FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

About a graph-theoretical algorithm for approximate vertex-enumeration

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
im Studiengang Mathematik

eingereicht am 29. August 2021 von

David Hartel

Erstgutachter:
Prof. Dr. Löhne
Zweitgutachter:
Dr.rer.nat. Weißing

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

Abstract

This thesis refers to Andreas Löhne's article *Approximate Vertex Enumeration* published in 2020. There, Löhne addresses the problem of computing a V-polytope that is close to a given H-polytope P and develops an approximate variant of Motzkin's Double Description Method. For the correctness proof of dimension three, he presents a graph-theoretical algorithm.
In this thesis, Löhne's algorithm is used to state a graph-theoretical version for approximate vertex enumeration that relies more on the polytopes combinatorics and less on numerical computations. For the practical implementation of the algorithm, an edge-centered data structure is used, which allows for the efficient manipulation of planar graphs.
Finally, we compare the implementation with an implementation of Löhne's initial algorithm. The graph-theoretical algorithm computes fewer vertices and thus leads to simpler polytopes. However, the runtime can not compete with the runtime of the initial algorithm. The practical part consists of the implementation of the data structure and the algorithms.

Zusammenfassung

Diese Bachelorarbeit bezieht sich auf Andreas Löhnes 2020 erschienenen Artikel *Approximate Vertex Enumeration*. In Löhnes Artikel wird das Problem, ein V-polytop zu berechnen, welches nah an einem gegebenenen H-Polytop liegt, diskutiert und eine approximative Variante von Motzkins Double Description Methode vorgestellt. Aus dem Korrektheitsbeweis für Dimension drei entspringt ein graphentheoretischer Algorithmus.
In dieser Arbeit wird dieser Algorithmus als Algorithmus zur approximativen Ecken-Enumeration formuliert. Er basiert mehr auf der kombinatorischen Struktur des Polytops und beruht weniger auf numerischen Berechnungen. Die Implementierung erfolgt unter Verwendung einer Halfedge-Datenstruktur, welche eine effiziente Bearbeitung von planaren Graphen ermöglicht. Abschließend wird die Implementierung mit einer des ursprünglichen Algorithmus von Löhne verglichen. Während der graphentheoretische Algorithmus weniger Ecken und damit simplere Polytope erzeugt, kann sich seine Laufzeit nicht mit der des ursprünglichen Algorithmus messen. Der praktische Teil dieser Arbeit besteht in der Implementierung der Datenstruktur und der Algorithmen.

Contents

List of figures

1 Introduction

There are different ways to represent a convex polytope. Two fundamental representations will be introduced, following [Zie95, pp. 27–29] and [HJZ99].
An H-polyhedron is a set $P \in \mathbb{R}^d$ is as intersection of finitely many closed halfspaces represented in the form:

$$P := \{x \in \mathbb{R}^d \mid Ax \leq b\}$$

where $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$.
An H-polytope is an H-polyhedron that is bounded in the sense that there is a constant N such that $||x|| \leq N$ holds for all $x \in P$.
A V-Polytope is the convex hull of a finite set $\mathcal{V} = \{x_1, ..., x_n\}$ of points in $\mathbb{R}^d$:

$$P = \text{conv}(\mathcal{V}) := \{\sum_{i=1}^{n} \lambda_i x_i \mid \lambda_i \geq 0, \sum_{i=1}^{n} \lambda_i = 1\}$$

Lower dimension examples illustrate the two concepts well: An H-polytope is constructed by intersecting halfspaces and a V-polytope is constructed by the convex hull of points in the space (see figure 1).

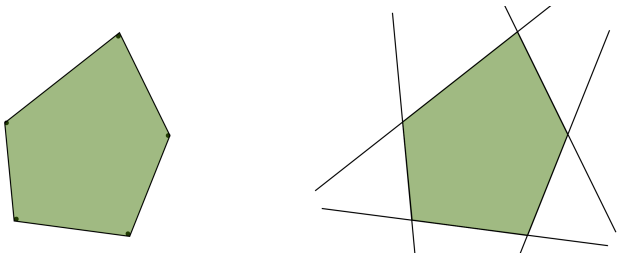


Figure 1: The two polytope representation types: V-representation and H-representation of one and the same polytope

The following *Main Theorem of Polytope Theory* proves, that both characterizations are mathematically equivalent.

Theorem 1.1 ([HJZ99])**.** *The definitions of V-polytopes and of H-polytopes are equivalent. That is, every V-polytope has a description by a finite system of inequalities, and every H-polytope can be obtained as the convex hull of a finite set of points (its vertices).*

The problem of determining the vertices of a given H-polytope is known as the *vertex enumeration*. Although the result might seem "geometrically reasonable", the computational step from one representation to the other is far from trivial [HJZ99, p. 244]. There exists a vast amount of literature related to this problem as, for example, Motzkin's double description algorithm [FP96]. Still, theoretically correct geometric algorithms do not necessarily lead to valid computer programs. This is due to the fact that those algorithms are mostly designed for a machine model with exact real arithmetic. Implementations using imprecise arithmetic as floating point arithmetic instead of costly exact arithmetic can

cause numerical errors. Examples as [Ket+04] show that numerical errors in geometrical algorithms can lead to inconsistencies in the combinatorial structure.
In the article "Approximate Vertex Enumeration" [Löh20] Löhne refers to such obstacles in the implementation of Motzkin's Double Description Method (DDM) and raises the question if an approximate variant of the DDM is easier to realize than exact vertex enumeration. Further, Löhne provides an algorithm for the so-called *approximate vertex enumeration* which is correct for dimension two and three even under the use of imprecise arithmetic [Löh20, p. 1]. By showing the correctness for dimension three, Löhne developed a cutting scheme for planar graphs, whose vertices refer to the vertices of a polytope. [Löh20, p. 14]. This cutting scheme represents one iteration in the final algorithm.

Our goal is to use this initially as a side product emerged scheme, to reformulate a graph-based algorithm for the approximate vertex enumeration. It can be seen as an advantage, that this algorithm relies mostly on the combinatorial structure of a polytope and uses its metric information, i.e. its 3D-coordinates for the vertices and linear inequalities for its faces only as side information. In this way, the dependency on numerical computations is reduced.
Such a general separation of the combinatorial information from the numerical information is further described and discussed by Sugihara in *Topology-Oriented Implementation—An Approach to Robust Geometric Algorithms* [Sug+00], where three different classes of algorithms are described: In the first category "approaches rely on the numerical computation "moderately."" in the sense, that imprecise arithmetic is used, but the computational error is assumed to be bounded; in the second category "approaches rely on the numerical computation "completely"", in the sense that they use exact arithmetic and thus always obtain correct results and in the third category, "The approaches rely on numerical values the least, i.e. they are robust even if numerical errors are large" [Sug+00, pp. 5–6]. The algorithm we want to present in this work can be seen as an approach of the third category.
By implementing the algorithm we have to use an appropriate data structure for planar graphs, which allows for a time- and space-efficient and ideally also comfortable implementation. Therefore, an edge-based data structure will be introduced, which fulfills these requirements.
Finally, our goal is to compare the implemented algorithm with Löhne's original algorithm.

1.1 Approximate Vertex Enumeration

Löhne addresses in "Approximate Vertex enumeration" [Löh20] the problem of computing a V-polytope, which is close to a given H-polytope. Following the notation of [Löh20], we will introduce the concept of the approximate vertex enumeration.

Since every H-Polytope can be shifted, so that 0 lies in its interior, we assume that

$$P = \{x \in \mathbb{R}^d \mid Bx \leq b\}$$

is an H-Polytope with zero in its interior, given by a matrix $B \in \mathbb{R}^{m \times d}$ and a vector $b \in \mathbb{R}^d$. By setting $A_{i,j} \coloneqq \frac{B_{i,j}}{b_i}$ for $i = 1...m,\ j = 1...d$ we can express P as

$$P = \{x \in \mathbb{R}^d \mid Ax \leq e\}$$

with $e = (1, ..., 1)^T$ as the all-ones vector with the dimension given by the context.
For a tolerance $\epsilon > 0$, we are now able to define an ϵenvironment. We set

$$(1+\epsilon)P \coloneqq \{x \in \mathbb{R}^d \mid Ax \leq (1+\epsilon)e\}.$$

as the "inflated" version of P. This notation allows us to introduce the concept of an approximate V-polytope Q of P:

Definition 1.2 ([Löh20])**.** *An* ϵ-approximate V-representation *is a finite set* $\mathcal{V} = \{v_1, ..., v_k\} \in \mathbb{R}^d$ *such that the V-Polytope* $Q \coloneqq conv(\mathcal{V})$ *satisfies*

$$P \subseteq Q \subseteq (1+\epsilon)P.$$

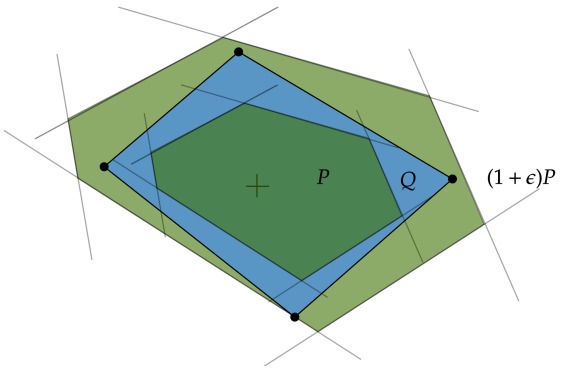


Figure 2: Two-dimensional example of an approximate V-polytope Q, lying between P and $(1+\epsilon)P$. Q is not required to have the same combinatorial structure. Q consists of four vertices whereas P has six vertices.

Given an H-Polytope P, the goal of the approximate vertex enumeration is to construct iteratively an ϵ-approximate V-representation Q.

According to Löhne, approximate vertex enumeration can be motivated by the fact that vertex enumeration is usually not stable concerning imprecise computations, for instance, caused by floating point arithmetic.[Löh20, p. 2] The result of the method developed by him is a V-polytope, that approximates the given H-polytope by a prescribed tolerance. The topology can differ between the H-polytope and the resulting V-polytope and the combinatorial information will not be preserved. This fact can be seen as an advantage since an approximate V-polytope with fewer vertices than the given H-representation could be obtained with less effort compared to an exact vertex approximation [Löh20, p. 2].
In the following, we will present the general idea of the approximate algorithm and further focus on the results for the 3-dimensional case, which will yield the graph algorithm. Following "Approximate Vertex Enumeration" [Löh20, chapter 4] we will discuss the basic cutting scheme of the approximate algorithm of the DDM. For $\epsilon = 0$, the basic cutting scheme is similar to a typical iteration step of the DDM.

2 A basic cutting scheme

Given a half-space $H_{\leq} := \{x \in \mathbb{R}^d \mid h^T x \leq 1\}$, defined by an $h \in \mathbb{R}^d \setminus \{0\}$ and an ϵ-approximative V-representation $\mathcal{V}$ of P we aim to compute by the cutting scheme an ϵ-approximative V-representation $\mathcal{V}'$ of $P' := P \cap H_{\leq}$. We divide the space into three subsets:

$$\begin{aligned} H_+ &:= \{x \in \mathbb{R}^d \mid h^T x > 1 + \epsilon\} \\ H_- &:= \{x \in \mathbb{R}^d \mid h^T x < 1\ \} \\ H_0\ &:= \{x \in \mathbb{R}^d \mid 1 \leq h^T x \leq 1 + \epsilon\} \end{aligned}$$

Now we divide up $\mathcal{V}$ into $\mathcal{V}_+ := \mathcal{V} \cap H_+$, $\mathcal{V}_- := \mathcal{V} \cap H_-$ and $\mathcal{V}_0 := \mathcal{V} \cap H_0$ respectively.
A vertex $v \in \mathcal{V}_+$ does not lie in $(1+\epsilon)P'$ by definition of P'. Thus v needs to be removed and replaced by at least one new vertex $v' \in \mathcal{V}$ by the cutting scheme. The crucial question is, how we can decide where a new vertex v' has to be placed. In other words, to decide, if the line segment $\text{conv}(v_1, v_2)$ between a pair of vertices $(v_1, v_2) \in \mathcal{V}_- \times \mathcal{V}_+$ lies outside of P and thus allows us to insert a new vertex on $\text{conv}(v_1, v_2) \cap H_0$. Vertices of $\mathcal{V}$ in H_- and H_0 will not be changed as they lie in the $(1+\epsilon)P'$.
To discuss this question, we need to introduce another concept, which describes the relation between vertices of P and points outside of P.
For a matrix $A \in \mathbb{R}^{m \times d}$ and a subset $I \subseteq \{1...m\}$, we describe by A_I the submatrix of A which consists of the rows of A with indices in I. For a single row, i.e. $I = \{i\}$, we write A_i instead of $A_{\{i\}}$. Further, for $\prec \in \{>, \geq, =, \neq, <, \leq\}$ and $u \in \mathbb{R}^d$ we define

$$J_{\prec}(u) := \{i \in \{1...m\} \mid A_i u \prec 1\}$$

That is, J selects the rows of A, which fulfill a certain relation regarding u.

Definition 2.1 ([Löh20]). *A vertex u of P is* covered *by a point $v \in \mathbb{R}^d$ if*

$$u \in conv(vert(P) \setminus \{u\} \cup \{v\}),$$

where $vert(P)$ denotes the set of vertices of P and $conv(M)$ is the convex hull of a set M.

The following proposition gives a more intuitive description of the concept: A vertex is covered by a point, if the point lies outside of the half-spaces, in which the vertex lies:

Proposition 2.2 ([Löh20]). *For $u \in vert(P)$ and $v \in \mathbb{R}^d$, it is equivalent:*

(i) *u covers v*

(ii) $J_=(u) \subseteq J_\geq(v)$

Proof. For the proof we refer the reader to [Löh20, p. 4] □

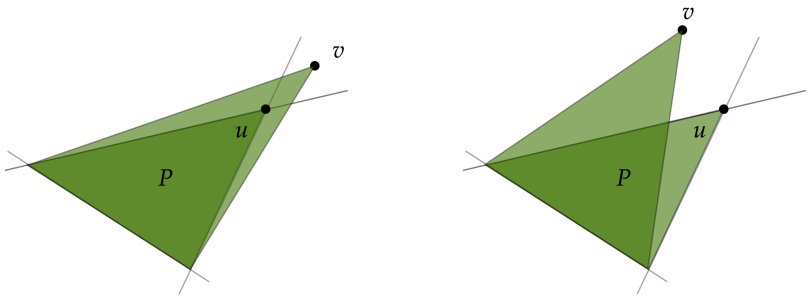


Figure 3: Two-dimensional illustration of the concept. A point v covers a vertex u if and only if v lies outside of the half-spaces, for which u lies on the border. On the left: v covers u. On the right: v does not cover u

This concept allows us to formulate a necessary condition for two vertices u_1, u_2 of P being endpoints of an egde of P: If u_1 and u_2 are endpoints of an egde of P, then u_1 and u_2 have at least $d-1$ common inequalities. i.e. $|J_=(u_1) \cap J_=(u_2)| \geq d-1$. If v_1 covers u_1 and v_2 covers u_2 we get the necessary condition

$$|J_\geq(v_1) \cap J_\geq(v_2)| \geq d-1.$$

The following algorithm represents Löhne's basic cutting scheme. For $\epsilon = 0$, the algorithm is similar to a typical iteration step of the DDM.

Algorithm 1: Basic cut

input : Polytope P given by $A \in \mathbb{R}^{m \times d}$, $\epsilon > 0$,
half-space $H_{\leq} := \{x \mid h^T x \leq 1\}$,
ϵ-approximate V-representation $\mathcal{V}$ of P

output : ϵ-approximate V-representation $\mathcal{V}'$ of $P' = P \cap H_{\leq}$,

begin
  $\mathcal{V}_+ \leftarrow \mathcal{V} \cap H_+ = \{v \in \mathcal{V} \mid h^T v > 1 + \epsilon\}$
  $\mathcal{V}_- \leftarrow \mathcal{V} \cap H_- = \{v \in \mathcal{V} \mid h^T v < 1\ \}$
  $\mathcal{V}_0 \leftarrow \mathcal{V} \cap H_0 = \{v \in \mathcal{V} \mid 1 \leq h^T v \leq 1 + \epsilon\}$
  for $(v_1, v_2) \in \mathcal{V}_- \times \mathcal{V}_+$ **do**
    if $|J_{\geq}(v_1) \cap J_{\geq}(v_2)| \geq d - 1$ **then**
      compute $v \in \text{conv}\{v_1, v_2\} \cap H_0$
      $\mathcal{V} \leftarrow \mathcal{V} \cup \{v\}$
    end
  end
  $V' \leftarrow \mathcal{V} \setminus \mathcal{V}_+$
end

Computing an ϵ-approximative V-representation requires a repeated application of the cutting scheme. As illustrated in figure 4 and 5, algorithm 1 can fail: If $\mathcal{V}$ is an ϵ-approximative V-representation, the output $\mathcal{V}'$ is not necessarily one. Therefore, we introduce a stronger concept by making assumptions about the location of the vertices in $\mathcal{V}$.

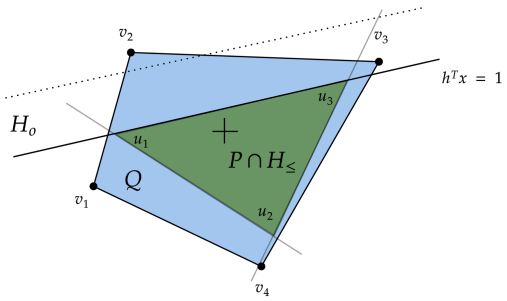


Figure 4: Algorithm 1 does not see edge v_1v_2 as it is $J_{\geq}(v_1) \cap J_{\geq}(v_2) = 0 < 1$

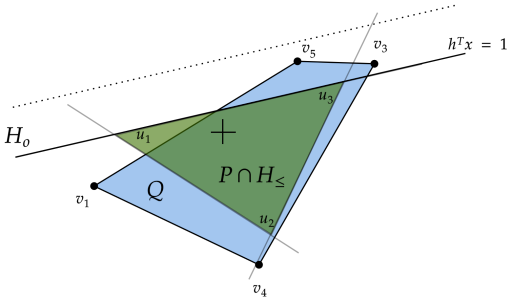


Figure 5: $V = \{v_1, v_2, v_3, v_4\ \}$ is no ϵ-approximate representation since $u_1 \notin Q = conv(v_1, v_2, v_3, v_4)$

Definition 2.3 ([Löh20]). *A finite set $\mathcal{V} \in (1 + \epsilon)P$ is called* strong ϵ-approximate V-representation *of P if every vertex of P is covered by some element of $\mathcal{V}$.*

As illustrated in figure 6 and 7, the property of $\mathcal{V}$ being a strong ϵ-approximate V-representation is not invariant under algorithm 1, as the output can be a ϵ-approximate V-representation, but not a strong one.

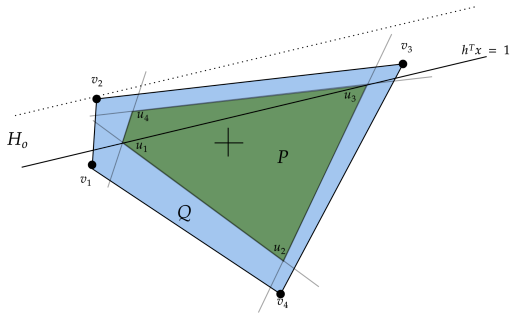


Figure 6: $\mathcal{V} = (v_1, v_2, v_3, v_4)$ is a strong ϵ-approximative V-representation of P. Algorithm 1 does not change the edge v_1v_2 as v_2 lies in H_0.

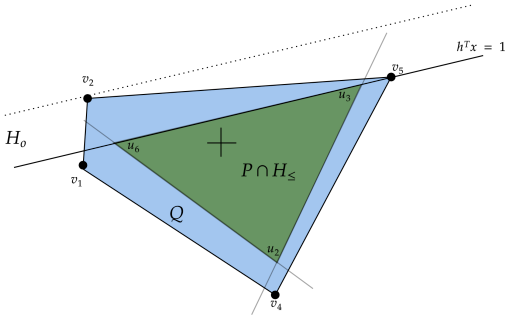


Figure 7: $\mathcal{V}' = (v_1, v_2, v_4, v_5)$ is not a strong ϵ-approximative V-representation as u_6 is not covered by any vertex in Q

We refer the reader to [Löh20, p. 6] for an even stronger assumption, which ensures that the output of algorithm 1 remains a strong ϵ-approximate V-representation. By modifying the output before every new application of algorithm 1, so that the stronger assumption is fulfilled, the correctness of algorithm 1 under iterative use can be proven.
For dimension three, Löhne provides an invariant property, which consequently does not need any modification step before a repeated application.

3 The graph algorithm

The idea is to focus on the topological information of a polytope and observe the effect of algorithm 1 on the combinatorial structure. Following [Löh20, p. 14] we will present an invariant property of a polytope under the transition of its combinatorial structure affected by algorithm 1. This invariant property justifies the application of algorithm 1 without any modification step.

3.1 Graph theory fundamentals

As the combinatorics of a polytope will be represented by a planar graph, some basic graph theory is required. We will follow the books from Diestel [Die06] and Ziegler [Zie95], to introduce some fundamental properties:

Definition 3.1 (Graph)**.** *A* graph G *is a pair* (V, E)*, where* V *is a finite set of* vertices *and* E *is the set of* edges*,* $E \subseteq [V]^2$*.*

For simplicity, an edge $\{u, v\}$ is usually written as uv (or vu).

Definition 3.2 (Path)**.** *A* path *in* G *is a subgraph* $P = (V', E')$ *of* G *of the form* $V' = v_0, v_1, ..., v_k$ $E' = v_0v_1, v_1v_2, ..., v_{k-1}v_k$*, where the* v_i *are all distinct vertices.*

Often, a path is written as the natural sequence of its vertices, that is, $P = v_0v_1, ..., v_k$, and called a path from v_0 to v_k.

Definition 3.3 (Cycle)**.** *If $P = v_0, ..., v_{k-1}$ is a path and $k \leq 3$, then the subgraph $C := P + x_{k-1}x_0$ is called a* cycle*.*

Definition 3.4 (Connectivity)**.** *The graph G is called* connected *if any two of its vertices are linked by a path in G. Further, G is called k-connected, if G has at least $k+1$ vertices and the removal of $k-1$ or fewer vertices leaves the graph connected.*

Definition 3.5 (Component)**.** *A maximal connected subgraph of G is called a* component*.*

By the *Global Version of Menger's Theorem* [Die06, Theorem 3.3.6.] a graph is k-connected if and only if it contains k independent paths between any two vertices.

Definition 3.6 (Embedding, Drawing)**.** *An* embedding *or* drawing *is a mapping of a graph G into the sphere S^2. That is, by an injective function $\phi : G \to S^2$, each vertex $v \in V$ is mapped to a distinct point $\phi(v) \in S^2$ and each edge uv is mapped to an arc $\phi(uv)$ from $\phi(u)$ to $\phi(v)$ so that no other vertex is mapped to the interior of $\phi(uv)$.*

Definition 3.7 (Planar graph)**.** *A graph G is a* planar graph *if there exists a crossing-free drawing, that is, a drawing in which no two arcs intersect except at common endpoints.*

Definition 3.8. *The embedded graph of a planar graph G is called a* plane graph*.*

We will use the name G of an abstract, planar graph also for his embedded *plane graph* as it defines G in a natural and definite way.

Definition 3.9. *A plane graph G devides the plane into the* faces *of G, that is a finite collection of open sets $S^2 \setminus G$, which are bounded by arcs.*

Definition 3.10. *The* boundary *of a face is the union of those bounding arcs.*

In a 2-connected plane graph G, every face is bounded by a cycle [Die06, Proposition 4.2.6] and an arc lies on the boundary of exactly two faces of G [Die06, Proposition 4.2.6. and Lemma 4.2.2].

3.2 Basic cutting scheme

For some arbitrary dimension, the combinatorial structure of a polytope P can be described by a graph G in the way that vertices and edges of P are associated by vertices and edges in G respectively. For a three-dimensional polytope P, additionally the faces of P can be represented by faces of a planar graph G. If there exists such a distinct mapping between G and P, we call G the graph $G(P)$ of P. Steinitz (1871-1928) provides a fundamental relationship between planar graphs and convex polyhedra with his theorem, i.e. convex 3-polytopes [Zie95, p. 103]:

Theorem 3.11 (Steinitz' theorem)**.** *G is the graph of a three-dimensional polytope P if and only if it is simple, planar and three-connected.*

It seems obvious that the graph $G(P)$ is simple, as an edge of a polytope has two distinct endpoints and thus G does not contain any loop. Further G is planar, as we can obtain a planar drawing of G by projecting P by rays, outgoing from an interior point to the sphere S^2. Finally, it is easy to see that every vertex of P has at least 3 adjacent edges and hence it seems plausible that the graph is three-connected.
The other direction, from a simple, planar three-connected graph G to a polytope P is more difficult. For the proof, we refer the reader to [Zie95, p. 104].

Later, we will see that the graph algorithm does not necessarily represent a convex polytope. Non-convex bounding cycles can occur, which finally could produce separate subgraphs that are not connected (one can think of a non-convex polytope with two peaks, where after a cut only the peaks remain). Thus, for the graph algorithm, a more general 2-connected plane graph is considered, with all its components being two-connected.
Löhne [Löh20, p. 14] considers a map $f : V \to \mathcal{V}$ from the vertices of such a plane graph $G = (E, V)$ with all its components being two-connected to an ϵ-approximate V-representation $\mathcal{V}$. He defines further a graph $G' = (E', V')$ with all its components being two-connected and a map $f' : V' \to \mathcal{V}'$, where $\mathcal{V}'$ is the result of $\mathcal{V}$ under algorithm 1. The invatiant property under the transition from G to G' and f to f' will be illustrated in chapter 3.3. The transition from G to G' is described in [Löh20, p. 14] as a pure graph-theoretical cutting scheme and will now be presented as algorithm 2. In order to state the algorithm, we first will introduce further properties of 2-connected graphs and terms for the edges and vertices that will be effected by cutting off some vertices.

Definition 3.12 ([Löh20])**.** *Let G be a 2-connected plane graph. Then, a cycle C of G, which belongs to the boundary of some face of G is called a* bounding cycle*.*

Proposition 3.13 ([Löh20])**.** *In a plane graph G with each component being 2-connected, every face is bounded by a set of cycles.*

Proof. Since in a 2-connected component every face is bounded by a cycle, the cycles of G are exactly the cycles of the components of G. By planarity, every component must be contained in some face of another component. Since faces of components are bounded by cycles, the claim follows. □

In a plane graph G with each component being 2-connected, a *bounding cycle* is a cycle C in G, which belongs to the boundary of some face of G.

Definition 3.14 ([Löh20])**.** *For a set $Z \subseteq V$, the set*

$$\delta(Z) := \{uv \in E \mid u \in Z, v \in V \setminus Z\}$$

is called cut *generated by Z*

Definition 3.15 ([Löh20]). *Two edges $e, f \in \delta(Z)$ are called* neighbors *if there is a path with first edge e and last edge f which belongs to exactly one bounding cycle and has internal vertices in Z only.*

Both terms describe intuitively what happens to the planar graph of a polytope when vertices will be cut off: The corresponding vertices, represented by Z will be removed from the graph. Edges with exactly one endpoint in Z will be cut, thus they are said to be in the cut $\delta(Z)$. The term neighbor defines an adjacency relation among those edges. As an edge $e \in \delta(Z)$ belongs to the boundary of exactly two faces, e belongs to exactly two bounding cycles and can have at most two neighbors.

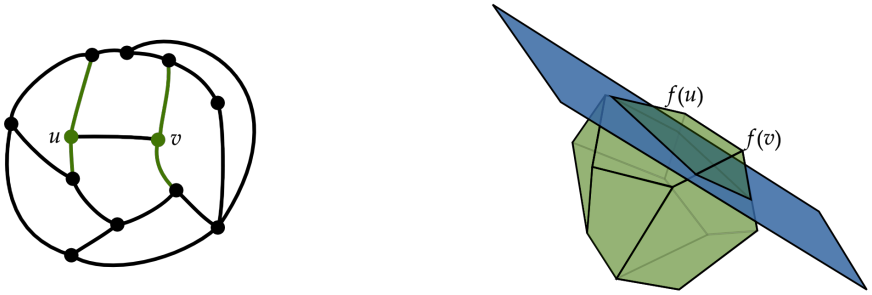


Figure 8: On the left: A planar, two-connected graph $G = (E, V)$. The vertices u and v in Z and the corresponding edges in $\delta(Z)$ are colored green. On the right: The vertices of G under the corresponding map. The illustrated cut with the hyperplane $H_{\leq}$ motivates the partition in Z and $\mathcal{V} \setminus Z$

With the definition of a cut and a neighbor, we can formulate algorithm 2, which represents in a way the equivalent to algorithm 1 for an as planar graph embedded polytope.

Algorithm 2: Basic graph cutting scheme

input : plane graph $G = (E, V)$ whose components are 2-connected, ;
Cut $\delta(Z)$, generated by a non-empty set of vertices $Z \subseteq V$
output : plane graph $G' = (E', V')$ whose components are 2-connected

1 **begin**
2 **for** $e \in \delta(Z)$ **do**
3 add a new vertex v_e to V, draw it on the arc and replace $e = uv \in E$ by uv_e and $v_e v$
4 **end**
5 **for** $e, f \in \delta(Z)$ **do**
6 **if** e, f *are neighbors* **and** $v_e v_f \notin E$ **then**
7 add $v_e v_f$ to E. Draw the corresponding arc inside a common face
8 **end**
9 **if** e, f *are neighbors* **and** e, f *have no other neighbors* **then**
10 add a new vertex w to V and two edges $v_e w$, $w v_f$ to E
11 **end**
12 **end**
13 $V' \leftarrow V \setminus Z$
14 $E' \leftarrow E \setminus \{uv \in E \mid u \in Z \vee v \in Z\}$
15 **end**

Löhne proofs the correctness and some further properties by the following Lemma [Löh20, p. 15]:

Lemma 3.16. *Let G be a plane graph with all components being 2-connected and let $Z \subseteq V$ be an arbitrary non-empty set of vertices. The result of Algorithm 2 is a plane graph $G' = (V', E')$ with all components being 2-connected. The set of edges added in line 7 and 12 is the edge set of finitely many bounding cycles in G'.*
Let $W \subseteq V'$ denote the set of vertices added in line 11 and let $\bar{G} = (\bar{V}, \bar{E})$, denote the graph arising from G after the lines 2-4 have been executed. For the vertices $V(C')$ of a bounding cycle C' in G', either $V(C') \cap V = \emptyset$ or there is a bounding cycle $\bar{C}$ in $\bar{G}$ such that $V(C') \setminus W$ are vertices in $\bar{C}$.

For the proof, we refer to [Löh20, p. 15].

Example. *Given a graph G consisting of six vertices as illustrated in figure 9, the algorithm locates all corresponding edges in $\delta(Z)$ in the first step. In the second step, i.e. in the loop beginning with line 2, every edge in $\delta(Z)$ will be divided by a newly added vertex. In the loop beginning with line 5, two of those newly added vertices subsequently will be connected by a new edge if the corresponding edges are neighbors. Finally, in the last step, every vertex in Z as well as its adjacent edges will be removed from the graph. The output is given by the remaining graph.*

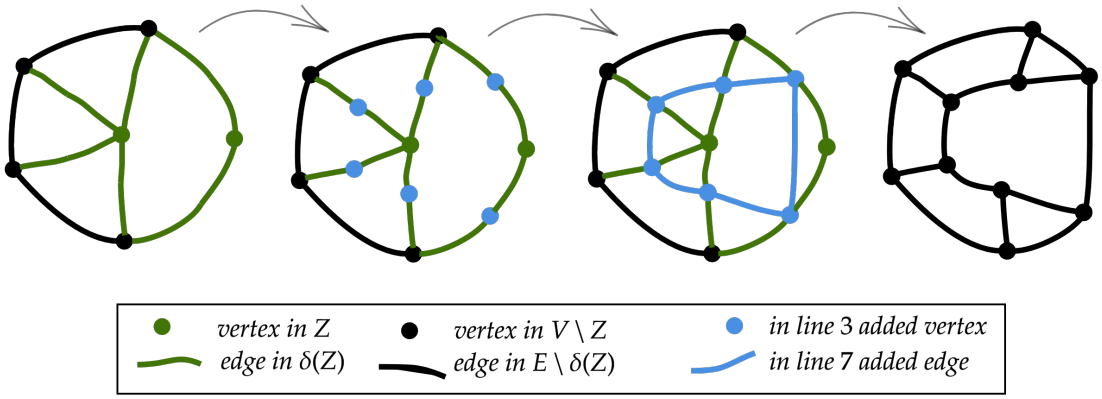


Figure 9: Schematic example of algorithm 2 on a graph with two vertices in Z.

The example above gives us a sensation for the idea of algorithm 2. Still, the condition in line 9 is met at no point. The following example, illustrated by figure 10, shows us where this situation may arise. The graph consists of only one pair of neighbors. If in the next step, these neighbors were connected by only one new edge, the graph would not be 2-connected in the end. If they were connected by two edges, the graph wouldn't be simple. Hence, a new vertex needs to be added, such that the output is a 2-connected graph.

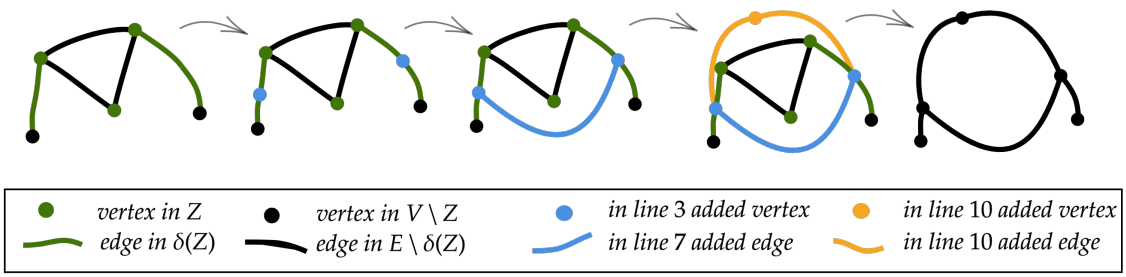


Figure 10: Schematic example of algorithm 2 executing line 10

3.3 An invariant property

Using in the previous chapter described basic cutting scheme, Löhne shows that there exists a property for an ϵ-approximative V-representation $\mathcal{V}$, which is invariant under algorithm 1. Following [Löh20, pp. 15–16] we will introduce a more general notation.

Definition 3.17. *[Löh20] Let $G = (V, E)$ be a plane graph with all its components being 2-connected, C a bounding cycle in G and $f : V \to \mathbb{R}^3$ a function. Then*

1. *$f(C)$ is called a* skew polygon.
2. *Every vertex v in C is identified with the point $f(v)$ and called* vertex *of $f(C)$.*
3. *Every edge $e = uv$ is identified with the line segment between $f(u)$ and $f(v)$ in $\mathbb{R}^3$ and called* edge *of $f(C)$.*

The function f is not required to be injective, i.e. vertices and edges in $f(C)$ are not necessarily distinct.

Definition 3.18 ([Löh20])**.** *Let $d \in \mathbb{R}^3$ be a direction. For a vector $a \in \mathbb{R}^3$, linearly independent of d, the set*

$$M := M(d, a) := \mathbb{R} \cdot d + \mathbb{R}_+ \cdot a$$

defines a half-plane in $\mathbb{R}^3$, whose relative boundary is the line $\mathbb{R} \cdot d$.

Definition 3.19 ([Löh20])**.** *Let $B_+ := \{x \in \mathbb{R}^3 \mid b^T x \geq \beta\}$ be a half-space such that $b^T d > 0$ and $0 \notin B_+$. We say d* runs through *a skew polygon $f(C)$ in B_+ if*

(i) $f(C)$ belongs to B_+

(ii) the number m of edges of $f(C)$ crossing the half plane M is odd.

These definitions allow us to state the invariant property. It requires the existence of a plane graph and a mapping from the vertices of the graph to the set of points in $\mathbb{R}^3$, such that three conditions are satisfied:

Definition 3.20 ([Löh20])**.** *The set $\mathcal{V}$ is called* regular *if there is a plane graph $G = (V,E)$ with all its components being 2-connected and a function $f : V \to \mathcal{V}$ such that*

(A) $|J_{\geq}(f(u)) \cap J_{\geq}(f(v))| \geq 2$ for every $uv \in E$.

(B) For every bounding cycle C in G with vertices $V(C)$ there exists an index κ_C such that

$$\kappa_C \in \bigcap_{v \in V(C)} J_{\geq}(f(v))$$

(C) For almost every direction d in $\mathbb{R}^3$, the number of bounding cycles C in G such that d runs through the skew polygon $f(C)$ in $B_+(C) := \{x \mid A_{\kappa_C} x \geq 1\}$ is odd

As an arbitrary direction could intersect a vertex or a line segment of the embedded graph $f(G)$, Löhne defines by *almost every direction*, that for an arbitrary direction $\bar{d} \in \mathbb{R}^3$ and an arbitrary small $\gamma > 0$ there is a direction d that runs through an odd amount of bounding cycles such that

$$\left\| \frac{d}{||d||} - \frac{\bar{d}}{||\bar{d}||} \right\| < \gamma.$$

The first condition resembles the concept that a vertex u is covered by a point v from section 2. It ensures, that every edge in the graph lies under the mapping outside of at least two inequalities, e.g. it covers an edge of P. The second condition represents a similar argument for the faces of the graph: every bounding cycle in the graph lies under the mapping outside of at least one inequality. And the third condition ensures that conv($\mathcal{V}$) contains the point of origin and P.

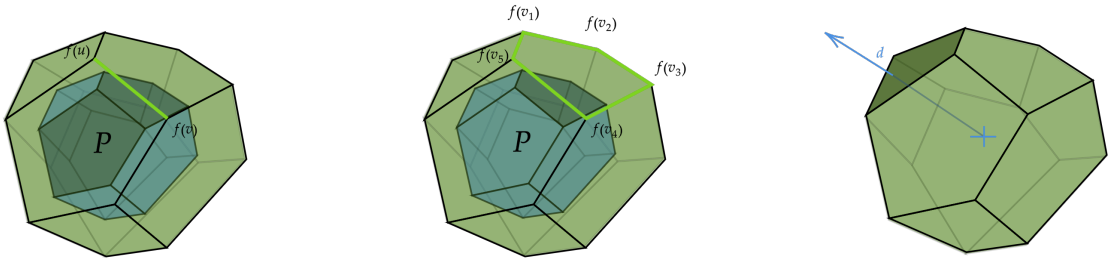


Figure 11: Illustration of the concept *regular*: On the left Property (A): An edge $uv \in E$ under the map f (light green) lies outside of two halfplanes defining P (dark green). In the center property (B): A bounding cycle C in G under the map f (light green) lies outside of one halfplane defining P (dark green). And on the right Property (C): An arbitrary direction d (blue) runs through exactly one skew polygon.

With these notions, we are able to present the results of [Löh20] that allow the iterative application of algorithm 1 for the approximate vertex enumeration.
The following proposition gives us a hint for the first step of the approximate vertex enumeration, which consists in finding an initial ϵ-approximate V-representation $\mathcal{V}$ of P that subsequently allows the repeated application of the cutting scheme.

Proposition 3.21 ([Löh20]). *Let P a 3-dimensional simplex with $0 \in int(P)$. A strong ϵ-approximate V-representation $\mathcal{V}$ of P is regular.*

Theorem 3.22 ([Löh20]). *For $d = 3$, the property of $\mathcal{V}$ being regular is invariant in Algorithm 1.*

Proposition 3.23 ([Löh20]). *Let $\mathcal{V}$ be regular. Then $P \subseteq Q$*

Theorem 3.22 can be seen as the main theorem, as it describes the desired invariance of the property being regular under algorithm 1.
Proposition 3.23 ensures that a regular set $\mathcal{V}$ contains P.

3.4 Vertex assignments

By formulating the approximate vertex enumeration as graph algorithm, we have to clarify which points in the space we assign to vertices in Z and to the newly added vertices v_e in line 3 and w in line 10 of algorithm 2. The information can be extracted from the definition of the function f' in the proof of Theorem 3.22 [Löh20, p. 17].
Given an ϵ-approximative V-representation $\mathcal{V}$ of P, the corresponding graph G and function f and a half-plane $H_{\leq} := \{x \in \mathbb{R}^3 \mid h^T x \leq 1\}$, defined by an $h \in \mathbb{R}^3 \setminus \{0\}$, we want to compute G' and f', which together define an ϵ-approximative V-representation $\mathcal{V}'$ of $P \cap H_{\leq}$.
First, we define the set Z: If there is a vertex $v \in V$ with $f(v) \in H_+$, v is assigned to Z. That is $Z := f(V) \cap H_+$. If Z is empty algorithm 2 will do nothing.
If in line 3 a vertex v_e is added on the edge $e = uv$ to the graph G, we assign a new point in $\mathbb{R}^3$ to v_e. Without loss of generality, we assume $u \in Z$, since e is in $\delta(Z)$ per definition of algorithm 2: If $f(v) \in H_-$, we compute a new point $w \in \text{conv}\{u, v\} \cap H_0$ and set $f(v_e) := w$. If $f(v) \in H_0$ we do not compute a new point but set $f(v_e) := f(v)$.
Third, if in line 10 is a vertex w and two edges $v_e w$, $w v_f$ added to the graph G we set $f(w) := f(v)$.

In the last cases, we do not compute new points as we use already existing points, which can be seen as an advantage to algorithm 1 as we need fewer computations. On the other side, the vertices of a face do not necessarily lie in a plane (see Figure 12). Thus, the final graph G' under the map f' does not necessarily yield a polytope, as we will see in figure 14.

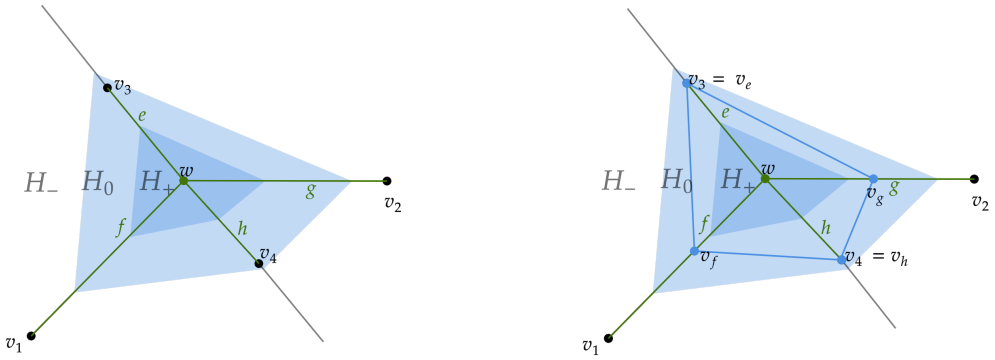


Figure 12: Schematic example of the vertex assignment: four vertices v_e, v_f, v_g and v_h are added. The vertices v_1 and v_2 lie (under the map f) in H_- and thus v_f and v_g are assigned to new points on the line $\text{conv}\{w, v_1\} \cap H_0$ and $\text{conv}\{w, v_2\} \cap H_0$, respectively. In contrast v_3 and v_4 lie in H_0 and hence v_e and v_h are assigned to v_3 in H_4, respectively.

Finally, we can formulate the 'embedded' form of the basic graph cutting scheme:

Algorithm 3: Basic graph cutting scheme with vertex assignment

input : regular ϵ-approximate V-representation $\mathcal{V}$ of $P = P(A)$, given by a plane graph (V, E) and function $f : V \to \mathcal{V}$;
half-space $H_{\leq} := \{x \mid h^T x \leq 1\}$, given by $h \in \mathbb{R}^3$

output : regular ϵ-approximate V-representation $\mathcal{V}'$ of $P \cap H_{\leq}$, given by a plane graph $G' = (V', E')$ and function $f' : V' \to \mathcal{V}'$

begin
  $Z \leftarrow \{v \in V \mid f(v) \in H_+\}$
  $\delta(Z) \leftarrow \{uv \in E \mid f(u) \in \mathcal{V}_+,\ f(v) \in \mathbb{R}^3 \setminus H_+\}$
  for $e = uv \in \delta(Z)$ **do**
    add a new vertex v_e to V, draw it on the arc and replace $e = uv \in E$ by uv_e and $v_e v$
    if $f(v) \in H_-$ **then**
      compute $w \in \mathrm{conv}\{u, v\} \cap H_0$
      $f(v_e) \leftarrow w$
    else
      $f(v_e) \leftarrow f(v)$
    end
  end
  for $e, f \in \delta(Z)$ **do**
    if e, f *are neighbors* **and** $v_e v_f \notin E$ **then**
      add $v_e v_f$ to E. Draw the corresponding arc inside a common face
    end
    if e, f *are neighbors* **and** e, f *have no other neighbors* **then**
      add a new vertex w to V and two edges $v_e w$, $w v_f$ to E
      $f(w) \leftarrow f(v_e)$
    end
  end
  $V' \leftarrow V \setminus Z$
  $E' \leftarrow E \setminus \{uv \in E \mid u \in Z \vee v \in Z\}$
  $f' \leftarrow f$
end

Lemma 3.24. *Let $\mathcal{V}$ be a regular ϵ-approximate V-representation of $P = P(A)$ given by a plane graph (V, E) and function $f : V \to \mathcal{V}$ and $H_{\leq} := \{x \mid h^T x \leq 1\}$ a half-space. The result of algorithm 3 is a regular ϵ-approximate V-representation $\mathcal{V}'$ of $P \cap H_{\leq}$ given by a plane graph $G' = (V', E')$ and function $f' : V' \to \mathcal{V}'$*

Proof. The function f' is well defined, as for every newly added vertex v_e either a new point $w \in \mathbb{R}^3$ or an existing point $f(v)$ for a $v \in V$ is assigned. According to Theorem 3.22, $\mathcal{V}'$ is regular and according to Proposition 3.23, P is a subset of $Q' = \text{conv}(\mathcal{V}')$. As a vertex v with $f(v) \in V_+$ will be removed by the algorithm 3, it is clear that $Q' \subseteq (1 + \epsilon)P$. Hence $\mathcal{V}'$ is an ϵ-approximate V-representation. □

3.5 Approximate graph based double description method

With the results of the previous section, we can compute iteratively an ϵ-approximate V-representation $\mathcal{V}$ of a given H-polytope:

Algorithm 4: Approximate graph algorithm

input : 3-dimensional H-Polytope $P(A) := \{x \in \mathbb{R}^3 | Ax \leq e\}$, given by $A \in \mathbb{R}^{m \times 3}$, where for simplicity we assume that $P(A_{[4]})$ is bounded; tolerance $\epsilon \geq 0$

output : ϵ-approximate V-representation $\mathcal{V}$ of $P = P(A) = P(A_{[m]})$, given by a plane Graph $G = (V, E)$; and an embedding $f : V \to \mathcal{V} \subseteq \mathbb{R}^3$.

```
1 begin
2     Compute a strong ε-approximate V-representation V with |V| = 4 of the simplex
        P(A_[4]) and assign V to the vertices of a tetrahedral graph G by the function f
3     for i ← 5 to m do
4         h ← A_i^T
5         update G by Algorithm 3 applied to G and h
6     end
7 end
```

By the following Lemma we recall results from [LW16] that prove the correctness of Correctness of algorithm 4.

Lemma 3.25. *Let $P = P(A)$ be a 3-dimensional H-Polytope $P = P(A) := \{x \in \mathbb{R}^3 | Ax \leq e\}$, given by $A \in \mathbb{R}^{m \times 3}$. The result of algorithm 4 is an ϵ-approximate V-representation $\mathcal{V}$ of P, given by a planar Graph $G = (V, E)$, and a function $f : V \to \mathcal{V} \subseteq \mathbb{R}^3$.*

Proof. By Proposition 3.21, a strong ϵ-approximate V-representation $\mathcal{V}$ with $|\mathcal{V}| = 4$ of the simplex $P(A_{[4]})$ is regular. By Theorem 3.22, the property of being regular is invariant under agorithm 2 and by Lemma 3.24, the result is a ϵ-approximate V-representation $\mathcal{V}$ of P . □

In line 2 it is not clarified how a strong ϵ-approximate V-representation $\mathcal{V}$ of P can be computed. A possibility is to find three linear independent rows in A (which are by assumption the first three rows A_1, A_2, A_3). By finding a fourth row which is independent of each of the first three and by assumption A_3, we obtain with $P(A_{[4]})$ a 3-simplex, which contains P. Such a fourth inequality can also be created by adding up the negative of each of the three linear independent rows to a row $h = -(A_1 + A_2 + A_3)$ and "moving" the corresponding plane $h^T x = b$ so that P is lying completely in the half-plane. This means that the linear program $max_x : h^T x$ s.t. $Ax \leq e$ needs to be solved.
The four vertices of the simplex are given by the intersection of three of the four half-planes. By multiplying the vertices by a factor $1 + \frac{\epsilon}{2}$, we inflate the simplex and get a strong ϵ-approximate V-representation $\mathcal{V}$ of P, as every vertex of the simplex now lies outside of two half-planes. According to Lemma 3.21 the simplex is regular.

4 Implementation

4.1 Introduction

When it comes to implementing a graph algorithm, the ease of the implementation and the efficiency of the algorithm depend on the chosen data structure. For the implementation of algorithm 4, a data structure is required that stores both the topological information (such as adjacent vertices, in the sense that they are connected by an edge) of the graph and the geometrical information of its embedding in the $\mathbb{R}^3$ (e.g. the coordinates of the vertices).
There are many possible representations for graphs. The most popular data structure for simple undirected graphs are adjacency lists, where each vertex stores a linked list with its adjacent vertices and adjacency matrices, in which the rows represent source vertices and the columns represent destination vertices and connected pairs are marked by a 1-entry (0 otherwise) [Din04, pp. 4–1].
Although both adjacency data structures allow some very efficient graph operations as adding a vertex or an edge in constant time for the adjacency list and adding or removing edges in constant time for the adjacency matrix, they both have disadvantages for our purpose: Operations as removing a vertex or an edge for the adjacency list or adding and removing a vertex for the adjacency matrix have a running time linear or even quadratic to the whole amount of vertices and edges of the graph, respectively. For example, if we want to remove a vertex (and its adjacent edges) in a graph represented by an adjacency list, in the worst case we have to query all vertices and traverse all edges. Or, if we want to add a new vertex to the graph represented by a $V \times V$-dimensional adjacency matrix, the matrix needs to be copied to a $(V+1) \times (V+1)$-dimensional matrix, which needs $(V+1)^2$ operations.
Thus, one disadvantage is that data from the entire graph is required, even if we are only interested in local manipulations (as removing an edge) or queries (finding all adjacent edges for one vertex). Moreover, for the manipulation of planar graphs, often needed information about the faces is not available in adjacency lists or matrices.
A simple solution would be a face-to-edge-list, which stores for every face all edges enclosing the face. However, such a list alone contains very little information about the structure of the graph, which makes traversal and graph modification without another data structure slow.
Another obvious possibility is to store all adjacencies for every edge, face and vertex. In terms of object-oriented programming, we would have three different types of objects:

A vertex object, storing a list of all adjacent faces (that is, all faces, which have the vertex on its boundary) and a list of all adjacent edges (that is, all edges, which have the vertex as its end).

```
class Vertex:
    def __init__(self):
        self.adjacent_faces = []
        self.adjacent_edges = []
```

An edge object, storing two references each to the adjacent faces and the adjacent vertices along one arbitrary orientation regarding the face.

```
class Edge:
    def __init__(self):
        self.origin_vertex = None
        self.target_vertex = None
        self.left_face = None
        self.right_face = None
```

A face object, storing a list of all adjacent vertices and edges

```
class Face:
    def __init__(self):
        self.adjacent_vertices = []
        self.adjacent_edges = []
```

Now, the data structure allows querying comfortably all adjacencies and manipulating the planar graph in an intuitive way. But such a data structure that stores all adjacencies also has several important drawbacks, as stated in [Gha08, p. 246]:

1. As the number of faces, vertices, and edges of the graph varies during the runtime of the algorithm, the storage requirements for the vertex objects and face objects, which store all adjacent faces, edges, and vertices are not constant. By removing and adding objects to the lists of those objects, the system may need to constantly allocate and reallocate memory. That could lead to a significant memory segmentation.

2. It is not possible to query the vertices around a given face in clockwise or counter-clockwise order in constant time, as there is no information about the order of the vertices. The same counts for querying the faces around a given vertex. Since our graph can both have faces and vertices with numerous adjacent vertices and faces, respectively, manipulating such highly connected objects could have a serious impact on the running time.

3. The orientation of an edge does agree with one of its adjacent faces, but not with

both. For example, if we assume the orientation to be counterclockwise regarding the vertices in a face. Such inconsistencies can easily lead to difficulties when implementing the algorithm.

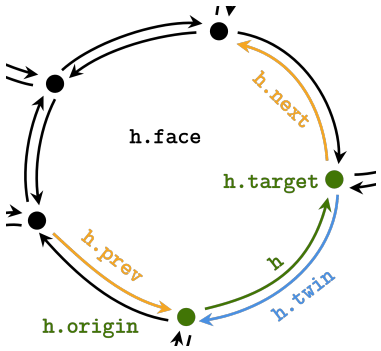
4.2 The halfedge data structure

The *halfedge data structure* (HEDS) is aside from the winged-edge data structure and the quad-edge data structure a well-known data structure for planar graphs [Din04, pp. 17–2]. The main feature of this edge-based data structure is that each edge is divided up into two directed twin halfedges, each pointing to the opposite vertex. By paying a small price in storage requirements, according to [Gha08, p. 246] it solves the three described problems:

1. Each object has constant space requirements as it contains a set amount of references.
2. For vertices and faces, lists of adjacencies can be reconstructed in time proportional to their size.
3. By splitting an edge into two halfedges, it is possible to assign a halfedge to each of the adjacent faces and thus give them a consistent orientation.

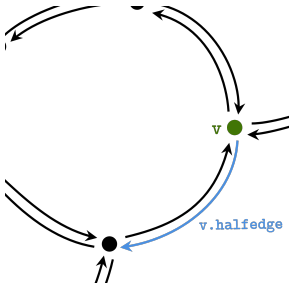
This edge-based data structure is also known as Doubly Connected Edge List (DCEL) [Ber+00, p. 30] even if the first DCEL described by Muller and Preparata in [MP78] is different from what we consider as DCEL or HEDS nowadays [GH13, p. 50]. Further similar data structures for planar graphs like the quad-edge data structure and the winged-edge data structure are introduced and compared in [Ket99]. The HEDS is applied in geometric computation and computer graphics where it is used to represent polytopes, polygon meshes or other orientable, two-dimensional surfaces, as for example in the Computational Geometry Algorithms Library (CGAL) [Ket21] or the OpenMesh Library [Bot+02].
The HEDS stores a list of halfedges, vertices, and faces, which are unordered, but interconnected by references of its objects [GH13, p. 53]:
Halfedges are the main ingredients of the data structure and store each five different references. Two opposite directed halfedges represent an edge and are connected by a reference to its `twin`-halfedge. Further, each halfedge object `h` stores a reference to its succeeding (`h.next`) and preceding (`h.prev`) halfedge along a face. This makes it easy to traverse all edges of a face. The vertices, which are connected by the halfedge `h` are given by the references `h.origin` and `h.target`. Finally, the reference `h.face` marks the adjacent face.

```python
class HalfEdge:
    def __init__(self):
        self.origin = None
        self.target = None
        self.twin = None
        self.face = None
        self.next = None
        self.prev = None
```



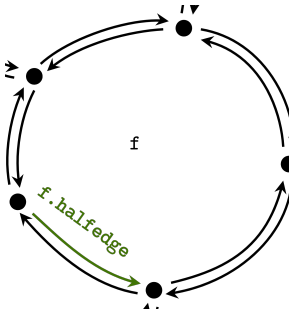
Vertices are in the HEDS light objects as most of the connectivity information is already contained in the halfedges. Each vertex object `v` stores a reference `v.halfedge` to an arbitrary halfedge with the vertex as the origin. Furthermore, we store in `v.coordinates` the corresponding point in the $\mathbb{R}^3$.

```python
class Vertex:
    def __init__(self):
        self.coordinates = None
        self.halfedge = None
```



A **face** of a planar graph is defined by its bounding cycle. However, a face object `f` in the HEDS stores only a reference `f.halfedge` to an arbitrary adjacent halfedge instead of all bordering vertices or halfedge objects.

```python
class Face:
    def __init__(self):
        self.halfedge = None
```

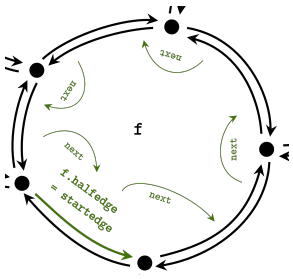


4.3 Querying in the halfedge data structure

Even if this large amount of references is not required to store a unique representation of a planar graph, it allows time-efficient traversal, which is an important factor for a comfortable and straightforward manipulation of the data structure. We will take a closer look at the querying abilities of the HEDS:

Given a face object `f` it is possible to traverse the list of vertices or the edges of the face in time proportional to their cardinality: We choose the arbitrary halfedge, which is linked with the `f` by the reference `f.halfedge` as startedge and call the successive halfedges as long as we don't reach the start edge again.

```
startedge = f.halfedge
halfedge = startedge
while True:
    do_something(halfedge.target)
    halfedge = halfedge.next
    if halfedge == startedge:
        break
```

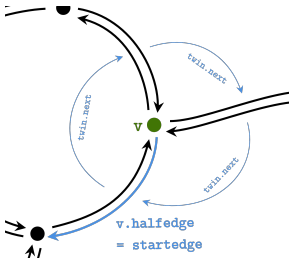


And similarly, by choosing `do_something(halfedge.twin.face)`, it is possible to traverse the faces adjacent to `f`.

Given a vertex `v`,we can traverse the edges outgoing from or incoming to a vertex as well as the adjacent faces in time proportional to their cardinality. Therefore, we start with the arbitrary halfedge `v.halfedge` and use the reference *twin* to get to the adjacent faces until we finally reach `v.halfedge` again:

```
startedge = v.halfedge
halfedge = startedge
while True:
    do_something(halfedge.face)
    halfedge = halfedge.twin.next
    if halfedge == startedge:
        break

```



Since the runtime of these queries depends only on the local complexity of the chosen face or vertex, the HEDS is a convenient data structure for manipulating highly connected planar graphs, i.e. planar graphs with many edges per vertex on average.

4.4 Euler operators

To implement the algorithm functions are needed, which make both adding and deleting edges and vertices to and from the HEDS respectively possible, without harming the planar properties of the graph. The halfedge data structure is not only useful because it allows adjacency queries in time proportional to their cardinality. Another advantage is that it enables to implement a class of operators that locally modify the planar graph which can be executed in time proportional to their local complexity [Gha08].
These functions rely on a fundamental property about 3-polytopes (i.e. polyhedra), which is named after Leonhard Euler. He observed that in any convex polyhedra the sum of the total number of vertices $|V|$ and faces $|F|$ surpasses the total number of edges $|E|$ by 2. By recalling Theorem 3.11 (Steinitz's Theorem), this property can also be translated into graph theory:

Theorem 4.1 (Euler's Theorem)**.** *Let G be a connected planar graph and $|V|$, $|E|$ and $|F|$ its number of vertices, edges and faces, respectively. Then*

$$|V| - |E| + |F| = 2$$

.

Proof. We begin with the trivial graph, which only has one vertex and one (outer) face. Thus it holds $1 - 0 + 1 = 2$. We observe that G can be iteratively created by adding a vertex and an edge or connecting two existing vertices by a new edge. In the first case, the value of $|V|$ goes up by one, but so does the value of $|E|$. In the second case, the value of $|E|$ goes up by 1 but so does the value of $|F|$, since we split the existing face into two faces. Thus, after obtaining G from the trivial graph by applying these two operations theorem 4.1 still holds. □

Euler operators are functions that operate on planar graphs by adding or deleting vertices, edges and faces, so that Euler's Theorem remains valid for the planar graph. However, it is not said that Euler operators satisfy the equation during their execution [Gha08, p. 249]. Euler operators were introduced by Mantylla [Man84] in 1984. He further proved that Euler operators form a complete set of modelling primitives, that is, every topologically valid polyhedron can be constructed from an initial polyhedron by a finite sequence of Euler operations [Man84, p. 55].
There are two groups of Euler operators apart from few exceptions: The "make" and the "kill" group, which add or remove objects from or to the graph, respectively.
An Euler operator is written as `mxy` or `kxy`,dependent on which group it belongs to. For the objects the obvious acronyms `v` for vertex, `e` for edge and `f` for face are used. The operator `kev` for example, stands for kill-edge-vertex and removes an edge and a vertex.

operation	description
kev	kill-edge-vertex
kef	kill-edge-face
mev	make-edge-vertex
mef	make-edge-face
split_edge	add edge by inserting vertex on existing edge
join_edge	delete edge by merging the end vertices

Table 1: A partial list of Euler operators.

For the graph algorithm, we need to create new vertices on existing edges (`split_edge`), connect two existing vertices inside a face (`mef`) and add a new vertex inside an existing vertex and connect it to an existing vertex (`mev`). Further, we need to remove edges that are adjacent to two different faces (`kef`) and remove vertices with only one adjacent edge (`kev`).
Euler operators also demonstrate the uncomplicated usage of the HEDS: Only very few

parameters are required for Euler operators to be unambiguously defined. The operator `mev`, for example, requires one halfedge as input. Its target vertex will be joined with a newly added vertex. The face in which the new vertex and halfedges are inserted is defined by the given halfedge.

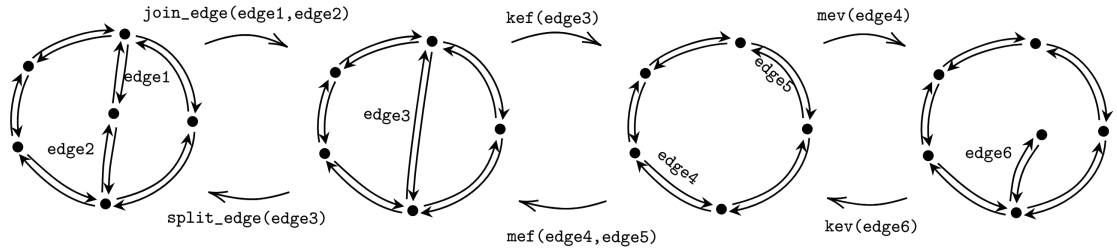


Figure 13: A subset of Euler operators that are required for algorithm 2

Assuming that Euler's equation $|V| - |E| + |F| = 2$ is valid for a given planar graph, we can easily verify that Euler's equation remains valid for the manipulated planar graph:

operation	Euler equation after operation completes
mev	$(\|V\| + 1) - (\|E\| + 1) + \|F\| = 2$
kev	$(\|V\| - 1) - (\|E\| - 1) + \|F\| = 2$
mef	$\|V\| - (\|E\| + 1) + (\|F\| + 1) = 2$
kef	$\|V\| - (\|E\| - 1) + (\|F\| - 1) = 2$
split_edge	$(\|V\| + 1) - (\|E\| + 1) + \|F\| = 2$

We take a closer look at the runtime of some of these operations:
The simplest operator is `kev` (kill_edge_vertex): Two halfedges and one vertex need to be removed from the HEDS and only the adjacent halfedges and the adjacent vertices of those halfedges need to be reassigned. Clearly, these operations have a constant runtime.

```
def kill_edge_vertex(self,edge1):
    #reassing prev and next halfedge
    edge1.prev.next = edge1.twin.next
    edge1.twin.next.prev = edge1.prev
    #reassign halfedge of the adjacent vertex:
    edge1.origin.halfedge = edge1.twin.next
    #reassign halfedge of the adjacent face:
    edge1.face.halfedge = edge1.twin.next
    #remove halfedges and the vertex:
    self.vertices.remove(edge1.target)
    self.halfedges.remove(edge1)
    self.halfedges.remove(edge1.twin)
```

By removing an edge between two vertices via `kef` (kill_edge_face) more assignments are required: Let `edge1`be the argument of the function. We need to reassign references of the adjacent halfedges and vertices of `edge1` and `edge1.twin`. Furthermore, we need to reassign `face`-reference for all edges adjacent to the face we want to remove. Finally, we remove the halfedges and the face. Thus the runtime is proportional to the cardinality of the face of `edge1.twin`.

```
def kill_edge_face(self,edge1):
    face = edge1.face #face to keep
    face_to_remove = edge1.twin.face
    #assign face to the edges of the face to remove:
    edge = edge1.twin
    startedge = edge
    while(True):
        edge.face = face
        edge = edge.next
        if(edge == startedge):
            break
    #reassign prev and next for the adjacent edges
    edge1.prev.next = edge1.twin.next
    edge1.twin.next.prev = edge1.prev
    edge1.next.prev = edge1.twin.prev
    edge1.twin.prev.next = edge1.next
    #reassign halfedge references of the adjacent vertices
    edge1.origin.halfedge = edge2.next
    edge2.origin.halfedge = edge1.next
    edge1.face.halfedge = edge1.prev
    edge2.face.halfedge = edge2.next
    #remove halfedges and the face
    self.faces.remove(face_to_remove)
    self.halfedges.remove(edge1)
    self.halfedges.remove(edge1.twin)
```

The `mev` (make_edge_vertex) function has a constant runtime: Since there is only a vertex and two halfedges to be added, it is sufficient to make some local reassignments without querying the whole face.

```
def make_edge_vertex(self,edge1):
    '''
    adds a new vertex to edge1.face and joins it with edge1.target
    '''
    face = edge1.face
    vertexnumber = len(self.vertices)
    vertex_new = Vertex(vertexnumber+1)
    h1 = HalfEdge(edge1.target, vertex_new)
    h2 = HalfEdge(vertex_new, edge1.target)
```

```
    vertex_new.halfedge = h2
    h1.twin = h2
    h2.twin = h1
    h1.face = h2.face = face
    h1.next = h2
    h2.next = edge1.next
    h1.prev = edge1
    h2.prev = h1
    edge1.next = h1
    h2.next.prev = h2
    self.vertices.append(vertex_new)
    self.halfedges.extend([h1,h2])
```

For implementation details of further Euler operators, we refer to the full code in the appendix.

4.5 Space and time complexity

For a face object `f`, let $V_{\texttt{f}}$ be the number of the vertices adjacent to `f` and for a vertex object `v`, let $E_{\texttt{v}}$ be the number of the edges adjacent to `v`. The following table sums up the time complexity of the Euler operators and some queries, which are used in the implementation of algorithm 4.

operator	description	time complexity
`kev(e)`	kill-edge-vertex	$O(1)$
`kef(e)`	kill-edge-face	$O(V_{\texttt{e.face}})$
`mev(e)`	make-edge-vertex	$O(1)$
`mef(e_1,e_2)`	make-edge-face	$O(V_{\texttt{e1.face}})$
`split_edge(e)`	add edge by inserting vertex on existing edge	$O(1)$
`face_traversal(f)`	traverse around a face f	$O(V_{\texttt{f}})$
`vertex_traversal(v)`	traverse around a vertex v	$O(E_{\texttt{v}})$

For a planar graph with E edges the HEDS stores solely for the halfedges $12E$ references. Additionally, for V vertices and F faces $2V$ and F references are stored, respectively. The storage size of an halfedge `h` can be reduced by discarding one of the references `h.target` or `h.origin`. The downside would be that every time a discarded reference is needed, two indirect memory accesses of the form `h.next.origin` or `h.prev.target` are required instead. As we aim for a smooth and comprehensible implementation we will stick with both references.
For merely storing the planar graph and the coordinates of its vertices other data structures are more convenient. The Object File Format (OFF), for instance, is a common format for storing polyhedral objects. We will introduce it in the following chapter.

5 Results and comparison of the implementation

As mentioned in chapter 1, the resulting ϵ-approximate V-representation has not necessarily the same combinatorics as the polytope P, which can be seen as an advantage, as less information might be obtained with a less computational expense. Now, we want to take a closer look at the computational effort of algorithm 4 and the topological properties of its output. We use the HEDS from the previous chapter for implementing algorithm 4 in Python. Further, we want to compare this implementation to an implementation of the original approximate double description method developed in [Löh20, p. 19], which iteratively uses the in chapter 1 described algorithm 1 to compute $\mathcal{V}$. The pseudo-code of this algorithm can be found in the appendix) as well as the Python code of the implementation of the HEDS and the algorithms.

5.1 Topological properties

The output of the algorithm is given by a graph G and its embedding $f : V \to \mathbb{R}^3$. In the implementation, both the topological and the geometrical information are stored in the resulting HEDS.
As input polytopes we use combinations of the basic polytopes, denoted in [HJZ99] as follows:
The **cube**, given by

$$\begin{aligned} C_3 &:= \operatorname{conv}\{\lambda_1 e_1 + \lambda_2 e_2 + \lambda_3 e_3 \mid \lambda_1, \lambda_2, \lambda_3 \in \{+1, -1\}\} \\ &= \{x \in \mathbb{R}^3 \mid -1 \leq x_i \leq 1 \text{ for } i = 1, 2, 3\}, \end{aligned}$$

where e_1, e_2, e_3 are the coordinate unit vectors in $\mathbb{R}^3$.
The **cross-polytope** (in our case called octahedron), given by

$$C_3^{\Delta} := \operatorname{conv}\{\pm e_1, \pm e_2, \pm e_3\} = \{x \in \mathbb{R}^3 \mid \sum_{i=1}^{3} |x_i| \leq 1\}.$$

The **simplex**, given by

$$T_3 := \operatorname{conv}\{e_1, e_2, e_3 \; \frac{1 - \sqrt{3+1}}{3}(e_1 + e_2 + e_3)\}$$

We scale a polytope P with a scalar multiple λ such that $\lambda P := \{\lambda x \mid x \in P\}$ and combine two polytopes P and P' with the **Minkowski sum**, which is given as

$$P + P' = \{p + p' \mid p \in P, p' \in P'\}.$$

For these operations as well as for computing the H-representation of such a polytope we use functions from the package **bensolvetools**, which is based on theoretical results from

[LW16] and [CLW18]. The package also defines `bensolvehedron(3,m)` as the image of an $(2m+1)^3$-dimensional hypercube under a $(3 \times (2m+1))$-matrix, whose columns consist of all $(2m+1)^3$ possible arrangements of the numbers $\{m, m-1m, .., 1, 0, -1, ..-m+1, -m\}$.

To visualize the output, we convert the resulting HEDS to an OFF-file (**O**bject **F**ile **F**ormat), which uses an indexed face set to describe the object. At the top of the file, the number of vertices and faces are defined, which have to match the ensuing description of the faces. The following lines consist of the 3D-coordinates of the vertices, and, subsequently of the number of vertices and the corresponding indices for every face.
All this information can be easily extracted from the DCEL by querying the vertices for every face and eventually reassigning its indices. After converting the DCEL to an OFF-file, we use JavaView (`http://www.javaview.de/`) to visualize the output.

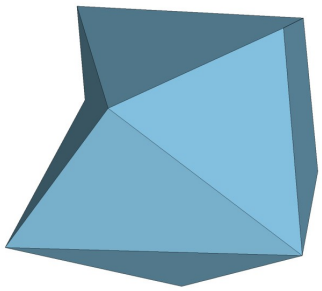


Figure 14: Visualized output of the dual polytope of bensolvehedron(3,2) under algorithm 4 for $\epsilon = 5$

These visualizations via OFF-file also allow us to see the non-convex properties of the embedded graph, as we can see in Figure 14: Due to the fact that the algorithm does not compute new but assign already existing coordinates to new vertices, the faces do not necessarily lie in a plane.

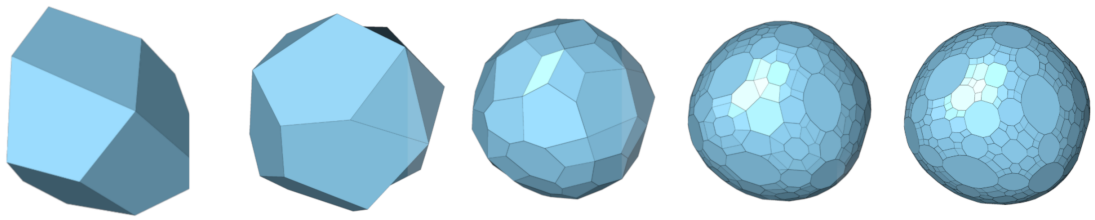


Figure 15: OFF-file output of Q for $\epsilon \in \{5, 1, \frac{1}{10}, \frac{1}{100}, \frac{1}{10000}\}$ in the corresponding order

Figure 15 illustrates the impact of the choice of ϵ on the topology of the output. It seems that with a decreasing ϵ-value, the number of vertices, edges, and faces increases until the shape of the input polytope is reached. The data for another test instance, the Minkowski sum of a bensolvehedron(3,1) and basic polytopes that consists of 170 inequalities and has 120 vertices, supports the assumption (see figure 16): for almost every ϵ the graph algorithm computes fewer vertices than the original approximate vertex enumeration algorithm and even fewer vertices than the input polytope. Only for $\epsilon = 0$ the number of vertices exceeds the vertices of the input polytope multiple times.

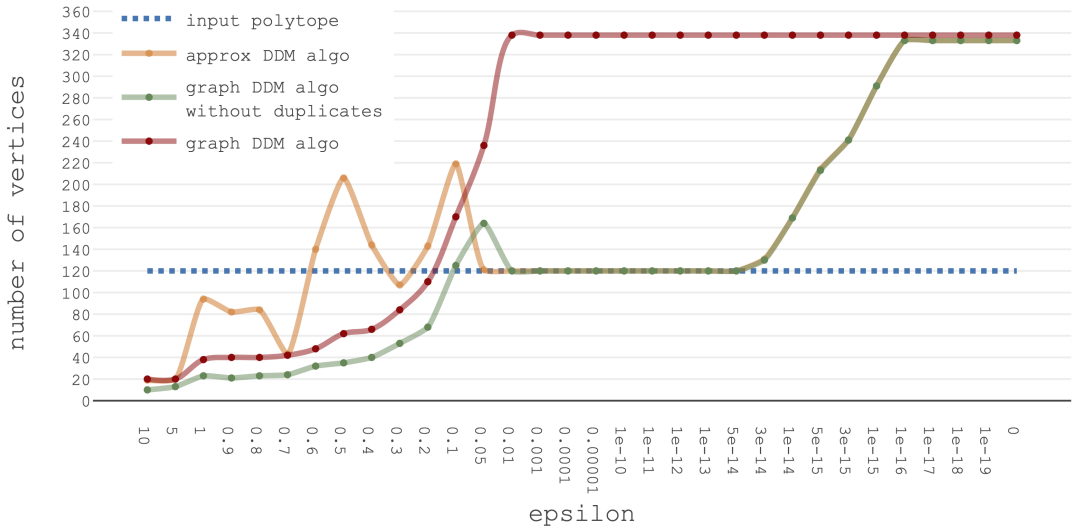


Figure 16: Given a polytope with 170 inequalities and 120 vertices, the diagram shows the number of vertices of the output from algorithm 4 (red), and the approximate DDM algorithm (yellow). Furthermore, the output 4 after the removal of vertices with the same coordinates. Table 2 with the exact data can be found in the appendix.

5.2 Runtime

For determining the runtime, we use random polytopes with n edges. Such a polytope is created by projecting n uniformly distributed points in $[0,1)^3$ to the 2-sphere S^2. The projected points represent the edges of the random polytope. By using a Python binding of Komei Fukuda's C library *cddlib* [Fuk18] we calculate its H-representation. This H-representation is used as an input polytope for the graph algorithm. As the runtime can vary depending on the choice of ϵ we measure the sum of the runtime for different ϵ-values and display depending on the amount vertices of the random polytope. For comparative purposes, we also measure the runtime of the vertex enumeration function in the cddlib that computes an exact representation of the H-polytope. Figure 18 shows clearly, that the runtime of the graph algorithm is not comparable to the runtime of Fukuda's C implementation of the DDM.

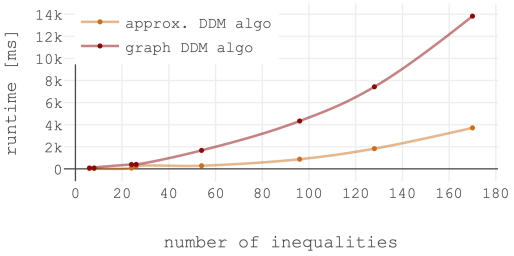


Figure 17: Sum of the runtime of both implementations for the ϵ values $10, 5, 1, 1/10, 1/100, 1/10000$ and 0

Comparing the runtime to our implementation of Löhnes approximate DDM, we use a different approach and use combinations of the basic polytopes as input polytopes. The

amount of vertices of these polytopes ranges from 8 to 170. The runtime of both implementations is comparable, though we can see in figure 17 that the runtime increases significantly faster for the implementation of the graph algorithm.

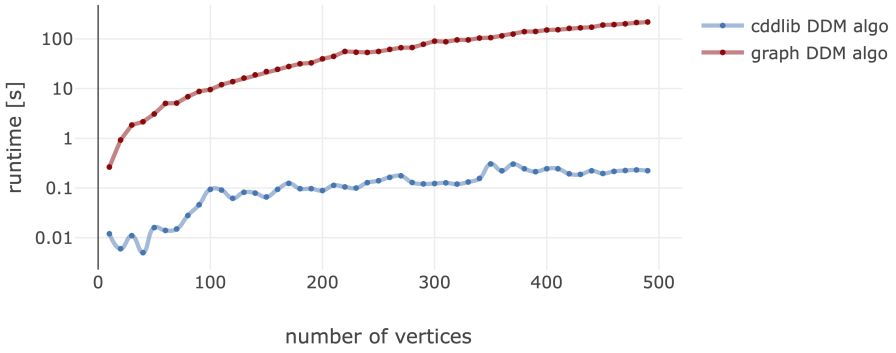


Figure 18: Sum of the runtime of the implementation of the graph algorithm and Fukuda's exact DDM algorithm for the ϵ values $10, 5, 1, 1/10, 1/100, 1/10000$ and 0

6 Conclusion and open questions

Löhne provides a graph-theoretical algorithm for the approximate vertex enumeration, whose implementation works in practice. The HEDS is a convenient data structure for planar graphs. Still, the algorithm can not compete with the approximate DDM from [Löh20] in terms of efficiency. However, in comparison with Löhne's approximate DDM the algorithm produces smaller V-polytopes in the sense that they have fewer vertices. The question of how we can implement the algorithm under the use of the HEDS more efficiently remains open and provides space for further investigations.

Acknowledgements

My wholehearted thanks go to Professor Andreas Löhne for his weekly guidance, valuable inputs, suggestions, and continuous support. Further, I thank Dr. rer. nat Benjamin Weißing for his supervision and feedback throughout the seminar. Finally, I want to express my gratitude to my family and friends not only for their corrections and comments but mainly for their unfailing support and continuous encouragement.

References

[Ber+00] de Berg et al. *Computational Geometry: Algorithms and Applications*. Springer, 2000. ISBN: 9783540656203. URL: https://books.google.com.bn/books?id=C8zaAWuOIOcC.

[Bot+02] M. Botsch et al. „OpenMesh: A Generic and Efficient Polygon Mesh Data Structure“. In: 2002.

[CLW18] Daniel Ciripoi, Andreas Löhne, and Benjamin Weißing. „A vector linear programming approach for certain global optimization problems“. In: *Journal of Global Optimization* 72.2 (Mar. 2018), pp. 347–372. ISSN: 1573-2916. DOI: 10.1007/s10898-018-0627-0. URL: http://dx.doi.org/10.1007/s10898-018-0627-0.

[Die06] Reinhard Diestel. *Graph theory*. 3rd. Graduate Texts in Mathematics. Springer, 2006.

[Din04] Sartaj Sahni (editors) Dinesh P. Mehta. *Handbook of data structures and applications*. 1st ed. Chapman & Hall/CRC computer and information science series. Chapman & Hall/CRC, 2004.

[FP96] Komei Fukuda and Alain Prodon. „Double description method revisited“. In: *Combinatorics and Computer Science*. Ed. by Michel Deza, Reinhardt Euler, and Ioannis Manoussakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 91–111. ISBN: 978-3-540-70627-4.

[Fuk18] Komei Fukuda. *cddlib*. July 12, 2018. URL: https://people.inf.ethz.ch/fukudak/cdd_home/.

[GH13] Bernd Gärtner and Michael Hofmann. *Computational Geometry Lecture Notes*. Jan. 2013. URL: https://www.ti.inf.ethz.ch/ew/courses/CG12/lecture/Chapter%5C%205.pdf.

[Gha08] Sherif Ghali. *Introduction to geometric computing*. 1st ed. Springer-Verlag London, 2008. ISBN: 1848001142,9781848001145.

[HJZ99] Martin Henk, Richter-Gebert Jürgen, and Günter Ziegler. „Basic Properties Of Convex Polytopes“. In: *Handbook of Discrete and Computational Geometry* (Mar. 1999). DOI: 10.1201/9781420035315.pt2.

[Ket+04] Lutz Kettner et al. „Classroom Examples of Robustness Problems in Geometric Computations“. In: vol. 40. Oct. 2004, pp. 702–713. ISBN: 978-3-540-23025-0. DOI: 10.1007/978-3-540-30140-0_62.

[Ket21] Lutz Kettner. „Halfedge Data Structures“. In: *CGAL User and Reference Manual*. 5.3. CGAL Editorial Board, 2021. URL: https://doc.cgal.org/5.3/Manual/packages.html#PkgHalfedgeDS.

[Ket99] Lutz Kettner. „Using generic programming for designing a data structure for polyhedral surfaces“. In: *Computational Geometry* 13.1 (1999), pp. 65–90. ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(99)00007-3. URL: https://www.sciencedirect.com/science/article/pii/S0925772199000073.

[Löh20] Andreas Löhne. *Approximate Vertex Enumeration*. 2020. arXiv: 2007.06325 [math.OC].

[LW16] Andreas Löhne and Benjamin Weißing. „Equivalence between polyhedral projection, multiple objective linear programming and vector linear programming“. In: *Mathematical Methods of Operations Research* 84.2 (Oct. 2016), pp. 411–426. ISSN: 1432-5217. DOI: 10.1007/s00186-016-0554-0. URL: https://doi.org/10.1007/s00186-016-0554-0.

[Man84] Martti Mantyla. „A note on the modeling space of Euler operators“. In: *Computer Vision, Graphics, and Image Processing* 26.1 (1984), pp. 45–60. ISSN: 0734-189X. DOI: https://doi.org/10.1016/0734-189X(84)90129-4. URL: https://www.sciencedirect.com/science/article/pii/0734189X84901294.

[MP78] D.E. Muller and F.P. Preparata. „Finding the intersection of two convex polyhedra“. In: *Theoretical Computer Science* 7.2 (1978), pp. 217–236. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(78)90051-8. URL: https://www.sciencedirect.com/science/article/pii/0304397578900518.

[Sug+00] Kokichi Sugihara et al. „Topology-Oriented Implementation–An Approach to Robust Geometric Algorithms“. In: *Algorithmica* 27 (May 2000), pp. 5–20. DOI: 10.1007/s004530010002.

[Zie95] Günter M. Ziegler. *Lectures on Polytopes: Updated Seventh Printing of the First Edition*. 1st ed. Graduate Texts in Mathematics 152. Springer-Verlag New York, 1995.

Appendix

The implementation can be found in **this repository:**
`https://github.com/davidimmanuelhartel/approximate_vertex_enumeration`

ϵ	input polytope	approx DDM algo	graph DDM algo without duplicatex	graph DDM algo
10	120	19	20	10
5	120	20	20	13
1	120	94	38	23
0.9	120	82	40	21
0.8	120	84	40	23
0.7	120	43	42	24
0.6	120	140	48	32
0.5	120	206	62	35
0.4	120	144	66	40
0.3	120	107	84	53
0.2	120	143	110	68
0.1	120	219	170	125
5e-2	120	121	236	164
1e-2	120	120	338	120
1e-3	120	120	338	120
0	120	333	338	333

Table 2: Number of vertices of a polytope with 170 inequalities and 120 vertices under algorithm 4

Declaration of Originality

I hereby declare that, to the best of my knowledge, this thesis is the product of my own independent work. All content and ideas drawn from external sources, directly or indirectly, published or unpublished, are indicated as such. This thesis has neither been previously submitted in whole, nor in parts, for a degree at this or any university.

Jena, August 29^{th}, 2021 David Hartel