathscr{K}_{16}$		MOF			241	112	184,025	14	751,497,188

References

1. Bayer, M.M., Hetyei, G.: Generalizations of Eulerian partially ordered sets, flag numbers, and the Möbius function. Discrete Math. **256**(3), 577–593 (2002)
2. Bruns, W., García-Sánchez, P., O'Neill, C., Wilburne, D.: Wilf's conjecture in fixed multiplicity. Int. J. Algebra Comput. **30**(4), 861–882 (2020)
3. Bruns, W., Ichim, B., Römer, T., Sieg, R., Söger, Ch.: Normaliz. Algorithms for rational cones and affine monoids. https://www.normaliz.uni-osnabrueck.de
4. Eliahou, S.: A graph-theoretic approach to Wilf's conjecture. Electron. J. Comb. **27**(2), # P2.15 (2020)
5. Gawrilow, E., Joswig, M.: polymake: a framework for analyzing convex polytopes. In: Polytopes—Combinatorics and Computation (Oberwolfach 1997). DMV Sem., vol. 29, pp. 43–73. Birkhäuser, Basel (2000)
6. Hampe, S., Joswig, M., Schröter, B.: Algorithms for tight spans and tropical linear spaces. J. Symb. Comput. **91**, 116–128 (2019)
7. Kaibel, V., Pfetsch, M.E.: Computing the face lattice of a polytope from its vertex-facet incidences. Comput. Geom. **23**(3), 281–290 (2002)
8. Lemire, D., Kaser, O., Kurz, N., Deri, L., O'Hara, C., Saint-Jacques, F., Ssi-Yan-Kai, G.: Roaring bitmaps: implementation of an optimized software library. Software **48**(4), 867–895 (2018)
9. Oxley, J.G.: Matroid Theory. Oxford Science Publications. Oxford University Press, New York (1992)
10. Ziegler, G.M.: Lectures on Polytopes. Graduate Texts in Mathematics, vol. 152. Springer, New York (1995)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.