LIÈGE université
Library

https://lib.uliege.be

https://matheo.uliege.be

Master thesis : Symbolic representation of polygons in discrete spaces

Auteur : Bertrand, Alexis
Promoteur(s) : Boigelot, Bernard
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2022-2023
URI/URL : http://hdl.handle.net/2268.2/17643

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

University Of Liège

INFO9014 : Master's thesis

Symbolic representation of polygons in discrete spaces

Author:
BERTRAND Alexis

Promotor:
BOIGELOT Bernard

June, 2023

Abstract

This master's thesis describes a new data structure called *Discrete Polyhedron Decision Diagram* (DPDD), suited for representing symbolically meshed polygons, i.e., the intersection between a discrete mesh and a convex polygon. We develop algorithms for manipulating the data structure such as performing the intersection of a meshed polygon with a set of constraints, or adding points to the structure. Each algorithm is thoroughly explained and presented in pseudo-code. Some examples are also given to illustrate important properties and operations. This master's thesis provides as well a prototype of the data structure in which all the main operations have been implemented.

The data structure is based on the double description method and automata-based representations. DPDD then explicitly stores the face lattice of the polygon while each of its faces also contain meshes. An advantage of this is that we can retrieve the canonical form of any meshed polygon thanks to a rounding process. This leads us to have more efficient operations over the data structure. We are able to easily and efficiently check whether two meshed polygons are equal, as example, and as we have the submesh of each face, we can avoid redundant computations.

Acknowledgements

First, and foremost, I would like to thank my promotor, Bernard Boigelot, for sharing with me its contagious enthusiasm about computer system verification during his lectures and our discussions during the redaction of this master's thesis. Without his listening ear, his patience, and his knowledge, I would not have the opportunity to have completed such a work. His office door was always open, as well as his email box to provide me feedbacks or to answer my questions.

I would also like to thank Pascal Fontaine, and Quentin Louveaux for willing to read and evaluate my work.

Finally, I would like to thank my family for supporting me during my studies.

I would eventually like to give a special thank to my girlfriend, Marie Puissant, for encouraging me throughout these years and especially during the hardest times. Thank you for listening to me while I was constantly talking about my researches and ideas for this master's thesis, reading my work, and helping me when I was stumbling over problems during my redaction. I would not have been able to work that well on this master's thesis without your support.

Contents

Introduction **1**

1 Basic notions **5**
1.1 Linear Algebra 5
1.1.1 Vector Spaces 5
1.1.1.1 Linear Dependence 6
1.1.1.2 Vector Subspaces 6
1.1.1.3 Basis and Dimension 7
1.1.2 Affine Spaces 8
1.1.2.1 Affine Subspaces 8
1.1.2.2 Coordinates 9
1.1.3 Representations of Vector and Affine Spaces 9
1.2 Polyhedra 9
1.2.1 Convex Cones 10
1.2.2 Polyhedra 11
1.2.3 Representation of Polyhedra 13
1.2.4 Operations 14
1.2.4.1 Representation Conversion 14
1.2.4.2 Point Membership 16
1.2.4.3 Emptiness 16
1.2.4.4 Intersection 16
1.2.4.5 Inclusion 17
1.2.4.6 Equality 17
1.2.4.7 Complexity 17
1.2.5 Face Lattices 18
1.2.6 Meshes 19

2 Representing Meshed Polyhedra **21**
2.1 Objectives 21
2.2 Polyhedra Representation 22
2.3 Mesh Representation 25
2.4 Meshed Polyhedra 27
2.4.1 Data Structure 30
2.5 Operations 32
2.5.1 Point membership 32
2.5.2 Intersection 33
2.5.3 Adding a point to a meshed polygon 41

2.5.4 Finding all the points of a mesh that satisfy a set of constraints . . . 47
2.5.5 Rounding a (half-)line . . . 48
2.5.6 Advanced intersection . . . 49

3 Implementation **55**
3.1 Data Structure . . . 55
3.1.1 Representation . . . 56
3.2 Operations . . . 57
3.2.1 Point Membership . . . 57
3.2.2 Coloring . . . 57
3.2.3 Insert Mesh . . . 58
3.2.4 Intersection . . . 61
3.2.5 Marking . . . 62
3.2.6 Add Point . . . 62
3.2.7 Find Points . . . 63
3.2.8 Translation . . . 64
3.2.9 Advanced Intersection . . . 64

4 Conclusions **67**

Bibliography **69**

Introduction

Computer-aided verification is a branch of computer science that aims to verify the correctness of computer systems and software in an automated way. Verification is then the process of checking algorithmically whether the system or software satisfies a set of properties, in all of its executions. To be verified, a program first needs to be written in an abstract formalism before being analyzed[4]. One way to create an abstract model is through state machines: the program is represented as a state machine where each state represents a certain control location, while the actions of the system correspond to the transitions of the machine. It is then possible to verify the program by exploring the reachable states to check whether a given property is violated or not. However, as the number of states of the state machine created may grow exponentially with the description of the system, we may need an additional abstraction to prevent the exploration of each state by the model checker.

Symbolic state-space exploration consists in symbolically representing states to avoid enumerating each one individually. It is then possible to use a specific kind of verification technique called Symbolic Model Checking[11]. Polyhedra, which correspond to finite Boolean combinations of linear constraints, are often used for representing sets of systems of configurations for such state spaces. Symbolically representing polyhedra then plays an important role in symbolic state-space exploration, as it allows efficient algorithms for performing the operations needed during the analysis of systems.

Several symbolic representations of different types of polyhedra have already been covered by Boigelot et al.[5], for Nef polyhedra[3], or I. Mainz[19] with convex polyhedra in high-dimensional spaces. This master's thesis aims to focus on representing symbolically meshed polygons, i.e., 2-dimensional polyhedra. Meshed polygons can be defined as convex sets of integers with discrete periodicities. They may be used to represent state spaces of systems relying on integer variables and that perform linear operations on them. This variant of a convex polyhedron, stemming from the intersection between a polygon and a mesh, can be bounded or not. This master's thesis covers bounded and unbounded meshed polyhedra.

Polyhedra can be symbolically represented in several ways, such as the double description method, and the automata-based representation. The double description method consists in describing polyhedra by two representations: a set of inequalities, and the combination of their set of vertices and extreme

rays. Keeping both these two representations helps speeding up operations over polyhedra.

There are several automata-based representations suited for polyhedra, such as *Real-Vector Automaton* (RVA), *Implicit Real-Vector Automaton* (IRVA)[5], *Convex Polyhedron Decision Diagram* (CPDD), *Decomposed Convex Polyhedron* (DCP), or *Internally Decomposed Convex Polyhedron* (IDCP)[19]. These representations already combined the double description method with the concept of face lattice. The face lattice of a polyhedron represents a partial ordering of its faces. RVA and IRVA can represent convex and non-convex, bounded and unbounded polyhedra. CPDD, DCP, and IDCP only represent convex polyhedra but address one of the main drawbacks of RVA and IRVA, which suffer from combinatorial explosion in high dimensional spaces like the double description method[19].

This master's thesis introduces an original data structure, called the *Discrete Polygon Decision Diagram* (DPDD), to symbolically represent meshed polygons. The data structure, which can be considered as an automata-based representation of meshed polygons, combines the double description of polyhedra, its face lattice, and the notion of submeshes of faces to represent meshed polygons. This data structure presents two main advantages. First, as it is based on the face lattice representation, the data structure allows us to easily compute the canonical form of a meshed polygon. The advantage of having canonical representations can be seen during some operations. As an example, equality testing between two meshed polygons amounts to just performing a syntactic check between the two representations. Another example of this advantage is the point membership operation that consists in checking whether the point belongs to one of the faces of the face lattice. Another advantage of the data structure is that each face contains its own submesh. By including the mesh into the face lattice, the data structure allows us to perform operations such as finding the appropriate rounding of the given meshed polygon more easily, and saves unnecessary computations that would have occurred if the mesh was stored separately from the face lattice of the polygon.

We first present the basic notions used in this master's thesis in Chapter 1. This chapter is divided in two main sections. The first one focuses on linear algebra. We define concepts such as vector and affine spaces, and the representation of such spaces. This section also presents useful theorems and properties used in this work, such as the change of basis, or the decomposition of a vector with respect to a basis. The second section introduces polyhedra. We first express the notion of convex sets and cones before presenting convex polyhedra and a list of operations that can be performed over them. This section also introduces the notion of face lattice which is fundamental in this master's thesis, as well as the definition of meshes.

Chapter 2 first defines the different representations that are the basis of the data structure created in this master's thesis. Meshed polyhedra are defined before explaining the canonical representations of polyhedra, meshes, and meshed polyhedra. Thanks to these representations, we then describe the original data structure called *Discrete Polygon Decision Diagram* (DPDD) by giving a formal

definition. This definition is based on the notion of face lattice and submesh. Indeed, the data structure represents the face lattice of the given polyhedron, where each face corresponds to a subset of the constraints and mesh of the meshed polygon. This chapter also develops algorithms for several operations that can be performed on the data structure.

Chapter 3 presents a prototype implementing the data structure as well as the operations presented as algorithms in Chapter 2. The chapter focuses on the implementation issues, presenting how the different operations work on the data structure through Python code.

Eventually, Chapter 4 concludes this master's thesis with a summary of its contributions, as well as some perspectives for future work.

Chapter 1

Basic notions

In this chapter, we introduce the basic notions that one needs in order to fully understand this master's thesis. We give definitions, theorems and properties on specific topics of linear algebra. Furthermore, we define polyhedra as well as their major operations and further concepts used for the conception of a data structure and its operations.

1.1 Linear Algebra

In this section, we introduce linear algebra definitions, properties, and theorems. The different subsections will help the reader to have the needed background on linear algebra concepts used in this master's thesis.
We also introduce vector and affine spaces and their properties as well as useful theorems that have been used in this work. We also present a canonical representation of vector and affine spaces that has been used in order to develop the data structure created in this master's thesis.

We based the definitions, theorems, and propositions on papers and lectures of E. Charlier[10], M. Rigo[20], I. Mainz[19], Dodson et al.[13], and R. Beezer[2].

1.1.1 Vector Spaces

Definition 1.1.1 *A vector space on $\mathbb{R}$ is a set E associated with two operations:*

$$+ : E \times E \to E$$

and

$$\cdot : \mathbb{R} \times E \to E$$

that satisfies the following properties:

1. *There exists an element 0 of E neutral for $+$: $\forall \vec{e} \in E, \vec{e} + 0 = 0 + \vec{e} = \vec{e}$*
2. *For any element of E there exists an inverse one: $\forall \vec{e} \in E, \exists \vec{f} \in E \mid \vec{e} + \vec{f} = \vec{f} + \vec{e} = 0$*
3. *The addition operation is associative: $\forall \vec{e}, \vec{f}, \vec{g} \in E, (\vec{e}+\vec{f})+\vec{g} = \vec{e}+(\vec{f}+\vec{g})$*
4. *The addition operation is commutative: $\forall \vec{e}, \vec{f} \in E, \vec{e} + \vec{f} = \vec{f} + \vec{e}$*

5. *For any element $\vec{e}, \vec{f}$ of E and $k, \ell \in \mathbb{R}$, we have:*
   a) *The multiplication by a scalar operation is associative: $k \cdot (\ell \cdot \vec{e}) = (k \cdot \ell) \cdot \vec{e}$*
   b) *The scalar multiplication operation is distributive when used with the addition of two reals: $(k + \ell) \cdot \vec{e} = k \cdot \vec{e} + \ell \cdot \vec{e}$*
   c) *The scalar multiplication operation is distributive when used with the addition operation: $k \cdot (\vec{e} + \vec{f}) = k \cdot \vec{e} + k \cdot \vec{f}$*
   d) *The element 1 is neutral for the scalar multiplication operation: $1 \cdot \vec{e} = \vec{e}$*

If E is a vector space on $\mathbb{R}$, then the elements of E are called vectors *and the elements of $\mathbb{R}$ are called* scalars.

1.1.1.1 Linear Dependence

Definition 1.1.2 *Let $n \in \mathbb{N}$, and E be a vector space. A linear combination of the vectors $\vec{x_1}, \ldots, \vec{x_n}$ of E is a vector of the form:*

$$k_1\vec{x_1} + \ldots + k_n\vec{x_n}$$

with $k_1, \ldots, k_n \in \mathbb{R}$. The scalars $k_1, \ldots, k_n$ are called coefficients *of this linear combination. A linear combination of linear combinations of vectors $\vec{x_1}, \ldots, \vec{x_n}$ is still a linear combination of $\vec{x_1}, \ldots, \vec{x_n}$.*

Definition 1.1.3 *The vectors $\vec{x_1}, \ldots, \vec{x_n}$ of E are* linearly independent *if*

$$\forall k_1, \ldots, k_n \in \mathbb{R}, k_1\vec{x_1} + \ldots + k_n\vec{x_n} = 0 \Rightarrow k_1 = \ldots = k_n = 0$$

If vectors are not linearly independent, they are linearly dependent. Therefore, $\vec{x_1}, \ldots, \vec{x_n}$ are linearly independent if the only way to obtain 0 as a linear combination of $\vec{x_1}, \ldots, \vec{x_n}$ is to take all the coefficients equal to 0. These vectors are linearly dependent if there exist non-zero scalars $k_1, \ldots, k_n$ such that $k_1\vec{x_1} + \ldots + k_n\vec{x_n} = 0$.

1.1.1.2 Vector Subspaces

Definition 1.1.4 *Let E be a vector space. A vector subspace of E is a part of E which contains all the linear combinations of the vectors of E.*

Proposition 1.1.5 *Let E, F be two vector spaces with identical definitions of vector addition and scalar multiplication. If F is a subset of E ($F \subseteq E$), then F is a subspace of E.*

Definition 1.1.6 *Let E, F be two vector spaces and $F \subseteq E$. The span of A (span(A)) is the set of all the linear combinations of vectors of A.*

Formally, the span of $\vec{x_1}, \ldots, \vec{x_n}$ is the set:

$$span(\vec{x_1}, \ldots, \vec{x_n}) = \left\{ \sum_{i=1}^{n} k_i\vec{x_i} \mid k_1, \ldots, k_n \in \mathbb{R} \right\}$$

.

Definition 1.1.7 *Let E be a vector space and F_1, F_2 be vector subspaces of E. The sum of these vector subspaces is defined as the set: $F_1 + F_2 = \{\vec{f_1} + \vec{f_2} \mid \vec{f_1} \in F_1, \vec{f_2} \in F_2\}$. This sum represents the smallest vector space containing F_1 and F_2.*

Definition 1.1.8 *Let E be a vector space, F_1, F_2 be vector subspaces of E, and F be the sum of F_1 and F_2. Let us suppose that every $\vec{f} \in F$ can only be uniquely written as $\vec{f} = \vec{f_1} + \vec{f_2}$, then F is called the direct sum and can be written as $F = F_1 \oplus F_2$. In other word, if there is no intersection of F_1 and F_2 ($F_1 \cap F_2 = 0$), then $F = F_1 \oplus F_2$ is the direct sum of F_1 and F_2.*

1.1.1.3 Basis and Dimension

Definition 1.1.9 *Let E be a vector space. The set $B \subseteq E$ is a basis of E if any element of E can be written as a linear combination of vectors of B and the elements of B are linearly independent. In other words, a basis is a linearly independent spanning set.*

Proposition 1.1.10 *If the vector space E is spanned by a finite number of vectors, then the basis of E consists of this finite number of vectors.*

Definition 1.1.11 *Let E be a vector space spanned by a finite number of vectors, the dimension of E (dim(E)) corresponds to the unique number of basis elements of E and E is called finite dimensional. If E is not spanned by a finite number of vectors, E is called infinite dimensional and dim(E) $= \infty$.*

Proposition 1.1.12 *Let E, F be vector spaces. Let us assume that E is finite dimensional and $F \subseteq E$, then:*

- *dim(F)$\leq$dim(E)*
- *dim(F) $=$ dim(E) $\iff E = F$*

Proposition 1.1.13 *Let E be a vector space of dimension $n \in \mathbb{N}$ and $F \subseteq E$ of dimension $m \in \mathbb{N}$. Given a basis of the vector E, the subspace F can be described as a system of $n - m$ independent homogeneous linear equations: $a_{i1}\vec{x_1} + \ldots + a_{in}\vec{x_n} = 0$.*

Theorem 1.1.14 *(Decomposition of a vector in a basis) Let $B = (\vec{b_1}, \ldots, \vec{b_n})$ a basis of E. Every vector of E has a unique decomposition in the basis B: $\forall \vec{x} \in E$, there exist unique scalars $x_1, \ldots, x_n \in \mathbb{R}$ such as $\vec{x} = x_1\vec{b_1} + \ldots + x_n\vec{b_n}$.*

Definition 1.1.15 *Let $B = (\vec{b_1}, \ldots, \vec{b_n})$ be a basis of the vector space E, and $\vec{x} \in E$. The coefficients $x_1, \ldots, x_n \in \mathbb{R}$ of the unique decomposition of $\vec{x}$ in the basis B are called the* coordinates *of $\vec{x}$ with respect to the basis B.*

Theorem 1.1.16 *(Change of basis) Let $B = (\vec{b_1}, \ldots, \vec{b_n})$ and $B' = (\vec{b'_1}, \ldots, \vec{b'_n})$ two bases of the vector space E and $\vec{e}, \vec{e'}$ be elements of respectively B and B', we obtain:*

$$\vec{e'} = \mathcal{M}(B, B')\vec{e},$$

where the matrix $\mathcal{M}(B, B')$ is called the coordinate transformation matrix *from the basis B to the basis B' and always exists.*

1.1.2 Affine Spaces

Definition 1.1.17 *Let E be a vector space, and A be a non-empty set. Let us define as well the operation $p+\vec{x} \in A$ for any vector $\vec{x} \in E$ and element $p \in A$ such that it satisfies the following conditions:*

- $p + 0 = p$
- $(p + \vec{x}) + \vec{y} = p + (\vec{x} + \vec{y})$
- $\forall q \in A, \exists \vec{x} \in E : q = p + \vec{x}$, *where $\vec{x}$ is unique.*

An element $p \in A$ is called a point *and $p + \vec{x}$, where $\vec{x} \in E$, is the* translation *of the point p by the vector $\vec{x}$.*

Definition 1.1.18 *Let $p_1, \ldots, p_n \in A$. An affine combinations of $p_1, \ldots, p_n$ is a linear combination:*

$$\sum_{i=1}^{n} k_i p_1 = k_1 p_1 + \ldots + k_n p_n \text{ such that } \sum_{i=1}^{n} k_i = 1$$

Definition 1.1.19 *Let $p_1, \ldots, p_n \in A$. The affine hull of this set is:*

$$\left\{ \sum_{i=1}^{n} k_i p_i \mid k_1, \ldots, k_n \in \mathbb{R} \text{ and } k_1 + \ldots + k_n = 1 \right\}$$

In other words, the affine hull of a set is the set of all its affine combinations.

Definition 1.1.20 *The points $p_1, \ldots, p_n \in A$ are affinely dependent if one of them belongs to the affine hull of the others. If not, they are affinely independent.*

1.1.2.1 Affine Subspaces

Definition 1.1.21 *Let A be an affine space over a vector space E. Then $A' \subseteq A$ is an affine subspace of A if there exists $p \in A$, and a vector subspace $F \subseteq E$ of E such that $A' = p + F = \{p + \vec{x} \mid \vec{x} \in F\}$.*

The set F is called the direction *of A'.*

Proposition 1.1.22 *The intersection between two affine subspaces is either empty or an affine subspace.*

Definition 1.1.23 *Let $A = p + F$ and $A' = p' + F'$ be two affine subspaces. If $F \subseteq F'$ or $F' \subseteq F$, then A and A' are parallel. This is denoted by $A \mid\mid A'$.*

Definition 1.1.24 *Let E be a vector space, A be an affine space and A' an affine subspace of A over E. The dimension of A corresponds to the dimension of its direction. If $A' = p + F$ where dim(F)$= 1$, then A' is called a line. If dim(F)$= 2$, then A' is called a plane. Finally, if dim(F)$= n - 1$, where $n =$dim(E), then F is called an* hyperplane.

1.1.2.2 Coordinates

As affine spaces are translated vector spaces, we can describe them thanks to the description of their vector space and its translation. Vector spaces can be described thanks to their basis. The elements of an affine space are characterized by *affine coordinates*, defined as follows:

Definition 1.1.25 *Let A be an affine space and E a vector space. The affine coordinates of A over E consists in a point $o \in A$ (origin), and a basis $B = \{\vec{b_1}, \ldots, \vec{b_n}\}$ of the vector space E. The coordinates of a point $p \in A$ are the coordinates of the vector $\overrightarrow{po}$ with respect to the basis B.*

Proposition 1.1.26 *Let A be an affine space over the vector space E of dimension n and an affine subspace $A' \subseteq A$ of dimension m. A' can be described as a system of $n - m$ independent linear equations: $a_{i1}\vec{x_1} + \ldots + a_{in}\vec{x_n} = b$. The direction of A' can also be described as: $a_{i1}\vec{x_1} + \ldots + a_{in}\vec{x_n} = 0$.*

1.1.3 Representations of Vector and Affine Spaces

Definition 1.1.27 *Let $M \in \mathbb{R}^{n\times m}$ be a matrix. Let m_{ij} be an element of M. m_{ij} is a* pivot *if and only if $m_{ij} = 0$ and $m_{k\ell} \neq 0$ when $i \leq k, \ell \leq j$ and $(i,j) \neq (k,\ell)$.*

Definition 1.1.28 *Let $M \in \mathbb{R}^{n\times m}$ be a matrix. M is said to be in* row echelon form *iff all its non-zero rows contain a pivot, all its zero rows are located below the non-zero rows, and each pivot is strictly to the right of the pivot of the previous non-zero rows.*

Definition 1.1.29 *Let $M \in \mathbb{R}^{n\times m}$ be a matrix. M is said to be in reduced row echelon form if:*

- *The pivot of each non-zero row is 1.*
- *M is in row echelon form and has no zero rows.*
- *The pivot of each row is the only non-zero element of each column.*

Theorem 1.1.30 *(Gaussian Elimination) Thanks to elementary row operations, any matrix representing an equation system can be transformed to an equation system under its unique reduced row echelon form.*

Thanks to Theorem 1.1.30, and Propositions 1.1.13 and1.1.26, stating that any vector or affine space can be described as systems of linear equations, it is possible to create a canonical representation. Canonical means that there is a unique way to represent this matrix. It is then possible to compare them as they have to be represented identically to be equal.

1.2 Polyhedra

We introduce polyhedra in this section. We start with a definition and description of convex sets, and then cones which are specific types of polyhedra. We also present the bounded and unbounded forms of polyhedra. Polygons, i.e., two-dimensional polyhedra, were the main focus of this master's thesis as the

data structure and operations were based on these two-dimensional polyhedra. After introducing the different representations of polyhedra, we compare their theoretical complexity on multiple operations on polyhedra. We then introduce the notion of face lattice of polyhedra as well as the components of the structure such as the faces of polyhedra. We then conclude this section with the notion of meshes. The two last notions are a really important part of this master's thesis as the data structure developed as well as its operations deeply depend on these principles.

We based the following definitions, theorems, and properties on papers of I. Mainz[19], W. Fenchel[14], Boyd et al.[7], A. Brøndsted[8], B. Grünbaum[16], and lectures of Gallier et al.[15] and KC. Border[6].

1.2.1 Convex Cones

Before defining convex cones we need to introduce the definition of convex sets. Once done, we give definitions, and properties of cones as well as examples of some particular convex cones.

Definition 1.2.1 *A set S is convex if, for any point $x_1, x_2 \in S$, the segment $[x_1, x_2] \in S$. In other words, a set is convex if any segment between two points in the set is entirely within the set.*

Definition 1.2.2 *Let $x_1, \ldots, x_n \in \mathbb{R}^m$. The convex combination of these points is the point $\sum_{i=1}^{n} k_i x_i$, where $k_1, \ldots, k_n \in \mathbb{R}_{\geq} 0$ and $\sum_{i=1}^{n} k_i = 1$.*

Definition 1.2.3 *Let C be a convex set. The* convex hull *of C (conv(C)) is the set of all the convex combinations.*

Definition 1.2.4 *Let C be a convex set. A point $p \in C$ is an* extreme point *of C if there does not exist two points $q, r \in C$ such that $p \in [q, r]$, $p \neq q$ and $p \neq r$.*

Definition 1.2.5 *Let E be a vector space on $\mathbb{R}^n$ with O as origin. A subset C of E is a* convex cone *if O is in C, and there exists $\vec{x}, \vec{y} \in E$ such that if $\vec{x}, \vec{y} \in C$, then $k\vec{x} + \ell\vec{y} \in C$ for any $k, \ell \in \mathbb{R}_{\geq 0}$. In other words, if the vectors $\vec{x}, \vec{y} \in C$, then the linear combination of non-negative coefficient is also in C. The vectors $\vec{x}$ and $\vec{y}$ are called* rays *of C.*

Definition 1.2.6 *Let $S \subseteq \mathbb{R}^n$. S is a* generalized cone *if there exists a cone $C \subseteq \mathbb{R}^n$, and a point $p \in \mathbb{R}^n$ such that $S = p + C$.*

Definition 1.2.7 *Let $x_1, \ldots, x_n \in \mathbb{R}^m$. A* conic combination *of these points is a point $\sum_{i=1}^{n} k_i x_i$, where $k_1, \ldots, k_n \in \mathbb{R}_{\geq 0}$ and $\sum_{i=1}^{n} k_i = 1$.*

Definition 1.2.8 *Let C be a convex set. The* conic hull *of C (cone(C)) is the set of all the conic combinations.*

Definition 1.2.9 *Let C be a convex cone. A point $x \in C$ is an* extreme ray *of C if there does not exist any linearly independent $y, z \in C$, and $k, \ell \in \mathbb{R}_{\geq 0}$ such that $x = ky + \ell z$.*

Note 1.2.10 *Here are some particular (convex) cones:*

- *The empty set* $\emptyset$.
- *The singleton* O.
- *Any closed (half)-line whose origin is* 0.
- *Any closed (half)-line passing through* O *is a generalized cone.*
- *Any affine subspace is a convex cone.*

1.2.2 Polyhedra

Convex polyhedra are convex sets. We introduce here the reader to unbounded and bounded polyhedra as well as polyhedral cones. We end this section with a quick definition of polygons and provide some examples.

Definition 1.2.11 *A set* $P \subseteq \mathbb{R}^n$ *is a* (convex) polyhedron *if it is a non-empty finite intersection of closed halfspaces – i.e. the sets of the form* $A\vec{x} \leq \vec{b}$ *when* $\vec{x} \in \mathbb{R}^n$ *which is the set of solutions to a finite system of linear inequalities, or the convex hull of a finite set of points.*

Definition 1.2.12 *A set* $P \subseteq \mathbb{R}^n$ *is a* polytope *if it is the convex hull of a non-empty finite set. In other words, a polytope is a bounded polyhedron.*

Figure 1.1 shows an example of a two-dimensional polytope defined by the set of inequalities:

$$3y\ geqx + 2, y \geq 2x - 6, 3y \leq -x + 17, \text{ and } y \leq 4x - 3$$

.

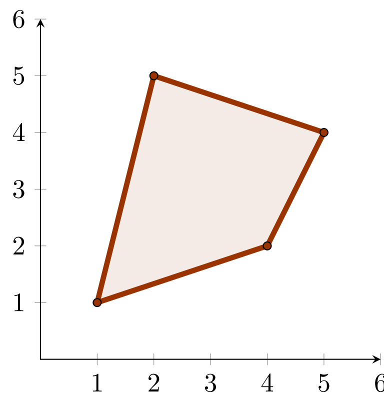


Figure 1.1: Polytope in dimension 2.

Definition 1.2.13 *The set* $C \subseteq \mathbb{R}^n$ *is a* polyhedral cone *if it is a cone that is also a polyhedron.*

Figure 1.2 represents a polyhedral cone defined by the set of inequalities:

$$4y - x \leq 0, 5y - 4x \geq 0$$

.

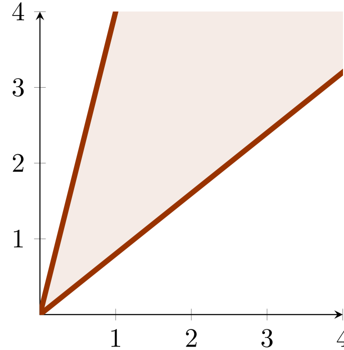


Figure 1.2: Polyhedral cone.

Theorem 1.2.14 *(Minkowski-Weyl) Every polyhedral cone in $\mathbb{R}^n$ is a finitely generated convex cone. Let $C \in \mathbb{R}^n$ be a polyhedral cone, then C can be written as $C = A\vec{x} \leq \vec{b}$, where $A \in \mathbb{R}^{m\times n}$, $\vec{b} \in \mathbb{R}^n$, and $m \geq 0$. In other words, C corresponds to the intersection of a finite set of closed half-spaces, which corresponds to a finitely generated convex cone.*

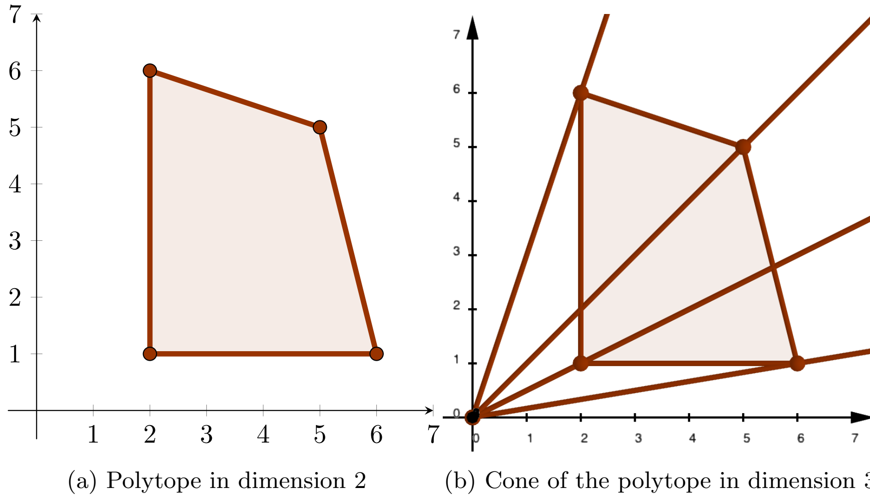
Definition 1.2.15 *Let P be a polyhedron on $\mathbb{R}^n$. Let us suppose that P is not a cone and $P = P(A, b) = \{A\vec{x} \leq \vec{b}\}$, with A a $m \times n$ matrix and $\vec{b} \in \mathbb{R}^m$. Its* homogenization *is defined as*

$$C(P) = P\left(\begin{pmatrix} A & -b \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}\right),$$

which is a polyhedral cone in $\mathbb{R}^{n+1}$

Proposition 1.2.16 *Let $P \subseteq \mathbb{R}^n$ be a polyhedron. P can be represented as a convex cone in $\mathbb{R}^{n+1}$.*

Figure 1.3 shows a polytope in two dimensions and its representation by a cone in three dimensions.



(a) Polytope in dimension 2 (b) Cone of the polytope in dimension 3

Figure 1.3: A 2D polytope and its cone representation in 3D.

In this master's thesis, we only consider polygons, i.e., 2-dimensional polyhedra. Now that we have defined polyhedral sets and their properties in any dimension, we can easily define polygons as two-dimensional polyhedra, keeping then the same properties given to polyhedra in this section.

1.2.3 Representation of Polyhedra

In Definition 1.2.11, we defined polyhedra in two ways. We will now describe in detail these two representations and show that they are equivalent.

Definition 1.2.17 *The* half-space representation *(H-representation) of a polyhedron is defined as the non-empty finite intersection of finite set of closed halfspaces. Let A be a matrix $n \times m$ and $\vec{b} \in \mathbb{R}^n$. The H-representation of the polyhedron P is the system of n inequalities $A\vec{x} \leq \vec{b}$, with $\vec{x} \in \mathbb{R}^n$.*

Definition 1.2.18 *The* vertex representation *(V-representation) of a polyhedron is defined as the convex hull of a finite set of points. Let P be represented by n vertices $x_1, \ldots, x_n$ and m rays $r_1, \ldots, r_m$. The V-representation of the polyhedron P is $P = conv(x_1, \ldots, x_n) + cone(r_1, \ldots, r_m)$*

Theorem 1.2.19 *(Minkowski-Weyl) The following statements are equivalent:*

- *Let A be a matrix $n \times m$ and $\vec{b} \in \mathbb{R}^n$, then $P = P(A, b) = \{A\vec{x} \leq \vec{b}\}$, with $\vec{x} \in \mathbb{R}^n$ (P is a H-polyhedron).*
- *Let P be represented by n vertices and m rays, then $P = conv(x_1, \ldots, x_n) + cone(r_1, \ldots, r_m)$ (P is a V-polyhedron).*

Let us take the V-representation of a polyhedron P. This representation is defined by the convex hull of the extreme points of the polyhedron: $conv(x_1, \ldots, x_n)$. Extreme points can be seen as a solution of a set of inequalities. This set of inequalities $A\vec{x} \leq \vec{b}$ represents the hyperplanes of the extreme points. We can then notice that the two representations are equivalent as the H-representation

is the set of inequalities determining the hyperplanes of the extreme points expressed in the V-representation through the convex hull of these points.

1.2.4 Operations

In the previous section, we have presented how polyhedra can be described, namely by H-description and V-description. In this section, we define several operations that can be performed over polyhedra. For each operation, we also give the complexity for the two representations. Then, we discuss another way of representing polyhedra based on using jointly both the H-representation and the V-representation.

The representation description and the operations complexity of the different representations below, as well as the Fourier-Motzkin algorithm, are based on the papers of I. Mainz[19], RJ. Jing[18], [16], and the lectures of J. Walsh[23].

1.2.4.1 Representation Conversion

Depending on the operations we wish to perform on polyhedra, it is sometimes better to put polyhedra into a specific form (H-representation, or V-representation). Performing this action before the operation can give us better complexity.

Thanks to Theorem 1.2.19, we know that the H-representation and the V-representation of a same polyhedron are equivalent. Therefore, we can convert one to another depending on which one is better to perform an operation. The *representation conversion* operation is then defined as the reconstruction of a representation from the other one.

Given a polyhedron in V-representation, we can determine its H-representation thanks to the Fourier-Motzkin elimination algorithm. This algorithm aims to eliminate variables from a system of linear inequalities. For representation conversion, we use Fourier-Motzkin as follows:

Let n be the number of variables in the V-polyhedron given to the Fourier-Motzkin elimination algorithm. The V-representation consists of linear inequalities of the form $\vec{a} \cdot \vec{x} \leq b$. The algorithm returns the system of inequalities representing the H-representation of the V-polyhedron given as input. The Fourier-Motzkin elimination algorithm then performs a loop from $i = 1$ to n with the following steps:

1. For each pair of inequalities involving x_i, the algorithm performs the following operations:
   a) Multiply one of the inequalities by a scalar such that the coefficient of x_i has an opposite sign with the coefficient of x_i in the other inequality.
   b) Eliminate the variable x_i by adding the two inequalities.
2. Remove x_i from the system of inequalities.

Grouping all the sets of inequalities found by the algorithm corresponds to the H-representation of the polyhedron. However, some constraints in this set may be redundant and finding this redundancy is not trivial.

Let us take a look at an example to better understand how the Fourier-Motzkin elimination algorithm works:

Let P be a polyhedron under its V-representation whose system of inequalities corresponds to the following one:

$$5x + 2y \leq 2 \tag{1.1}$$

$$10x - 3y \leq 6 \tag{1.2}$$

$$7x + 6y \leq 1 \tag{1.3}$$

We first eliminate the variable x.

1. We multiply Equation 1.1 by -2 and Equation 1.2 by 1 in order to have the two variables coefficients with opposite signs.

2. We add Equation 1.1 to Equation 1.2:

$$-10x - 4y + 10x - 3y \leq -4 + 6 \Leftrightarrow -7y \leq 2.$$

We then eliminate the variable y.

1. We multiply Equation 1.1 by -3 and Equation 1.3 by 1 in order to have the two variables coefficients in opposite signs.

2. We add Equation 1.1 to Equation 1.3:

$$-15x - 6y + 7x + 6y \leq -6 + 1 \Leftrightarrow -8x \leq -5.$$

The new system of inequalities is then:

$$-7y \leq 2$$

$$-8x \leq -5$$

This corresponds to the H-representation of the V-polyhedron given as input to the Fourier-Motzkin elimination algorithm.

Given a polyhedron in H-representation, we can determine its V-representation by finding the points where the hyperplanes defined in the H-representation intersect. This operation is known as vertex enumeration. However, such an operation is quite complex. Indeed, the complexity depends on the dimension of the polyhedron as well as the size of the H-representation. The complexity of such an operation is then exponential in the dimension of the polyhedron in the worst case.

Because of the complexity of these two operations, another representation of polyhedra is used: the *double description* method. This representation consists in jointly maintaining the H-representation and V-representation of polyhedra. Doing so prevents the use of representation conversion when no modification on the polyhedron has to be performed, allowing to use the representation giving the best complexity. Using this representation also helps with representation conversion: finding redundancy in the H-representation when performing the Fourier-Motzkin elimination is then easier if the two representations are kept.

The *double description* then maintains the H-representation, and V-representation of polyhedra, which eases the execution of several operations on polyhedra.

1.2.4.2 Point Membership

The *point membership* operation consists in determining if a given point is within a given polyhedron:
Given a polyhedron $P \subseteq \mathbb{R}^n$ and a point $x \in \mathbb{R}^n$, the goal is to determine whether $x \in P$.

From the point of view of complexity, it is easy to determine whether a point is located or not within a given polyhedron when in a H-representation as it corresponds to checking whether the point violates one of the inequalities. If the polyhedron is in V-representation, however, it corresponds to solving a system of linear equations which is a polynomial problem. When performing this operation with a polyhedron represented in *double description*, it corresponds to performing the operation on the H-representation of the polyhedron.

1.2.4.3 Emptiness

The *emptiness* operation consists in determining whether a given polyhedron is empty:
Given a polyhedron P, the goal is to determine whether $P = \emptyset$.

From the point of view of complexity, it is trivial to check if a polyhedron in V-representation is empty as it consists in checking if there is any point or ray in the polyhedron. When in H-representation, it consists in checking whether any point satisfies the set of inequalities which can be done in polynomial time. As there is no modification to the polyhedron with this operation, the *double description* allows to only take into account the V-representation to perform the operation.

1.2.4.4 Intersection

The *intersection* operation consists in computing a new polyhedron equal to the intersection between two given polyhedra, i.e., containing all the points that belong to both polyhedra:
Given two polyhedra $P, Q \subseteq \mathbb{R}^n$, the goal is to compute the new polyhedron $R \subseteq \mathbb{R}^n$ such that $R = P \cap Q$.

If the two polyhedra are under H-representation, the intersection consists in combining the constraints of the two polyhedra. Therefore, if we want to perform this operation, we need to convert any polyhedron from the V-representation to the H-representation. This initial conversion is not needed with *double description*. However, as the intersection operation operates only on the H-representation of polyhedra, it also only returns the H-representation of the new polyhedron. It is then needed to convert this H-representation into the V-representation of the polyhedron to keep the two representations for the *double description*.

1.2.4.5 Inclusion

The *inclusion* operation consists in determining whether a given polyhedron P_1 is entirely within another given polyhedron $P2$, i.e., when all points of P_1 belongs to P_2:
Given two polyhedra $P_1, P_2 \subseteq \mathbb{R}^n$, the goal is to determine whether $P_1 \subseteq P_2$.

In order to perform this operation, we can just take the V-representation of the polyhedron we wish to know whether it is included in the second polyhedron in H-representation. We then have to check if all points and rays of the polyhedron in V-representation satisfy the linear inequalities of the polyhedron in H-representation. From the point of view of complexity, depending on the representation of the polyhedra, we may need to convert them to another representation which is never the case in the *double description*.

1.2.4.6 Equality

The *equality* operation consists in determining whether a given polyhedron P_1 is equal to another given polyhedron P_2, i.e., when all points of P_1 belongs to P_2 and all points of P_2 belongs to P_1:
Given two polyhedra $P_1, P_2 \subseteq \mathbb{R}^n$, the goal is to determine whether

$$P_1 = P_2 \iff P_1 \subseteq P_2 \text{ and } P_2 \subseteq P_1.$$

This operation is quite simple to perform when the two polyhedra are represented in V-representation or H-representation as it consists in *checking syntactically* the equality between two representations. However, if we have a polyhedron under the V-representation and the other one under the H-representation, we need to convert them before checking each representation. When using the *double description*, we can just check syntactically representation type by representation type.

1.2.4.7 Complexity

You can find here a recapitulative table of complexity of the operations given above based on the representation of polyhedra. (Table 1.1) A color code is given depending on the complexity.

- Easy problem
- Difficult problem

Table 1.1: Recapitulative table of operations complexity

	Conversion	$x \in P$	$P = \emptyset$	$P \cap Q$	$P \subseteq Q$	$P = Q$
H-Representation						
V-Representation						
Double Description						

1.2.5 Face Lattices

We present in this section face lattices which are combinatorial information about the geometrical components of polyhedra. We first define faces of polyhedra and their properties and relations before describing the face lattice method to represent polyhedra.

We based the definitions, theorems, and propositions of this section from the articles of Allamigeon et al.[1] and I. Mainz[19], and the lectures of K. Chandrasekaran[9] and Gärtner et al.[17].

Definition 1.2.20 *Let P be a polyhedron. The inequality $\vec{a} \cdot \vec{x} \leq b$ is valid for P if $\vec{a} \cdot \vec{x} \leq b$ for all $\vec{x} \in P$.*

Definition 1.2.21 *Let P be a polyhedron and $F \subseteq P$. If $F = P$ or is the intersection of P with an inequality $\vec{a} \cdot \vec{x} \leq b$, where $\vec{a} \cdot \vec{x} \leq b$ is a subset of the inequalities of P, then F is called a* face *of P.*

Theorem 1.2.22 *Let P be a polyhedron whose constraints are $C = A\vec{x} \leq \vec{b}$ and $F \subseteq P$. F is a face of P iff $F \neq \emptyset$ and $F = \vec{a} \cdot \vec{x} = b$, with $\vec{x} \in P$, for some subsystem $\vec{a} \cdot \vec{x} \leq b$ of $A\vec{x} \leq \vec{b}$.*

Theorem 1.2.23 *(Characterization of faces) Let P be a polyhedron whose constraints are $C = A\vec{x} \leq \vec{b}$. All its faces are obtained by converting some inequalities of $A\vec{x} \leq \vec{b}$ into equations, i.e., turning an inequality $\vec{a} \cdot \vec{x} \leq b$ into $\vec{a} \cdot \vec{x} = b$.*

Proposition 1.2.24 *A polyhedron P has a finite number of faces.*

Proposition 1.2.25 *Let P be a polyhedron and F, F' be faces of P. $F \cap F'$ is also a face.*

Definition 1.2.26 *A partial order $\preceq$ on a set S is a relation with the following properties for any $x, y, z \in S$:*

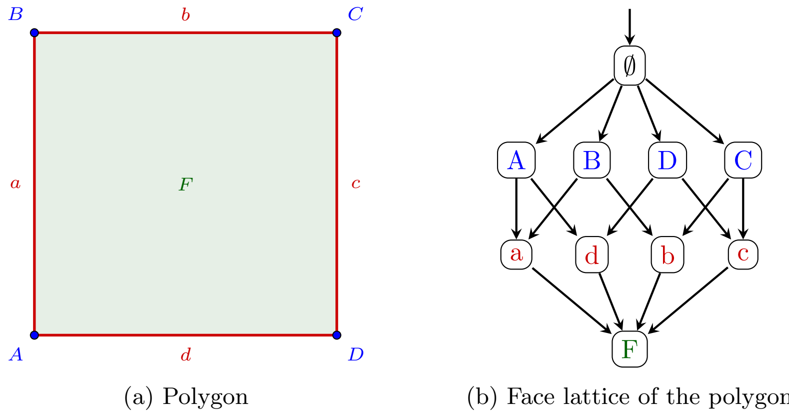
- *reflexive: $x \preceq x$*
- *anti-symmetric: if $x \preceq y$ and $y \preceq x$, then $x = y$*
- *transitive: if $x \preceq y$ and $y \preceq z$, then $x \preceq z$*

Definition 1.2.27 *A* lattice *is a partially ordered set S in which every pair of elements $x, y \in S$ has a unique minimal upper bound in S, called* join, *and a unique maximal lower bound in S, called* meet.

Definition 1.2.28 *Let P be a polyhedron. The* face lattice *$\mathcal{F}$ of P is the set of all the faces of P partially ordered by set inclusion.*

Theorem 1.2.29 *Let P be a polyhedron which constraints are $C = A\vec{x} \leq \vec{b}$, F a face, and $\mathcal{F}$ the face lattice of P. We have $F \in \mathcal{F}$ iff F correspond to the set of solutions of the system obtained after converting some inequalities of $A\vec{x} \leq \vec{b}$ into equations.*

To better understand the concept of face lattice, Figure 1.4 represents the face lattice of a given polygon.



(a) Polygon (b) Face lattice of the polygon

Figure 1.4: A polygon and its face lattice.

1.2.6 Meshes

In this section, we describe the notion of mesh, and explain the elements constituting meshes.

Intuitively, a mesh represents the values that a point may take. These values are characterized by a set of periods between them.

A mesh can then be represented as an initial point with a set of periods.

Definition 1.2.30 *A mesh in $\mathbb{R}^n$ is a set $S \subseteq \mathbb{R}^n$ of the form*

$$S = \{s_0 + k_1\vec{v_1} + \ldots + k_m\vec{v_m} \mid k_1, \ldots, k_m \in \mathbb{Z}\},$$

where $m \geq 0$ and $s_0, \vec{v_1}, \ldots, \vec{v_m} \in \mathbb{R}^n$. The point s_0 is called the initial point *of the mesh while the vectors $\vec{v_1}, \ldots, \vec{v_m}$ are called the* period *of the mesh.*

A point then belongs to a mesh if it is equal to the initial point of the mesh to which an integer combination of the vectors of the period has been added.

Chapter 2

Representing Meshed Polyhedra

In this chapter, we address the problem of representing symbolically meshed polyhedra which correspond to intersections between meshes and polyhedra. Meshed polyhedra are the focus of this master's thesis that aims to symbolically represent them in a data structure. We then define terms linked to meshed polygons and explain how they can be represented separately and combined. Once we have explained the scope, we introduce an original data structure for representing meshed polygons. We end this chapter by defining the operations we want to perform over this data structure and present algorithms that allow us to perform correctly these operations. In this section, we aim to explain more specifically what is the scope of this master's thesis by defining the different elements of meshed polygons and their respective representation before combining them. We also share with the reader the thought process behind some choices made for defining the representation that will be described in the next section.

2.1 Objectives

In this master's thesis, we aim to create a data structure to represent meshed polygons, also called discrete polygons, on which we could perform efficient operations.

I. Mainz[19], and Boigelot et al.[5] already published ways to symbolically represent polyhedra. We built on these papers to create the basis of our data structure, especially for the representation of (non-meshed) polygons.

The main difficulty we encountered was finding a canonical way to represent meshed polygons in order to easily compare two instances of the data structure and immediately see whether they are equal or not. To do so, we define the canonical representation of polygons, the canonical representation of meshes, and find a way to combine these two canonical representations together to form the canonical representation of meshed polygons.

Even if we only consider polygons, the data structure was created with the goal of being as general as possible, in order to easily reuse or enhance the data structure to use it with any dimensional meshed polyhedra. The data structure given could then be used to describe any meshed polyhedra. The algorithms

proposed for the manipulation operations would need to be adapted, which is a non-trivial task. When presenting the algorithms, we precise whether they can be used with polyhedra which have a higher dimension than polygons.

2.2 Polyhedra Representation

In order to represent polygons, we used the concept of face lattice of polyhedra that we already defined in Section 1.2.5.

We then define the representation of a polyhedron as a directed acyclic graph in which each node corresponds to one of its faces.

By definition, each node then corresponds to a subset of saturated constraints retrieved from the set representing the polyhedron. We can then label each node as this subset.

Transitions between nodes correspond to the transitive reduction of the partial order of the face lattice. A transition between two faces is labeled by the constraints that are no longer saturated by the second one. In order to better understand this concept, we give in Figure 2.1 an example of two faces of a polyhedron as well as the transition between them.

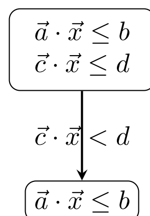
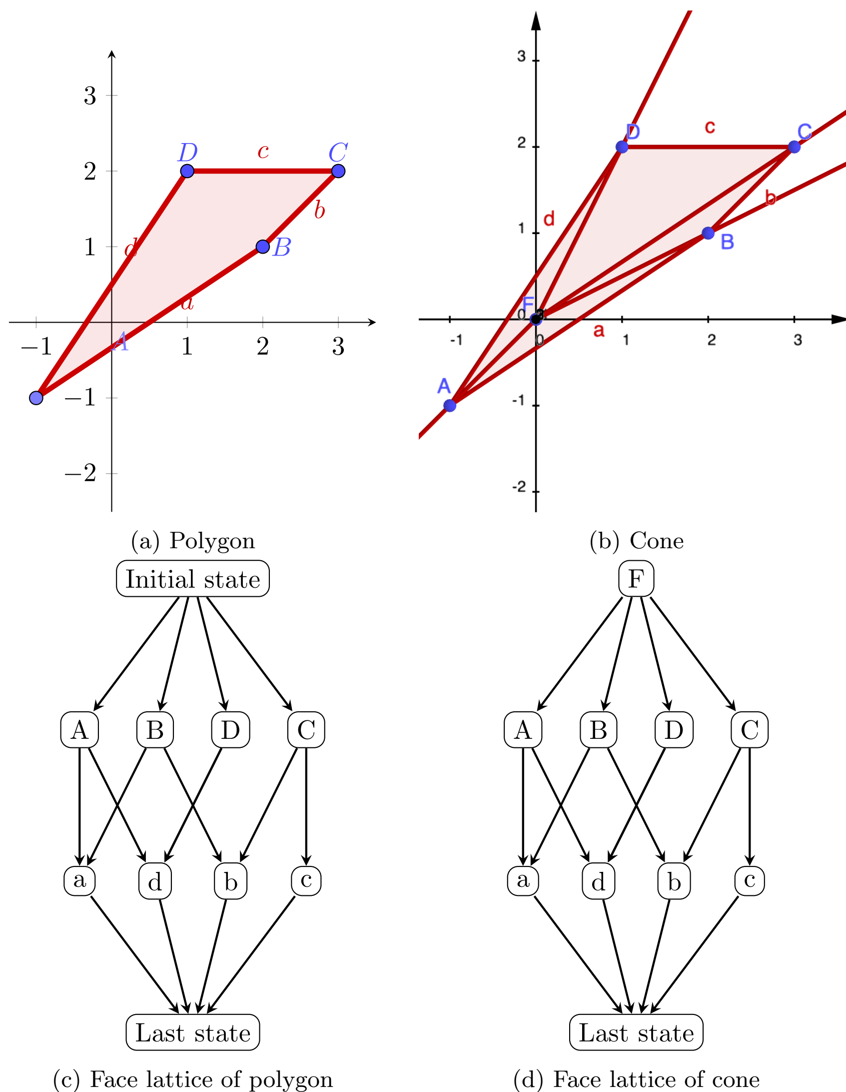


Figure 2.1: Example of the transition between two faces.

Finally, we define a special face as the initial state, which corresponds to the saturation of all the constraints of the polyhedron. It is easy to see which vertex it represents when given a polyhedral cone. However, it may be more complicated to understand this when given a polytope for example. To represent this graphically, we use Proposition 1.2.16, which states that polyhedra can be represented as a cone in the same dimension augmented by 1. Intuitively, we can then understand that the face saturating all constraints corresponds to the vertex of the cone of the same dimension augmented by 1 as it is not possible to saturate all constraints in the given dimension but it is in the dimension augmented by 1. Therefore the cone found by the homogenization (Theorem 1.2.15) of a polyhedron is represented identically as the given polyhedron. We can see this correspondence through Figure 2.2 which gives the representation of a polygon and the representation of the cone resulting from the homogenization of the given polygon. Proposition 1.2.16 already presented this property of the face lattice.



(a) Polygon

(b) Cone

(c) Face lattice of polygon

(d) Face lattice of cone

Figure 2.2: Face lattice of a polygon and its cone representation.

We give in Figure 2.3 a comparison of the face lattice with and without labels of a given polyhedron. This example may help anyone to better understand the concept of face lattice and how it is applied to polyhedra but also how it is linked to the double description method.

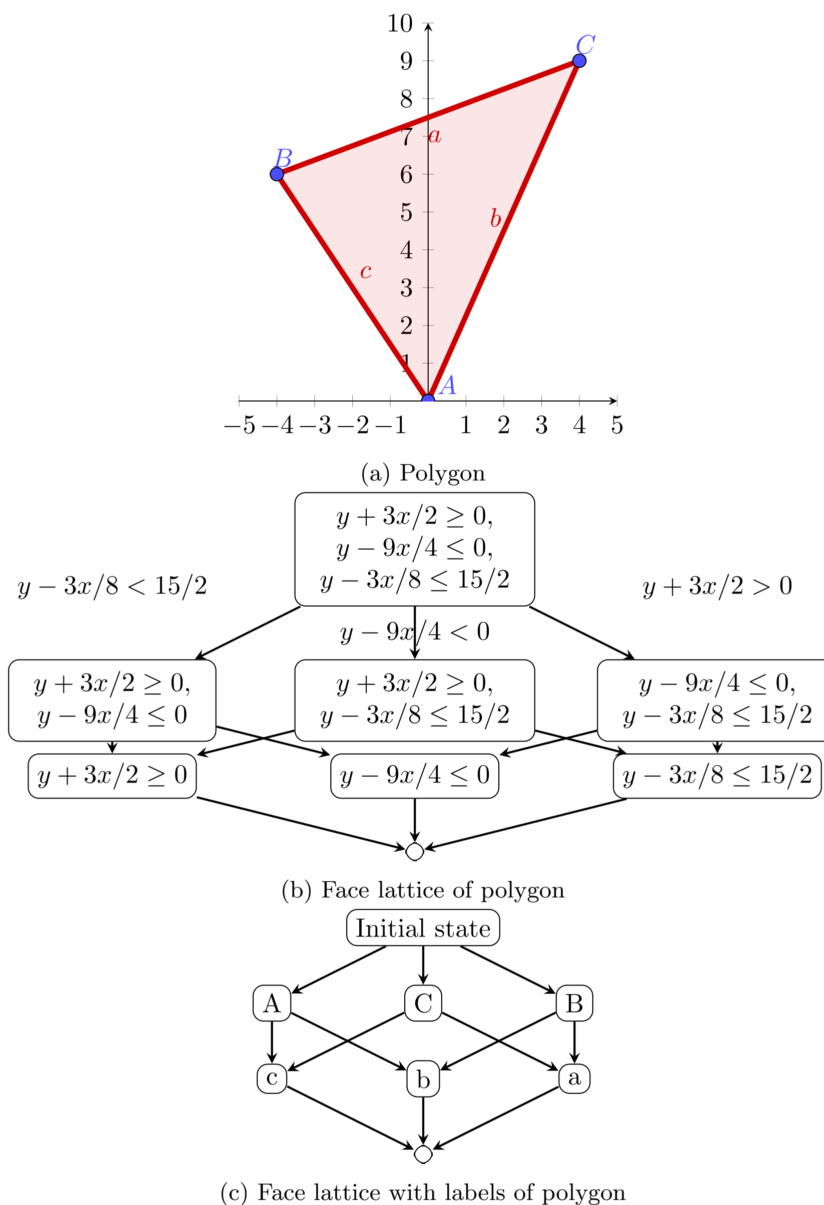


Figure 2.3: Face lattice of the polygon which constraints are:
$y + 3x/2 \geq 0, y - 9x/4 \leq 0$, and $y - 3x/8 \leq 15/2$.

In this example, we can easily notice that the representation includes the double description of the polyhedron. In Section 1.2.4, we defined the double description method as a representation containing both the V-representation and H-representation, defined in Section 1.2.3. By looking at the different faces in Figure 2.3, we can easily see the V-representation formed by all the faces consisting of vertices, but also the H-representation which is the set of inequalities of each face, in other words, the labels we give to each node of the face lattice.

We can then formally define the structure of the representation of polyhedra, and more specifically polygons as follows:

Definition 2.2.1 *A non-empty polyhedron P is represented by the tuple (S, T, s_0, ℓ_s) where:*

- *S is a finite set of states.*
- *$T \subseteq S \times S$ is a transition relation.*
- *$s_0 \in S$ is the initial state.*
- *$\ell_s : S \to 2^C$, where C is the set of constraints of P. The labeling function ℓ_s assigns labels to the states. The label consists in the set of equations that corresponds to the set of constraints of a given face, which is a set of inequalities. In other words, the label of a face corresponds to the set of equations that represent the set of points that belongs to the given face. The label of a face F of P, is then a subset $F \subseteq C$ of the set of constraints of P.*

We can notice in this definition that the tuple (S, T, s_0) is in fact the face lattice of the polyhedron. We then give a labeling function that allows us to label each state of the face lattice. This label, as explained before, corresponds to the subset of saturated constraints of the face it corresponds to. The initial state s_0 in the face lattice corresponds to the unique minimal element of the face lattice whose label consists in the saturation of all the inequalities defining the polyhedron.

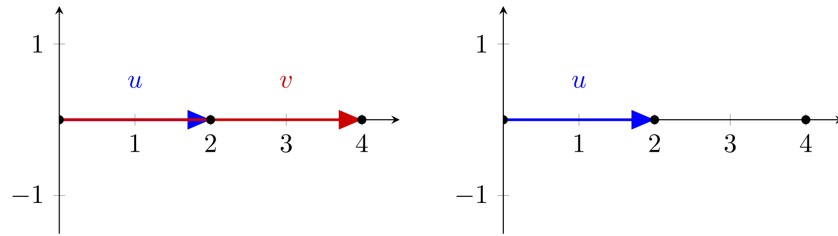
The definition of the structure of non-empty polyhedra will be the basis of the data structure we have created in this master's thesis.

2.3 Mesh Representation

Now that we have defined polyhedra representation, we need to explain how we can represent the other element of meshed polygons: meshes.

We defined meshes in Section 1.2.6. However, we do not have any canonical representation at the moment. As we aim to easily compare two meshes, a canonical representation is desirable. As defined in Definition 1.2.30, meshes are composed of two elements: an initial point, and a set of vectors called period. We first give a way to represent canonically a period, then define the canonical representation of any mesh with a canonical period.

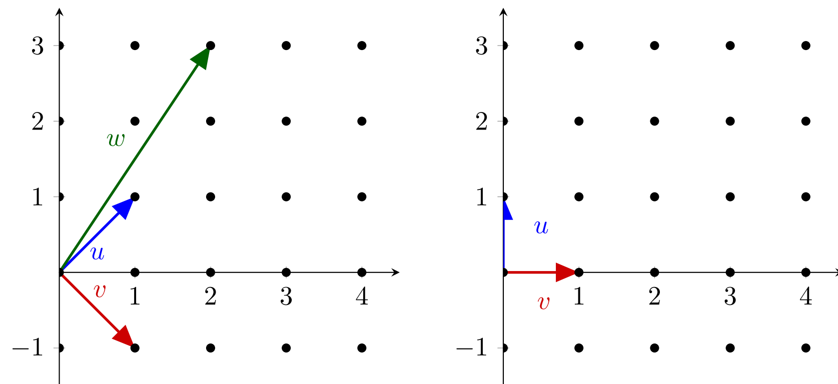
In order to find a canonical representation of a period, we first give in Figures 2.4 and 2.5 a set of examples of periods corresponding to the same set of points but that are represented with different vectors.



(a) Representation 1: initial point = $(0,0)$, and period = $(2,0),(4,0)$

(b) Representation 2: initial point = $(0,0)$, and period = $(2,0)$

Figure 2.4: Two representations of the mesh corresponding to the even numbers on the x axis.



(a) Representation 1: initial point = $(0,0)$, and period = $(1,-1),(1,1),(2,3)$

(b) Representation 2: initial point = $(0,0)$, and period = $(1,0),(0,1)$

Figure 2.5: Two representations of the mesh corresponding to the set of integers.

An easy way to modify a set of vectors without affecting their set of possible linear combinations is through the Gaussian elimination, explained in Theorem 1.1.30. The aim is then to redefine vectors as a subtraction of themselves by a combination of other vectors in the set. Performing the Gaussian elimination in the example given in Figure 2.5, we obtain:

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \\ 2 & 3 \end{pmatrix} L_2 \rightarrow L_2 - L_1$$
$$\begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 2 & 3 \end{pmatrix} L_3 \rightarrow L_3 - 2L_1$$
$$\begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 1 \end{pmatrix} L_2 \rightarrow L_2 + 2L_3$$
$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} L_1 \rightarrow L_1 - L_3$$
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

As we can notice, performing the Gaussian elimination on a set of vectors allows us to remove any redundant vectors but also to simplify the set of vectors.

The initial point of our definition of mesh can be any point of the mesh from which any other point of the mesh can be retrieved by adding some repetition of the period. One specific point of the mesh may be the origin. One may then want to define the initial point as the exact point where all the axis intersect. However, it is not always the case. When the mesh does not contain the origin, we use the offset as the initial point. The offset corresponds to the nearest point to the origin that is in the mesh. If the origin belongs to the mesh, then the offset is 0.

We eventually give a formal definition of the structure of a mesh that we use later when considering the data structure for meshed polygons.

Definition 2.3.1 *A* mesh $U = (s_0, V)$ *is a tuple where:*

- $s_0 \in \mathbb{Z}^n$ *is a point of the mesh, also called the initial point.*
- V *is an array of vectors* $\{\vec{v_0}, \dots, \vec{v_m}\}$ *where* $\vec{v_i} \in \mathbb{Z}^m$ *and* $m \in \mathbb{N}$*.*

The array V corresponds to the set of vectors representing the period between two points of the mesh. If the vector is empty, it means that the mesh is only constituted of one point. The empty mesh is represented as $\emptyset$.

2.4 Meshed Polyhedra

Convex meshed polyhedra[1] are defined as the new set resulting from the intersection of a mesh with a polyhedron. In other words, such a set is equal to the set of points of a mesh that satisfy the constraints of the given polyhedron.

[1] Beware that it is a misuse of language. Indeed, the set of a meshed polyhedron is not convex in general as it does not contain the segments between two points of the mesh. A convex meshed polyhedron is in fact a meshed polyhedron where its set of constraints is defined on a convex set.

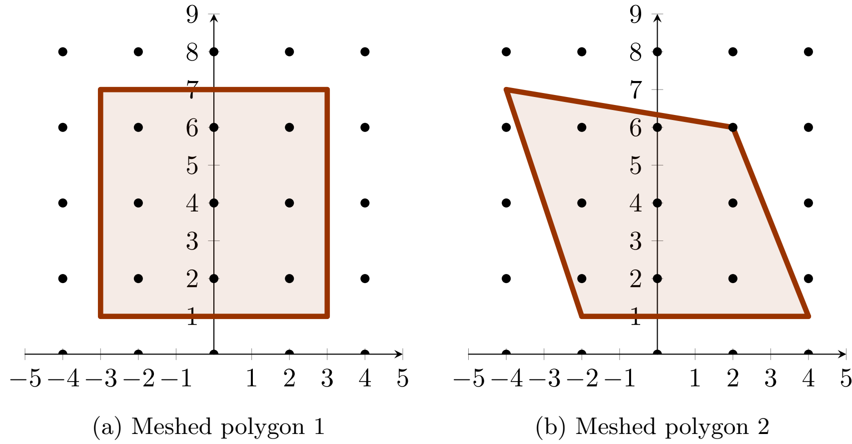
We have already given canonical representations of meshes and polyhedra. A way to represent a meshed polyhedron could be the intersection operation of the two sets, and therefore keeping and maintaining the two representations without interference from one to the other.

However, we found out that doing so may be inefficient when we have large sets for the polyhedron and the mesh. For example, performing the point membership operation on the data structure would require to first check that the point satisfies all the constraints of the polyhedron and then check the whole set of the mesh whether the point is in it or not. Our goal is to find a more efficient way to combine the two representations in one.

We decided to keep the principle of face lattice as the foundation of the data structure but we modified the labeling function in order to incorporate the concept of mesh in each face. Indeed each face does not have the same set of points in it therefore each face will keep the canonical representation of its own submesh as a label. By doing so, each label's face corresponds to the subset of constraints of its direct ascendants, but also to the submesh of its direct descendants.

Let us define exactly the notion of submesh of a face. Let M be a mesh. Then $M' \subseteq M$ is a submesh of M if the whole set of points belonging to M' also belongs to M. All the points of M do not compulsorily belong to M'. The submesh of a face $F = A\vec{x} \leq \vec{b}$ of the meshed polyhedron defined by the polyhedron P, and the mesh U, corresponds to the submesh of U where only the points belonging to the mesh U that are on the set of constraints $A\vec{x} = \vec{b}$ are retrievable. The period of a submesh then still corresponds to the distance between two points of the mesh with the only difference being that these points are only those that saturate the constraints of the face while satisfying the constraints of the transitions between the face and its direct ascendants. The initial point is still either the origin point or the nearest point of the mesh from the origin but the origin now corresponds to the exact point where the face intersects with one of its ascendants. The initial point consists then of the nearest point to the origin of the face that is in its submesh. Let us also precise that the empty mesh is by definition the submesh of any mesh.

However, this new representation is not canonical. As it can be seen in Figure 2.6, two meshed polyhedra representing the same set of points can be represented in several ways, even if the representation of the polyhedra and the meshes are unique.



(a) Meshed polygon 1 (b) Meshed polygon 2

Figure 2.6: 2 Meshed polygons containing the exact same set of points.

We then have to find a way to canonically represent this new structure. We can do that by rounding the polyhedron based on the points of the submesh of each of its faces. Intuitively, we can understand the rounding process with a metaphor. Let us consider the polyhedron as if it was a rubber band stretched, and the points of the mesh pins coming out of the plane. Rounding the meshed polygon then consists in un-stretching the polyhedron such that its new form passes through all the external points of the meshed polyhedron. In order to better understand the process, we give in Figure 2.7 the rounded form of the meshed polyhedra of Figure 2.6, proving that their new form is the same. The rounding operation then consists in adapting the constraints of the polyhedron of the meshed polyhedron to only have faces with non-empty meshes. The new meshed polyhedron then still consists in the same set of points but is under its canonical representation.

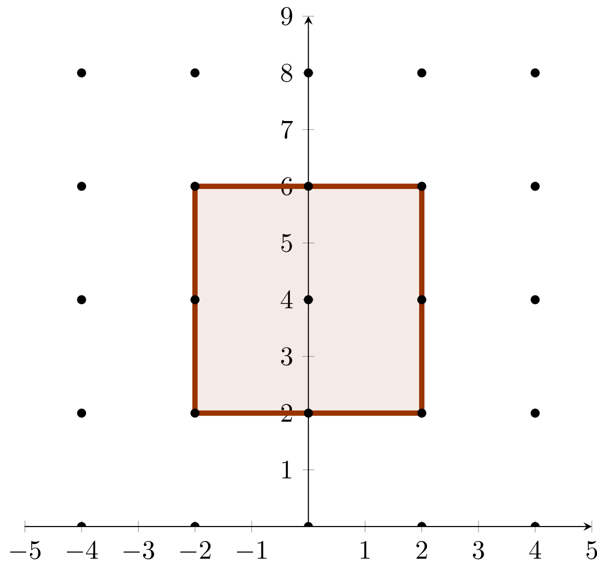
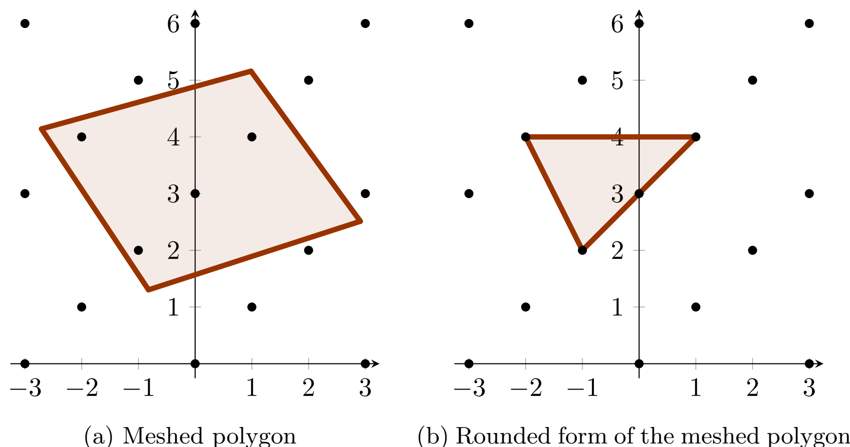


Figure 2.7: Rounded form of meshed polygon.

Meshes of meshed polygons may not be aligned with the axis of the coordinate system. Figure 2.8 shows such a meshed polygon as well as its canonical representation. We can also notice in this figure that the rounding process of a meshed polygon in order to retrieve its canonical representation may change the number of faces of the meshed polygon. The number of faces of the face lattice of the canonical representation of a meshed polygon is therefore not bound to the one of the same meshed polygon which is not in its canonical representation.



(a) Meshed polygon (b) Rounded form of the meshed polygon

Figure 2.8: Meshed polygon and its canonical representation.

2.4.1 Data Structure

Now that we have explained how we wish to represent meshed polyhedra, and more specifically meshed polygons, we are now ready to define the data structure we have created for discrete space polygons. This data structure that we call *Discrete Polygon Decision Diagram* (DPDD), includes the representation of the face lattice of the polygon on which each label's face contains the representation of the submesh of the current face as well as its subset of constraints. This data structure is defined as follows:

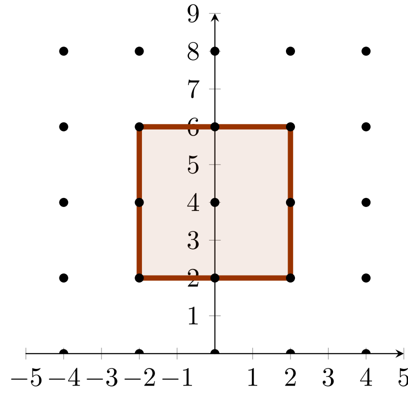
Definition 2.4.1 *A* Discrete Polygon Decision Diagram $D = (S, T, s_0, \ell_s)$ *representing the intersection between a non-empty polygon P, and a non-empty mesh U, is a tuple where:*

- *S is a finite set of states.*
- *$T \subseteq S \times S$ is a transition relation.*
- *$s_0 \in S$ is the initial state.*
- *$\ell_s : S \to 2^C \cup U'$, where C is the set of constraints of P, and $U' \subseteq U$, a submesh of the mesh U. The labeling function ℓ_s then assigns labels to the states as the set of constraints saturated as well as the representation of the submesh belonging to the saturated constraints. In other words, the label of a face F of P is then a subset $F \subseteq C$ of the set of constraints of P, and a subset of the mesh $U' \subseteq U$, and corresponds to the set of points belonging to the face of the meshed polygon.*

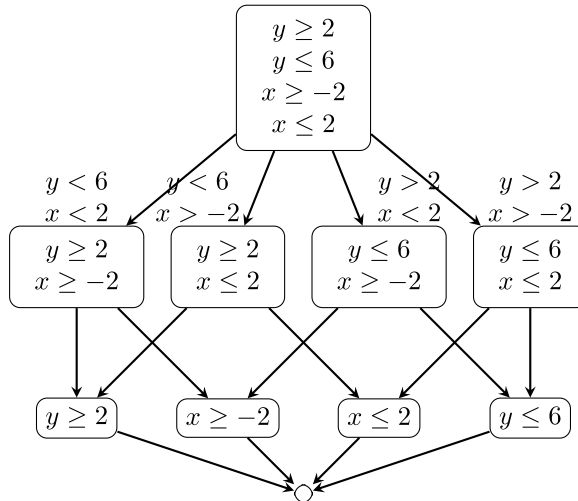
The tuple (S, T, s_0) corresponds to the face lattice of P with the mesh U. The state s is labelled by $\ell_s(s)$ corresponding to the subset of all the constraints of $\overline{P}$ (an affine subspace of P) that are saturated as well as U', the submesh of U which its whole set belongs to the saturated constraints. The initial state s_0 corresponds to the unique

minimal element of the face lattice of P, i.e., the saturation of all the constraints of P. The transition relation $T \subseteq S \times S$ corresponds to the partial order of the face lattice of P. The definition of the data structure does not apply when $P = \emptyset$ or $U = \emptyset$. Thus the notation $D_\emptyset$ denotes a special representation of $\emptyset$.

Figure 2.9 gives an example of a meshed polygon representation. As explained, the data structure is the face lattice of the polygon where each label of a face corresponds to the subconstraints and submesh of the face. We can also notice that labels of transitions do not contain any submesh. Indeed, as explained in Section 2.2, links correspond to transitions between two faces and correspond thus to the constraints that are saturated in the first face but not the second face while these constraints are still satisfied nonetheless. As the mesh of the first face is a submesh of the second face, and the transitions are directed, it has no sense to also label a transition with a mesh. A last thing to notice is that the submesh of a face can contain points that saturate the constraints belonging to the label of a transition leading to a face.



(a) Meshed polygon



(b) Structure of the meshed polygon

Figure 2.9: A meshed polygon and its representation.

2.5 Operations

In this section, we describe the operations that can be performed on *Discrete Polyhedron Decision Diagrams*. Each important operation is given under the form of an algorithm in pseudo code. The main operations that we cover in this section are the point membership, and the intersection operations. The latter one is presented in two versions: one that simply performs the intersection of two meshed polygons and another one that also converts the resulting meshed polygon into its canonical form, as defined in Section 2.4.

2.5.1 Point membership

The point membership operation consists in determining whether a given point x belongs to the set defined by the meshed polygon represented by a DPDD D.

Given a meshed polygon and a constraint, the *Point Membership* algorithm goes through the faces of the face lattice of the meshed polygon and checks whether the given point belongs to the set of saturated constraints of one of the faces and its mesh. In other words, the algorithm checks whether the point belongs to the label of a face. The algorithm passes from one face to another thanks to the transitions of the face lattice. If the point satisfies the set of constraints of the transition, the algorithm goes through the face at the other end of the transition. The algorithm then returns whether the point belongs to one of the faces of the face lattice.

Given a DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P with a mesh U, and x a point, the goal is to determine whether $x \in P \cap U$.

Algorithm 1 Point membership

```
 1: function PointMembership(DPDD D, Point x)
 2:     if ISEMPTY(D) then:
 3:         return False
 4:     s ← D.s0                    ▷Initialize s as the initial set of the structure given
 5:     while x does not saturate all constraints of D.ℓs(s) do:
 6:         s' ← {s' | (s, s') ∈ D.T}                      ▷Put in s' all descendants of s
 7:         found ← False
 8:         while not ISEMPTY(s') and not found do
 9:             if x satisfies all constraints of D.lT(s, s') then   ▷Check if point
    satisfies constraints of transition
10:                 s ← s'                                          ▷Move in next state
11:                 found ← True                                    ▷Found new states
12:             REMOVEFIRSTELEMENT(s')                           ▷Remove checked state
13:         if not found then:
14:             return False
15:     if x ∈ D.ℓs(s).Mesh then                          ▷Check point on current mesh
16:         return True
17:     return False
```

The main principle of this algorithm is to go through the states of the data structure from the initial state. We pass through a state only if the point satisfies the constraints labeling the transition between the state and its ascendant. For each state, we check whether the point saturates the constraints of the corresponding face,

contained in the label of the state. If the point saturates all the constraints of the face, then we check whether the point belongs to the set of points defined by the submesh of the face. If it is the case, the meshed polygon contains the point, otherwise the point does not belong to the mesh and therefore not to the meshed polygon. If no face containing the point membership has been found and there is no other state to go through, the algorithm states that the point does not belong to the meshed polygon.

Figure 2.10 shows an example of an execution of the point membership operation algorithm on a meshed polygon in DPDD.

In order to be more precise in the execution of the algorithm, here is a step by step execution of the algorithm for Figure 2.10.

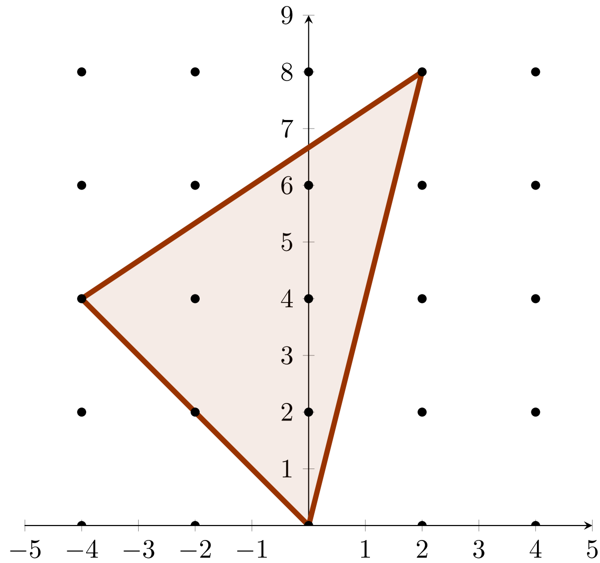
1. The point $(-2, 2)$ does not saturate the constraints of the initial state.
2. We retrieve the direct descendants of the initial state for which the point satisfies the label of the transition.
3. We go through one of the states retrieved. In this case, we selected the state $y \geq -x, y \geq 4x$.
4. The point does not saturate the constraints of the face.
5. We retrieve the direct descendants of the state for which the point satisfies the label of the transition.
6. The only state retrieved is the one with constraint $y \geq -x$.
7. The point saturates the constraints of the face.
8. We check whether the point is within the submesh.
9. The point $(-2, 2)$ belongs to the mesh $((0, 0), (-2, 2))$ of the face.

2.5.2 Intersection

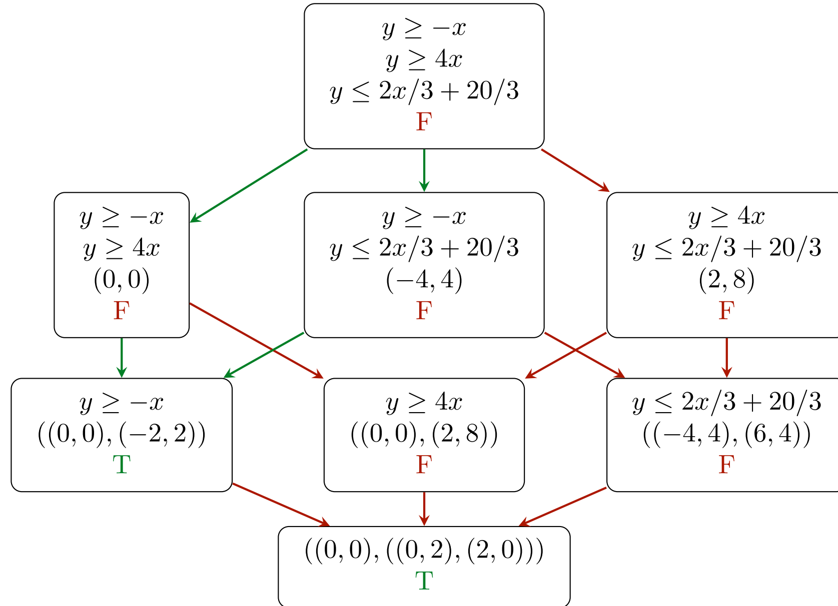
The intersection operation consists in constructing the meshed polygon formed by the intersection between a given meshed polygon and a given constraint. As a polygon is a set of constraints, we can also easily perform this intersection operation between a meshed polygon and a polygon. Indeed, this reduces to performing the intersection with each constraint, one by one. However, in this master's thesis, we do not address the problem of computing the intersection between a meshed polygon and a mesh. Therefore, performing the intersection between two meshed polygons that do not share the same mesh is not allowed by the algorithm described below.

The approach we follow to perform the intersection operation mainly relies on a coloring procedure that reflects how the faces are affected by the operation.

A color code of four different colors has been designed by I. Mainz[19]. As the intersection algorithm is significantly similar, we took the same approach. Therefore, each face is labeled by either the color Green, Blue, Yellow, or Red, depending on its position from the given constraint. Let then $\vec{a} \cdot \vec{x} \leq b$ be the constraint that intersects the meshed polygon. A face is Green if it is completely in the half-space created by $\vec{a} \cdot \vec{x} < b$. Blue faces are completely contained in $\vec{a} \cdot \vec{x} = b$. Faces that are labeled Red are "outside" of the intersection, meaning that they are completely contained in the half-space $\vec{a} \cdot \vec{x} > b$. Finally, a face that is labelled as Yellow has a non-empty intersection with both sides of the constraint.



(a) Meshed polygon



(b) Point membership over the DPDD of the meshed polygon

Figure 2.10: Point membership of the point $(-2, 2)$ over the polygon with the constraints $y \geq -x, y \geq 4x, y \leq 2x/3+20/3$ and the mesh $((0, 0), ((0, 2), (2, 0)))$.

Definition 2.5.1 *With respect to the constraint $\vec{a} \cdot \vec{x} \leq b$, a face F of the polygon P is colored thanks to the following rules:*

$$\begin{aligned}
\textit{Green} \quad & \text{if } \forall \vec{x} \in F : \vec{a} \cdot \vec{x} \leq b \wedge \exists \vec{x} \in F : \vec{a} \cdot \vec{x} < b, \\
\textit{Blue} \quad & \text{if } \forall \vec{x} \in F : \vec{a} \cdot \vec{x} = b, \\
\textit{Red} \quad & \text{if } \forall \vec{x} \in F : \vec{a} \cdot \vec{x} \geq b \wedge \exists \vec{x} \in F : \vec{a} \cdot \vec{x} > b, \\
\textit{Yellow} \quad & \text{if } \exists \vec{x} \in F : \vec{a} \cdot \vec{x} < b \wedge \exists \vec{x} \in F : \vec{a} \cdot \vec{x} > b.
\end{aligned}$$

As this list of colors is exhaustive, each face has a color. Based on the definition, the empty set is Blue. It is also possible to determine the color of a face based on the colors of its ascendants in the face lattice. Let us go through this property:

- If there is at least one direct ascendant of a face colored Yellow, then the face is also colored Yellow.
- If one of the direct ascendants of a face is colored Green, and another direct ascendant is colored Red, then the face is colored Yellow.
- If all direct ascendants of a face are colored Blue, then the face is also colored Blue.
- If at least one direct ascendant of a face is colored Green and all the other direct ascendants are either colored Green or Blue, then the face is colored Green
- If at least one direct ascendant of a face is colored Red and all the other direct ascendants are either colored Red or Blue, then the face is colored Red.

Figure 2.11 describes the coloring of a meshed polygon based on a given constraint. The data structure resulting from this meshed polygon then contains all the faces with the colors shown in the figure.

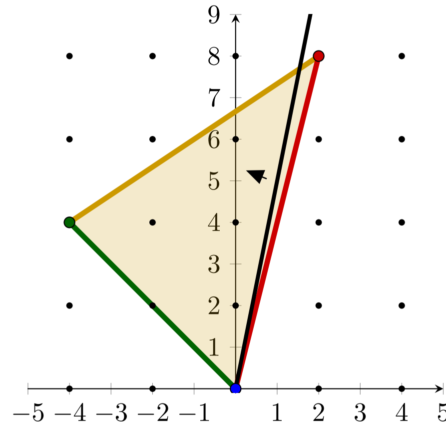


Figure 2.11: Coloring of a meshed polygon with a given constraint.

By giving a face of the face lattice of a meshed polygon, as well as a constraint to the *Coloring* algorithm, it colors the given face either in Blue, Green, Yellow, or Red. The color given to the face is determined by the set of colors of the direct ascendants of the given face, or by comparing directly the face with the constraint.

Given a face f from a DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P and a mesh U, and a constraint under the form $\vec{a} \cdot \vec{x} \leq b$, the aim is to color the face depending on which side of the constraint the face is.

Algorithm 2 Coloring faces in a face lattice

```
1:  procedure Color(Face F, Constraint a·x ≤ b)
2:      if F = ∅ then
3:          F.Color = Blue
4:      else if F is minimal then
5:          Compute F.Color by comparing the affine spaces F and {x | a·x ≤ b}
6:      else
7:          S ← set of colors of the direct ascendants of F
8:          if Number of direct ascendants = 1 then
9:              Compute ray r
10:             if ar < 0 then
11:                 S ← S ∪ {Green}
12:             else if ar = 0 then
13:                 S ← S ∪ {Blue}
14:             else
15:                 S ← S ∪ {Red}
16:         if Yellow ∈ S then
17:             F.Color = Yellow
18:         else if Green ∈ S and Red ∈ S then
19:             F.Color = Yellow
20:         else if S = { Blue } then
21:             F.Color = Blue
22:         else if S = { Green } or S = { Green, Blue } then
23:             F.Color = Green
24:         else if S = { Red } or S = { Red, Blue } then
25:             F.Color = Red
```

During the intersection operation, we also have to insert a new calculated mesh for each state created and those modified. Thanks to the properties of the representation, the mesh of a face is always a submesh of the meshes of the descendants of the face. It is then possible to easily calculate the unknown mesh of a face with only the direct successors of the face.

The *Insert Mesh* algorithm computes and sets the current mesh of the face of the face lattice given as input. To compute the mesh, the algorithm also needs the set of meshes from which the submesh of the face will be computed, as well as the set of constraints representing the transitions between the face and its direct ascendants. The algorithm then computes the canonical representation of the mesh of the given face by setting the initial point of the mesh, and then calculating the distance vector between this point and the nearest point belonging to the set of meshes and the set of constraints of the face.

Given a state s of the DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P and a mesh U, a set of meshes m that contains the set of meshes from which the mesh of s can be a subset, and t, the set of transitions going to the state s, this algorithm aims to calculate the mesh of the state s from the known set of meshes m and transitions t and insert it in the label of the face. The labels of the transitions, are of the form $\vec{a} \cdot \vec{x} \leq b$ and not $\vec{a} \cdot \vec{x} < b$ as a point saturating a transition belongs to the mesh of the face.

Algorithm 3 Calculate and insert mesh on a face of a meshed polygon

```
1: procedure InsertMesh(State s, Meshes m, Transitions t)
2:     if m = ∅ then ▷The succeeding meshes are all empty, the mesh of this face is then also empty
3:         ℓ_s(s').Mesh ← ∅
4:     firstPoint ← ∅
5:     if #t = 0 then                        ▷Retrieve a point that is on the mesh
6:         firstPoint ← p ∈ m
7:     else
8:         firstPoint ← p ∈ m | p ∈ ℓ_s(s).Constraints ∪ t
9:     ℓ_s(s).Mesh ← firstPoint      ▷Add the point to the mesh (empty if no point as been found)
10:    if firstPoint ≠ ∅ then  ▷Find another point on the mesh to calculate the periodicity
11:        secondPoint ← ∅
12:        if #t = 0 then
13:            secondPoint ← p ∈ m and p ≠ firstPoint
14:        else
15:            secondPoint ← p ∈ m | p ∈ ℓ_s(s).Constraints ∪ t
16:        if secondPoint ≠ ∅ then
17:            ℓ_s(s).Mesh ← ℓ_s(s).Mesh ∪ (secondPoint − firstPoint)    ▷Add the period to the state
```

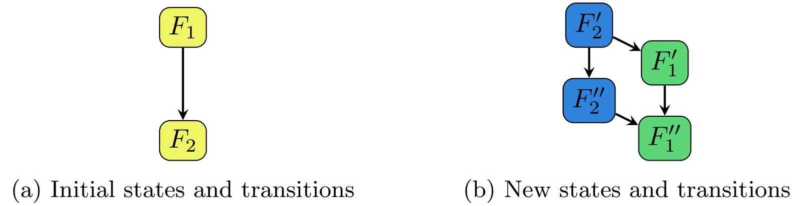
Now that it is possible to easily figure out which face remains or must be updated after an intersection operation. Moreover, any mesh can be calculated from its direct successors and the transitions from its predecessors. We can now explain how to perform an intersection between a meshed polygon and a constraint.

However, before doing so, let us see how faces are affected by an intersection operation, based on their color, computed by Algorithm 2.

- If a face is colored Blue, then the face is also a face of the meshed polygon computed by the intersection operation.
- If a face is colored Green, then the face is also a face of the new meshed polygon, and more precisely, a Blue face of this new meshed polygon.
- If a face is Red, then the constraint is a face of the new meshed polygon while the red face is not. The constraint will then be Blue face of the meshed polygon computed.
- If a face is Yellow, then two new faces for the new meshed polygon are created by truncating the face by the constraint.

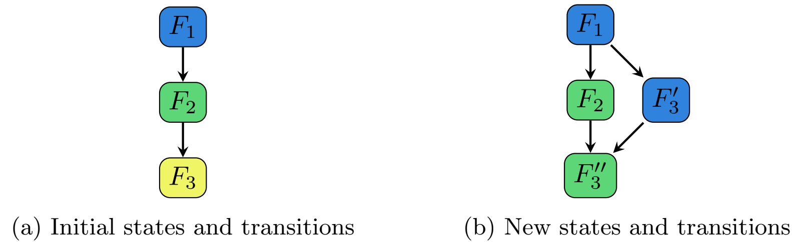
Now that we have discussed how faces are modified and created during the intersection operation based on their color, let us do the same for transitions:

- If a face is colored Blue, then all its transitions between the face and its ascendants are kept unchanged.
- If a face is colored Green, then all its transitions between the face and its ascendants are kept unchanged.
- If a face is colored Red, then none of its transitions is kept as the face is not within the meshed polygon.
- If a face is colored Yellow, then
  - if a direct ascendant of the face is also colored Yellow, then the two faces will create each two faces of the new meshed polygon. Figure 2.12 then shows the creation of the new transitions between the newly created faces.
  - if a direct ascendant of the face is colored Green which has a direct ascendant colored Blue, then only the Yellow face creates new faces. Figure 2.13 illustrates the creation of the new transitions between the new faces and the ascendants of the Yellow face.



(a) Initial states and transitions (b) New states and transitions

Figure 2.12: New transitions.



(a) Initial states and transitions (b) New states and transitions

Figure 2.13: New transitions.

The *Intersection* algorithm returns the meshed polygon retrieved by performing an intersection between a given meshed polygon and a given constraint. The algorithm first colors the faces of the meshed polygon thanks to the *Coloring* algorithm. Then, new faces are computed based on the colors of the different faces of the face lattice of the meshed polygon. After having retrieved all the faces of the new meshed polygon, the missing transitions are computed and added to the structure. Eventually, the mesh of each face is calculated thanks to the *Insert Mesh* algorithm.

Given a DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P and a mesh U, and a constraint under the form $\vec{a} \cdot \vec{x} \leq b$, the aim is to compute the new meshed polygon $P' = P \cap C$.

Algorithm 4 Intersection between a meshed polygon and a constraint

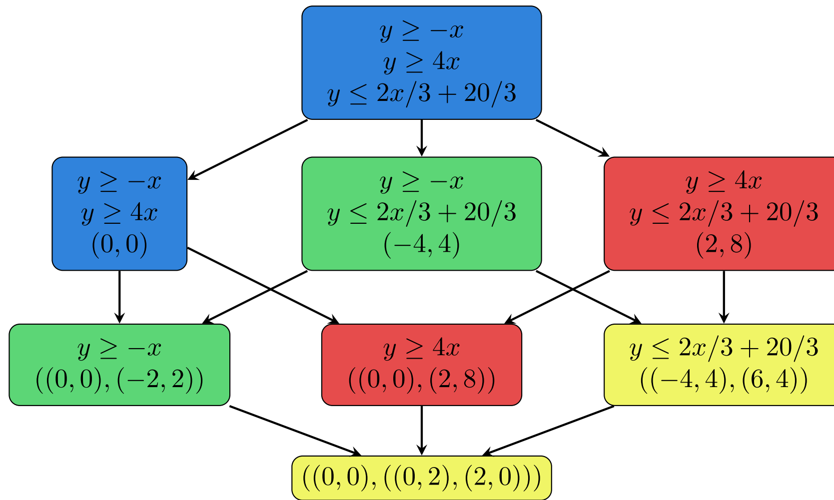
1: **function** INTERSECTIONCONSTRAINT(DPDD D, Constraint $\vec{a} \cdot \vec{x} \leq b$)
2: **if** ISEMPTY(D) or $\{\vec{x} \mid \vec{a} \cdot \vec{x} \leq b\} = \emptyset$ **then**
3: **return** $D_\emptyset$
4: COLOR(D, $\vec{a} \cdot \vec{x} \leq b$) ▷*Color the faces with the given constraint*
5: $S \leftarrow \emptyset$ ▷*Initialise new states*
6: $T \leftarrow \emptyset$ ▷*Initialise new transitions*
7: $NewS \leftarrow \emptyset$
8: **for all** $s \in D.S$ such that s.Color $\neq$ Red **do**
9: $S \leftarrow S \cup \{s\}$ ▷*Add the current state to the new states*
10: $\ell_s(s) \leftarrow D.\ell_s(s)$
11: **if** s.Color = Blue **then**
12: $\ell_s(s)$.Constraints $\leftarrow \ell_s(s)$.Constraints $\cup \{ax \leq b\}$ ▷*Add the new constraint in the label of the state*
13: **if** s.Color = Yellow **then**
14: **if** not ISEMPTY($\ell_s s$.Constraints) **then** ▷*The face that has no label contains all the mesh and does not need to be modified*
15: $newS \leftarrow newS \cup s$
16: $s' \leftarrow$ NEWSTATE() ▷*Create a new state*
17: $S \leftarrow S \cup \{s'\}$ ▷*Create the transition between the 2 states and add it*
18: $T \leftarrow T \cup \{(s', s)\}$
19: $\ell_s(s')$.Constraints $\leftarrow \ell_s(s)$.Constraints $\cup \{ax \leq b\}$ ▷*Add the label of s, as well as the new constraint*
20: $NewS \leftarrow NewS \cup s'$ ▷*Keep in mind all new states and the faces that have been modified to insert their mesh afterwards*
21: **if** $S = \emptyset$ **then** ▷*The intersection results to the empty meshed polygon*
22: **return** $D_\emptyset$
23: **for all** $s, s' \in S$ such that $(s, s') \in D.T$ **do**
24: $T \leftarrow T \cup \{(s, s')\}$
25: ADDNEWTRANSITIONS(D, S, T) ▷*Add the missing transitions between added states*
26: Sort($NewS$) ▷*Sort the states in NewS in reverse order of the depth in the face lattice*
27: **for all** $s \in NewS$ **do**
28: $mesh \leftarrow \emptyset$
29: $constraints \leftarrow \emptyset$
30: **for all** $s' \in S$ such that $(s, s') \in T$ **do**
31: $mesh \leftarrow mesh \cup \ell_s(s')$.Mesh
32: **for all** $s' \in S$ such that $(s', s) \in T$ **do**
33: $constraints \leftarrow constraints \cup \ell_T(s', s)$
34: INSERTMESH($s, mesh, constraints$) ▷*Calculate the mesh of the face s' and add it to $\ell_s(s')$*

Algorithm 5 Intersection between a meshed polygon and a constraint (cont)

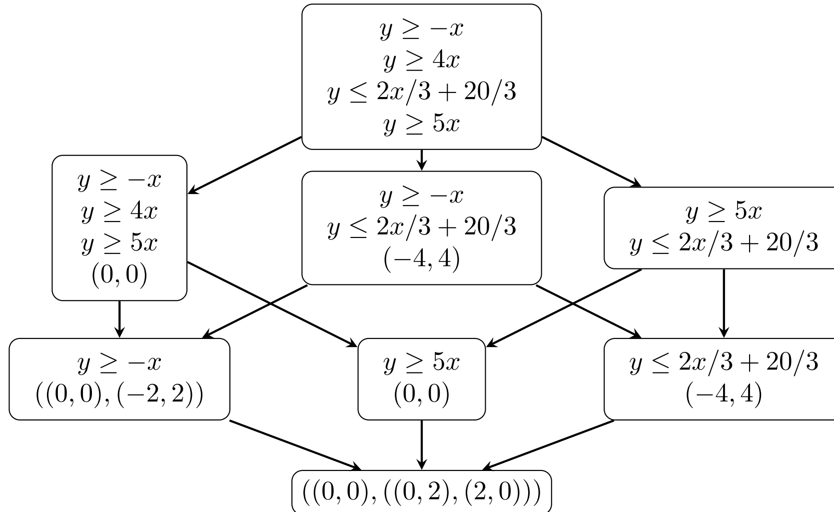
35: $\{s_1, ..., s_k\} \leftarrow \{s \in S \mid \nexists s' \in S : (s', s) \in T\}$ ▷ *Used to find new initial state*
36: **if** $k = 1$ **then** ▷ *Initial state already present in meshed polygon*
37: $\{s'_1, ..., s'_\ell\} \leftarrow \{s \in S \mid (s_1, s) \in T\}$
38: **if** $\ell = 1$ **then** ▷ *The current initial state has only one transition. The next state can be taken as new initial state*
39: $s_0 \leftarrow s'_1$
40: **else**
41: $s_0 \leftarrow s_1$
42: **else** ▷ *Have to create a new initial state*
43: $s_0 \leftarrow$ NewState()
44: $S \leftarrow S \cup \{s_0\}$
45: $T \leftarrow T \cup \{(s_0, s_1), ..., (s_0, s_k)\}$
46: $\ell_s(s_0)$.Constraints $\leftarrow$ $\ell_s(s_1)$.Constraints $\cup ... \cup$ $\ell_s(s_k)$.Constraints ▷ *Create the label of the new initial state (considering all constraints satisfied)*
47: DPDD $D' \leftarrow (S, T, s_0, \ell_s)$ ▷ *Create the new DPDD and allow to put sub-meshes in states*
48: **return** D'

Figure 2.14 provides an example of the intersection operation. This example is based on the meshed polygon and constraint of Figure 2.11.

The major drawback of this naive approach is that we do not fully exploit the DPDD data structure. Indeed, each face of the meshed polygon stores its own mesh. The main though behind this choice was to easily see if a point is contained in the mesh without having to perform each time an operation on the whole initial mesh. By performing the intersection algorithm, we could end up with multiple faces with empty meshes (possibly all of them). As Figure 2.14 illustrates, we would then lose the advantage of canonicity effect by our choice of data structure. In other words, performing the intersection algorithm does not ensure that, given a meshed polygon in its canonical representation, it returns another meshed polygon in its canonical representation. In order to do so we would need to round each face to contain at least a point of the mesh, as explained in Section 2.4. The Algorithm 4 does not cover this rounding operation of the meshed polygon. Another version of the intersection operation will be presented later in Section 2.5.6 in order to alleviate this drawback. Nonetheless, this algorithm would work if the DPDD structure stores a meshed polyhedron which has a higher dimension than a meshed polygon. This would not be the case however with the advanced version of the intersection.



(a) Structure before intersection



(b) Structure after intersection

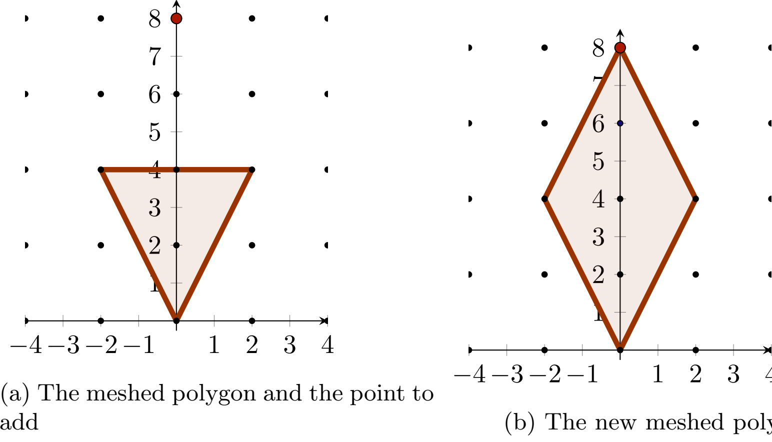
Figure 2.14: Intersection of a meshed polygon with a constraint.

2.5.3 Adding a point to a meshed polygon

While performing the rounding of a meshed polygon, we may need to add points belonging to the meshed polygon, which were removed during the operation. The operation to add a point then allows to add any point of the mesh into the meshed polygon, resulting to a new meshed polygon with all the points of the initial meshed polygon to which a new point has been added.

Adding a point into a meshed polygon amounts to create a new meshed polygon in which one of its vertices is the newly added point. The new meshed polygon results in the union of the initial meshed polygon and the new point as well as the edges created between the new point and each vertex of the initial meshed polygon. As a

meshed polygon is a convex set, several points may be added to the set while adding a single point. Figure 2.15 shows why and how it would happen.



(a) The meshed polygon and the point to add

(b) The new meshed polygon

Figure 2.15: Adding multiple points of the mesh while adding a given point.

In order to know which face will disappear or be modified during the operation of adding a point, we created another color code. This color code is only composed of two colors. Faces that are labelled as Red are faces that will disappear, meaning that it is an edge that only contains constraints that make the point to be added violates the point membership or a vertex that is between edges that are already labelled as Red. Faces that are **Black** are vertices that have at least an edge that is Red but not every direct successors are colored Red. That means that these faces have to be modified as the vertex will be used to create a new set of constraints by linking this vertex with the point that has to be added.

We have nonetheless two special cases to cover. The first one is when the given meshed polygon consists in an only vertex. As the operation looks for edges to see whether they are not compatible with the given constraint, and then going up the face lattice to mark vertices, we have to make a rule in the case there is no edge. As when adding a point to another point consists in creating an edge between the two vertices, we mark the vertex in **Black**. The other case to cover is when the meshed polygon is bounded and corresponds to a set of edges that does not form a plane. The easiest example of such a case can be represented as an edge whose constraint consists in $\vec{a} \cdot \vec{x} = b$, represented by the two inequalities $\vec{a} \cdot \vec{x} \leq b$ and $\vec{a} \cdot \vec{x} \geq b$, bounded by two vertices. By following what we have presented so far for this algorithm, the two vertices should be colored Red. However, if they are, we would not have any point to link to the point added to create the new edges. Therefore, if a vertex should be colored in Red but has only a direct descendant, the vertex is colored in **Black**.

Definition 2.5.2 *With respect to the set of constraints C, containing inequalities of the form $\vec{a} \cdot \vec{x} \leq b$, a face s of the DPDD D of a polygon P and mesh U is colored following these rules:*

$$
\begin{array}{rl}
\textit{Red} & \textit{if } s \textit{ is Edge and } \exists \vec{a} \cdot \vec{x} \leq b \in C \mid \vec{a} \cdot \vec{x} \leq b \in \ell_s s.Constraints \\
\textbf{Black} & \textit{if } \exists s' \in D.S \textit{ s.t. } (s,s') \in D.T \mid s'.Color = \textit{Red} \\
\textit{Red} & \textit{if } \forall s' \in D.S \textit{ s.t. } (s,s') \in D.T \mid s'.Color = \textit{Red} \\
\textbf{Black} & \textit{if } s \textit{ is Vertex and } \nexists (s,s') \in D.T \mid s' \in D.S \\
\textbf{Black} & \textit{if } s \textit{ is Vertex and } \forall s' \in D.S \textit{ s.t. } (s,s') \in D.T \mid s'.Color = \textit{Red} \textit{ and } \#s' = 1
\end{array}
$$

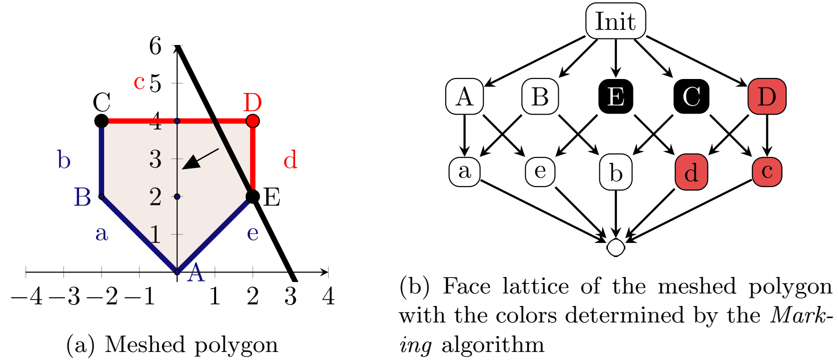
The *Mark Faces* algorithm sets the color to faces of the meshed polygon given as input. The colors are determined by the set of constraints given. Edges that intersect one of the constraints are marked in Red by the algorithm then set the color of the direct ascendants of the face either in **Black**, or Red. The latter color is given to a vertex face only if all its direct descendants are already colored in Red.

Given a DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P and a mesh U, and C, a set of constraints, the aim of this algorithm is to mark all the faces of D that represent an edge which intersects a constraint of C in Red. Furthermore, each face that has at least one direct successor colored in Red, is marked in **Black** and if all its direct descendants are colored in Red, the vertex is also colored in this color.

Algorithm 6 Mark faces of the meshed polygon depending on the constraints given

1: **procedure** MARKFACES(DPDD D, Constraints C)
2:   **if** not ISEMPTY(D) **then** ▷ *Cannot mark non-existing faces*
3:     **if** $\#D.S = 1$ **then**
4:       **if** s is Vertex $\mid s \in D.S$ **then** ▷ *There is only a vertex, it needs to be kept to create the constraints for adding a point*
5:         s/Color ← **Black**
6:     **for all** $s \in D.S \mid s$ is Edge **do**
7:       **if** $\exists c \in C \mid c \in \ell_s s$.Constraints **then** ▷ *The constraint is in the face of the edge*
8:         s.Color ← Red
9:         **for all** $s' \in S \mid (s', s) \in D.T$ **do** ▷ *Retrieve the predecessors of the edge*
10:           **if** $\forall succ \in D.S \mid (s', succ) \in D.T : succ$.Color = Red **then** ▷ *All the successors of the face are Red*
11:             **if** $\#succ = 1$ **then**
12:               s'.Color ← **Black**
13:             **else**
14:               s'.Color ← Red
15:           **else**
16:             s'.Color ← **Black**

Figure 2.16 illustrates the marking of a meshed polygon based on a constraint.



(a) Meshed polygon

(b) Face lattice of the meshed polygon with the colors determined by the *Marking* algorithm

Figure 2.16: Marking of a meshed polygon based on a constraint.

Thanks to this marking, we are now able to check to which faces the new point is linked and then create the new constraints of the meshed polygon.

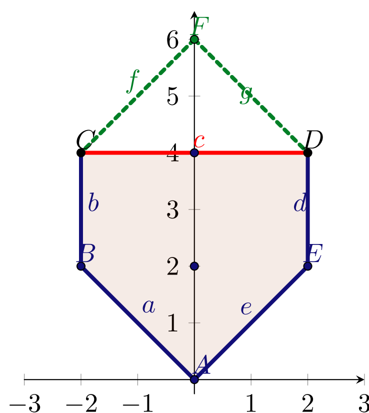
The *Add Point* algorithm would then add the point given as input to the given meshed polygon which has to be under its canonical representation. It would then returns the newly created meshed polygon which is also in its canonical representation as the point given belongs to the mesh and the initial meshed polygon is in its canonical representation. This ensures that each face created will have a non-empty mesh. The function uses the *Marking* operation to determine the faces that have to be modified or removed from the new meshed polygon before performing the intersection between the new constraints computed to add the point to the meshed polygon and the faces of the initial meshed polygon that have not been marked in Red. The constraints computed to add the point correspond to the set of constraints that represent the edge between the point and the vertices marked in **Black** during the marking phase of the algorithm.

Given a DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P and a mesh U, and x a point that is part of the mesh, the goal is to find a new DPDD D' such as the newly created meshed polygon is the minimal meshed polygon that contains all the points of the mesh of the initial meshed polygon and the new point.

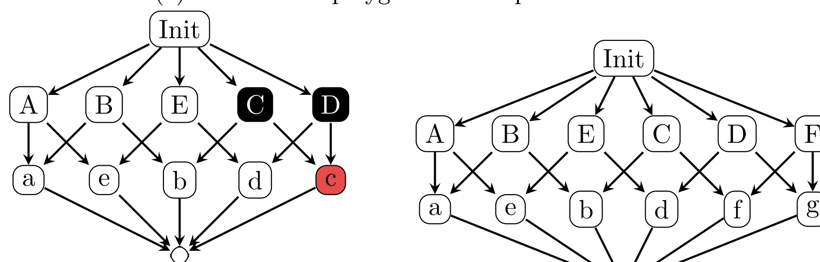
Algorithm 7 Add point of the mesh to a meshed polygon

1: **function** AddPoint(DPDD D, Point x)
2: **if** ISEMPTY(D) **then** ▷*If D is empty, the new DPDD is the point x*
3: $s_0 \leftarrow$ NewState()
4: $S \leftarrow s_0$
5: $T \leftarrow \emptyset$
6: $P \leftarrow$ CONSTRAINTS(x) ▷*The polyhedron is the point x only*
7: $U \leftarrow x$ ▷*Creates a mesh with only the point x in it and no period*
8: $\ell_s(s_0)$.Constraints $\leftarrow$ CONSTRAINTS(x) ▷*Add the constraints of x in the label of s_0*
9: $\ell_s(s_0)$.Mesh $\leftarrow U$ ▷*Add the new mesh in the label of s_0*
10: DPDD $D' \leftarrow (x, U, S, T, s_0, \ell_s)$
11: **else**
12: $s \leftarrow$ NEWSTATE() ▷*State that will retrieve the face that resulted to the negation of the point membership*
13: **if** POINTMEMBERSHIP(D, x, s) **then** ▷*The point is already in the polyhedron so no need to add it*
14: DPDD $D' \leftarrow D$
15: **else**
16: $c \leftarrow$ constraint $\in \ell_s(s)$.Constraints $\mid$ $\forall$constraint $\in \ell_s(s)$.Constraints : $x \notin$ constraint ▷*Check which constraints negated the point membership and only keep these ones*
17: MARKFACES(D, c)
18: $\ell \leftarrow \emptyset$
19: **for all** $s \in D.S \mid s$.Color $=$ **Black do** ▷*Retrieve all the faces marked in black*
20: $\ell \leftarrow \ell \cup$ CREATECONSTRAINTS(s, x) ▷*Create "link" between the point x and the faces by creating new constraints*
21: **for all** $s \in D.S \mid$ isEdge(s) and $\forall s \in D.S$ and $(s', s) \in D.T$: $\#(s', s) = 1$ **do** ▷*The edge has a ray as a predecessor*
22: $\ell \leftarrow \ell \cup$ CREATECONSTRAINTS(RETRIEVERAY(s), x) ▷*Create a constraint with the ray retrieved, starting from point x*
23: DPDD $D' \leftarrow NewDPDD(D.U)$
24: **for all** $constraint \in \ell$ **do** ▷*Do the intersection with the polyhedron face lattice and a new constraint*
25: $D' \leftarrow$ INTERSECTIONCONSTRAINT($D', constraint$)
26: **for all** $constraint \in \ell s \mid s \in D.S$ and not$s.Color$ $=$ red or black **do**
27: $D' \leftarrow$ INTERSECTIONCONSTRAINT($D', constraint$)
28: **return** D'

The principle of this algorithm is then to mark the faces to know which vertices of the meshed polygon have to be linked by edges in order to represent the new constraints of the new meshed polygons. Each vertex marked as **Black** is linked with the new point, creating a new edge and each face colored Red does not remain in the new meshed polygon. Figure 2.17 illustrates an example of a point added to a meshed polygon.



(a) The meshed polygon and the point to add

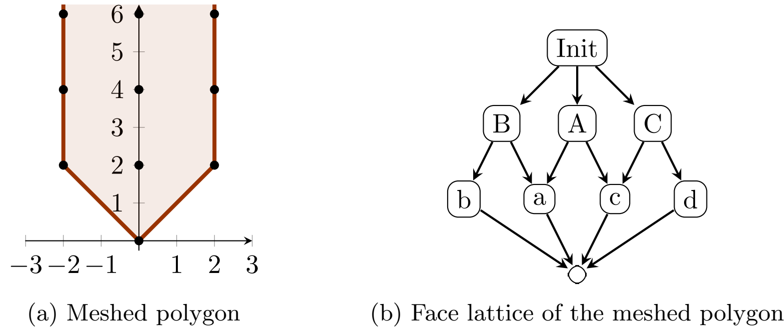


(b) The face lattice of the meshed polygon with the colors of the *Marking* algorithm

(c) The face lattice of the new meshed polygon after the point has been added

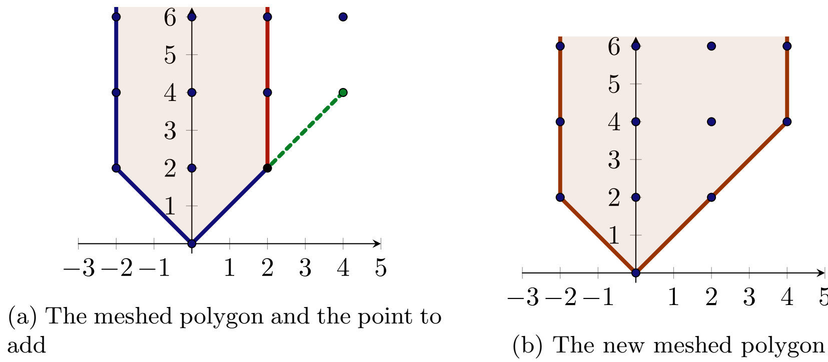
Figure 2.17: Add point operation on a meshed polygon.

When performing the operation to unbounded meshed polygons, some edges have only one direct ascendant in their face lattice, as shown in Figure 2.18. This happens, in particular, when they represent half-lines. In such a case, we can nonetheless represent a second ascendant as a vertex tending towards infinity. We can represent this point as the direction of the ray. When a point is added to a meshed polygon, if a vertex that represents a ray is marked in Red, then a new ray is created, coming from the new point and with the same direction as the ray marked in Red. Figure 2.19 gives an example of an unbounded meshed polygon to which a point is added.



(a) Meshed polygon (b) Face lattice of the meshed polygon

Figure 2.18: Unbounded meshed polygon.



(a) The meshed polygon and the point to add (b) The new meshed polygon

Figure 2.19: Add point operation on an unbounded meshed polygon.

2.5.4 Finding all the points of a mesh that satisfy a set of constraints

We study here the operation to retrieve all the points belonging to a given bounded meshed polygon. This operation corresponds to finding all the points of the mesh that satisfy the constraints of the bounded polygon. This operation only covers two-dimensional polytopes, i.e., bounded polygons, as it is not possible to retrieve all the points of an infinite set. We use this operation for the rounding operation of meshed polygons.

The *Find Points* algorithm returns the set of points that belong to the meshed polygon given as input. The meshed polygon has to be a meshed polytope. As the meshed polygon given is bounded, the set of retrieved points is finite. To retrieve all the points, the algorithm first creates a rectangle around the meshed polygon to bound the search of points. Once one of the vertices of the rectangle is on a point of the mesh, the algorithm uses this point as the initial point then computes all the points of the mesh within the rectangle. Each point found is then checked against the *Point Membership* algorithm to only retrieve the point belonging to the meshed polytope.

Given a DPDD $D = (S, T, s_0, \ell_s)$ representing a two-dimensional polytope P with a mesh U, the aim is to retrieve the list of all the points of the mesh that belong to the meshed polytope.

Algorithm 8 Finding all the points of the mesh strictly inside of a meshed polytope

1: **function** FINDPOINTS(DPDD D)
2: $p \leftarrow \emptyset$
3: **if** not ISEMPTY(D) **then**
4: $x \leftarrow \emptyset$
5: $y \leftarrow \emptyset$
6: **for all** $s \in D.S \mid s$ is vertex **do** ▷*Retrieve the min and max values of vertices to create a rectangle containing the polyhedron*
7: **if** $x = \emptyset$ **then**
8: $x[0] \leftarrow XCOORDINATE(s)$
9: $x[1] \leftarrow XCOORDINATE(s)$
10: **else**
11: $x[0] \leftarrow MIN(x[0], XCOORDINATE(s))$
12: $x[1] \leftarrow MAX(x[1], XCOORDINATE(s))$
13: **if** $y = \emptyset$ **then**
14: $y[0] \leftarrow YCOORDINATE(s)$
15: $y[1] \leftarrow YCOORDINATE(s)$
16: **else**
17: $y[0] \leftarrow MIN(y[0], YCOORDINATE(s))$
18: $y[1] \leftarrow MAX(y[1], YCOORDINATE(s))$
19: $coord[i] \leftarrow x[k], y[l] \mid 0 \leq i < 4$ and $0 \leq k, l < 2$ ▷*Retrieve the four vertices of the created rectangle*
20: $constraints \leftarrow$ CREATECONSTRAINTS(x, y) ▷*Create a constraint for each min and max x and y*
21: **while** POINTMEMBERSHIP($coord[i]$) $\mid 0 \leq i < 4$ **do** ▷*A vertex of the rectangle contains a point of the mesh*
22: Modify an arbitrary constraint to extend the rectangle and find new vertices
23: **for all** $point \in D.U \mid point = start +$ combination of period and $point \in constraints$ **do**
24: **if** $POINTMEMBERSHIP(point)$ **then**
25: $p \leftarrow p \cup point$
26: **return** p

2.5.5 Rounding a (half-)line

In order to reach a canonical representation of a meshed polygon, we have to round the meshed polygon to only have faces with non-empty meshes. The way it is usually done is explained in Section 2.5.6. However, the rounding of the faces may differ depending on the type of the face. Here, we cover the rounding operation of a (half-)line. Half-lines are present in unbounded meshed polygons as they are rays of these meshed polygons. If we have a (half-)line with an empty mesh, the rounding process we use can be related to the translation of the line to the nearest point of the meshed polygon. The translation of a line consists in shifting the line in a given

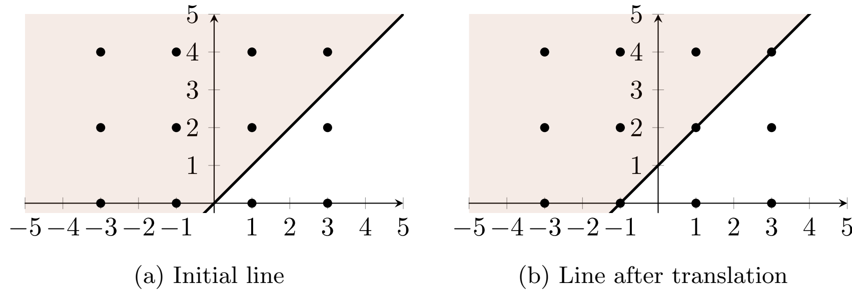
direction without changing its slope. All the points of the edge are then moved to the given destination defined by a given distance. Performing the translation on the line allows us to adapt the constraint of the edge to have a non-empty mesh in its label while keeping all the points of the initial meshed polygon within the structure.

To create the new constraint, we use Theorems 1.1.14, and 1.1.16. Based on these theorems, we can write the constraint forming the edge with the mesh of the meshed polygon, which is a basis of the meshed polygon. Let the constraint defining the edge $\vec{a} \cdot \vec{x} \leq b$, and $((v), ((\vec{u_1}), (\vec{u_2})))$, the mesh of the meshed polygon, then we have:

$$\vec{x} = k_1\vec{u_1} + k_2\vec{u_2} + v \mid k_1, k_2 \in \mathbb{Z}$$
$$k_1\vec{a} \cdot \vec{u_1} + k_2\vec{a} \cdot \vec{u_2} + \vec{a}v \leq b$$
$$k_1a'_1 + k_2a'_2 \leq b'$$
$$\Delta = b' - \left\lfloor \frac{b'}{gcd(a'_1, a'_2)} \right\rfloor gcd(a'_1, a'_2)$$

The new constraint of the (half-)line is then $\vec{a} \cdot \vec{x} \leq b - \Delta$.

Figure 2.20 illustrates the example of a line that has to be translated in order to pass through points of the mesh.



(a) Initial line (b) Line after translation

Figure 2.20: Translation of a line.

The operation of rounding a (half-)line then consists in finding the new constraint, which has the same slope as the constraint of the line, which is a translation of the line. The new line defined by the retrieved constraint passes through the nearest point belonging to the meshed polygon to the initial line. As this operation only consists in arithmetic computation, we do not provide any pseudo-code algorithm. However, such an algorithm would return the constraint of the translated line given as input. We would then be able to compute the canonical representation of the meshed polygon containing the (half-)line by performing an intersection between the meshed polygon and the newly computed constraint.

2.5.6 Advanced intersection

We can now cover the other way to perform the intersection operation that we call *Advanced Intersection*. The difference between this version and the previously shown one in Section 2.5.2 is that the advanced intersection always results in the minimal resulting meshed polygon, meaning that the meshed polygon has only faces with meshes that are non-empty, i.e., its canonical representation. The aim of such a representation is to take advantage of the data structure created for this master's thesis.

Before giving the algorithm, let us go through the different cases we may encounter when performing a rounding operation with the meshed polygon retrieved from the intersection of the canonical representation of a meshed polygon and a constraint.

- The first case we may encounter is the case of a meshed polytope. Rounding a meshed polytope consists in retrieving all the points belonging to the meshed polygon thanks to the Algorithm 8 and creating a new meshed polygon by adding each of them one by one thanks to Algorithm 7.
- The second case is when the meshed polygon is a (half-)line. In that case, we translate the line as shown in Section 2.5.5.
- The last case we may encounter is the case of an unbounded meshed polygon. The rounding operation for this case consists in dividing the meshed polygon into two meshed polygons including one which is a meshed polytope. The division into these two meshed polygons is done by creating a new edge between the two nearest points of the face that does not have any mesh. We then retrieve all the points inside of the meshed polytope with Algorithm 8 and add them to the other meshed polygon formed with Algorithm 7.

Now that the principle has been explained, let us define the algorithm for the intersection and rounding operation.

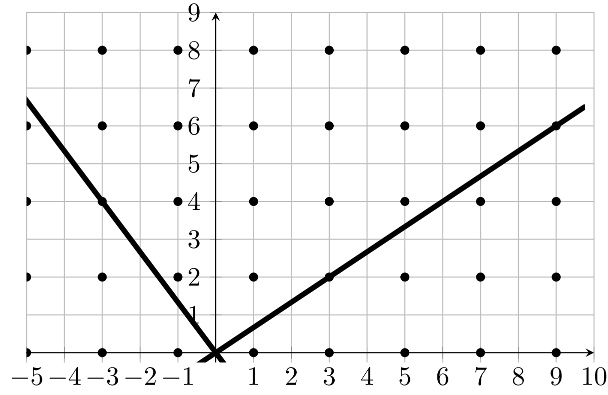
The *Advanced Intersection* algorithm takes as input a meshed polygon under its canonical representation, and a constraint. The algorithm's goal is to return the intersection between the given meshed polygon and the constraint in its canonical representation. It is done by first performing the *Intersection* algorithm on the given meshed polygon and the constraint. Once the intersection and the resulting meshed polygon have been computed and retrieved, the algorithm will go through all the faces of the face lattice. If one of the faces (but the initial state) has an empty mesh, then the algorithm rounds the meshed polygon to retrieve its canonical representation. The rounding operation is performed by adding all the points belonging to a meshed polygon one by one if the meshed polygon is a polytope. Otherwise, by creating new constraints, adapting the original constraints of the meshed polygon to constraints representing faces with non-empty meshes, and performing an intersection between the new constraints and the meshed polygon.

Given a DPDD $D = (S, T, s_0, \ell_s)$ representing a polygon P and a mesh U, and C, a constraint under the form $\vec{a} \cdot \vec{x} \leq b$, the aim is to compute the new meshed polygon $D' = D \cap C$ on which each face contains a non-empty mesh.

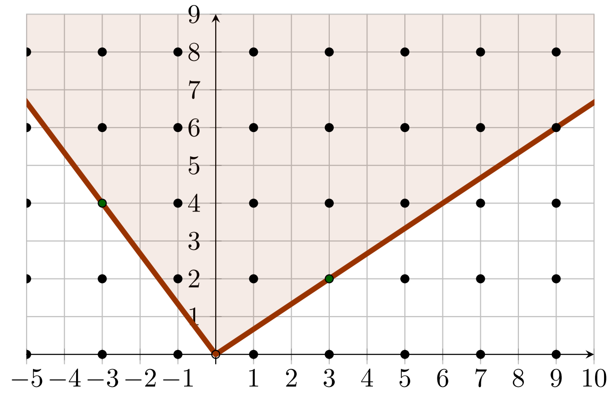
Algorithm 9 Advanced intersection between a meshed polygon and a constraint

1: **function** ADVANCEDINTERSECTIONCONSTRAINT(DPDD D, Constraint $\vec{a} \cdot \vec{x} \leq b$)
2: $D' \leftarrow$ INTERSECTIONCONSTRAINT($D, \vec{a} \cdot \vec{x} \leq b$) ▷*Perform the normal intersection*
3: **if** $\exists s \in D'.S \mid \ell_s(s).Mesh = \emptyset$ **then**
4: **if** D' is polytope **then** ▷*The polyhedron is a polytope, we can just find all the points inside it and add them to the polyhedron*
5: $p \leftarrow FindPoints(D')$
6: $D' \leftarrow AddPoint(D', p)$
7: **else**
8: **if** $\ell(s).Mesh = \emptyset \mid s \in D'.S$ and $isEdge(s)$ and $\forall pred \in D'.S$ and $(pred, s) \in D'.T : \#(pred, s) = 1$ **then** ▷*An edge that has a ray as a predecessor does not have any mesh*
9: $D' \leftarrow$ TRANSLATE(D', s) ▷*Translate the edge to have a mesh*
10: **else**Find nearest points to vertices with no mesh and create constraints between them
11: **for all** $s \mid s \in D.S$ and $\ell(s).Mesh = \emptyset$ **do** ▷*All the vertices that do not have any mesh*
12: $point1, point2 \leftarrow$ nearest points on the two nearest edges with mesh
13: $D'' \leftarrow$ INTERSECTIONCONSTRAINT(D',
14: CREATECONSTRAINTS($point1, point2$)) ▷*Create a polytope to get points that need to be added to the new polyhedron*
15: $p \leftarrow$ FINDPOINTS(D'')
16: $D' \leftarrow$ INTERSECTIONCONSTRAINT(D',
17: CREATECONSTRAINTS($point1, point2$)) ▷*Create the new polyhedron*
18: $D' \leftarrow$ ADDPOINT(D', p) ▷*Add the points that were missed by creating the new edge between the two nearest points*
19: **return** D'

Figures 2.21 and 2.22 illustrate an example of the advanced intersection with an unbound meshed polygon and a constraint.

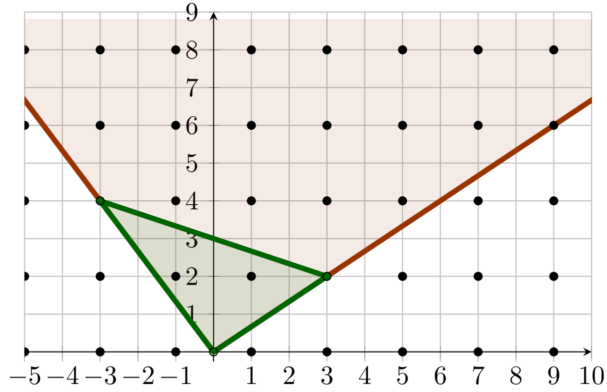


(a) Intersection between a meshed polygon and a constraint

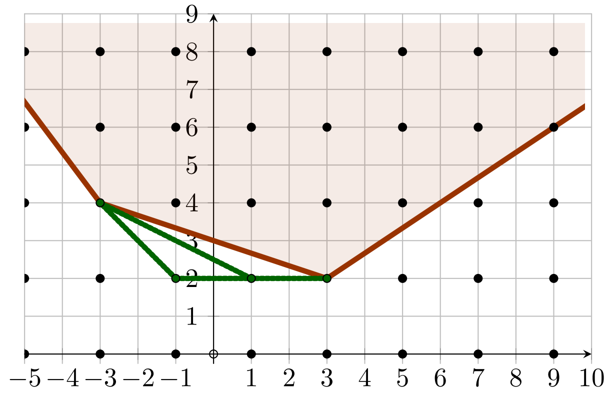


(b) Non-minimal meshed polygon resulting in the intersection operation

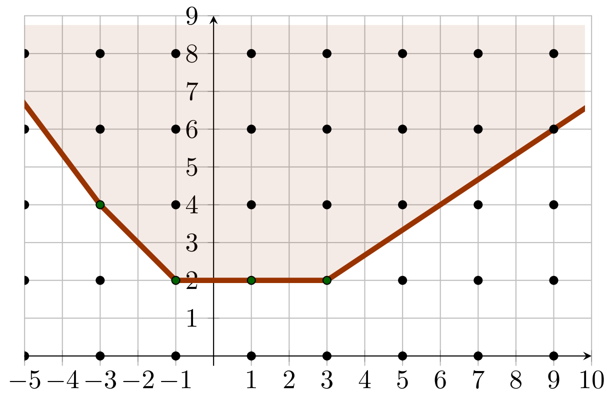
Figure 2.21: Intersection operation of the advanced intersection algorithm.



(a) Division of the meshed polygon into two meshed polygons



(b) Adding all the points found in the bounded meshed polygon to the unbounded meshed polygon



(c) Minimal meshed polygon of the intersection between the meshed polygon and the constraint

Figure 2.22: Rounding of the meshed polygon in advanced intersection.

Chapter 3

Implementation

In this master's thesis, we also implemented a prototype of the data structure and its operations in order to show the practical feasibility of our algorithms and definitions.

We decided to create this prototype in Python in order to have a quick working application but also because it provides a large library of linear algebra packages.

We mainly used Numpy to easily store arrays for our data structure but also operate on these lists, Matplotlib to have a visual representation of the meshed polygons created during the operations over our data structure, and the Sympy solver to find combinations of vectors, but also compute change of basis or create edges based on the vertices of the meshed polygon.

This section finally covers the implementations of the data structure and its operation in Python code. We explain how the algorithms have been implemented as well as the choices we made during this process. The prototype is available on the GitLab of the University of Liège for anyone to access.

3.1 Data Structure

We created 3 main classes for the data structure.

- State: this class represents the different faces of the face lattice of our data structure. Each state stores its own label as inequalities as well as its own submesh. We chose to separate the inequalities label and the mesh to have specific attributes for each and to easily update them without interfering with the other. Each state also has its type which can either be a *vertex*, an *edge*, a *plane*, or an *initial state* if the state corresponds to the cone created by homogenization of the polygon. These types will be used in order to know which part of the polygon they represent. Doing so will then help us to differentiate the different faces and possibly use that to our advantage for implementing some operations. The class *State* contains several methods and functions:
  - A function to check whether the set of constraints inside of the given face's label is saturated by a given point. This is done by *Sympy*'s *sympify* function that allows us to transform a given string into an equation in which the symbols can be replaced directly within the equation by the coordinates of the given point.
  - A function that checks whether a given point is within the mesh of the given face. This is done once again with the *Sympy* solver to find whether a diophantine equation of the mesh can be equal to the point.

  - A function that calculates the coordinates of a vertex face. This operation is done by finding the exact point where all the constraints of the vertex face intersect with each others.

- Transition: this class represents the different transitions between faces. We store the transition's label, which corresponds to a set of strict inequalities, but also the two faces that share the transition in the face lattice. As the face lattice is a directed graph, we make sure that the direction is respected by giving first the parent and then the child state. We create functions inside the class to easily retrieve the inequalities in the label of the transition as well as the states. Another function allows us to check whether a given point satisfies the strict inequality. This is done in the same way as the function to check whether a point saturates a constraint in the state class. To check the saturation of a point, we verify whether the point is directly on the constraint. We adapt this function to see whether the point is a possible solution to the strict inequality of the constraint. This function will be pretty useful when implementing the point membership operation in order to retrieve child faces that a given point might saturate.

- DPDD: this class represents the implementation of our DPDD data structure defined in Section 2.4.1. As labels are directly stored into the faces of the face lattice, the implementation of the data structure does not need a labeling function as input. To easily create an instance of the data structure, a new DPDD object requires the initial state, and the set of states and transitions. By default these sets are empty lists. We also store directly the initial given mesh as well as the set of constraints of the meshed polygon. We want the whole set of constraints and the mesh as input in order to easily verify that no constraints have been forgotten when creating faces and transitions, even if they are not required to perform the different operations.

The main drawback of our implementation is that we store labels, and therefore constraints, as string to be humanly readable to check the correctness of our operations. In order to not lose on precision and be able to perform correctly the operations, we needed to transform the rational numbers given by the solver as a fraction to store these numbers in the label. We do so through the *Fraction* function. However, to use this function, we need to already typecast the value into a string. Doing so already reduces the precision of the value and the retrieved fraction is not conclusive because of the error brought by the typecast. After testing, we figured out that giving a maximum value of 100000 to the denominator gave the best resulting fractions. To have a fully working implementation, we needed to avoid any imprecision, even if small, as we would not be able to perform correctly operations with the solver.

3.1.1 Representation

In order to represent meshed polygons, we decided to create Matplotlib figures. We take the vertices from the DPDD data structure and put them into the two-dimensional graph before linking them graphically. We represent the mesh by points directly on the graph. We found it easier to take the initial point of the initial mesh and then repeat a combination of the vectors representing the period of the mesh. By doing so, points of the mesh are visible on the figure even if they do not belong to the meshed polygon. As every point is reachable from an initial point through a combination of the vectors of the period, we set extremal values to the combination. Let p be the initial point of the mesh, and $\vec{u_1}, \ldots, \vec{u_n}$ the set of vectors of the period of the mesh, then the set of points retrieved corresponds to all the points p such as $p = v + k_1\vec{u_1} + \ldots + k_n\vec{u_n}$, where $\text{MIN} \leq k_1, \ldots, k_n \leq \text{MAX}$.

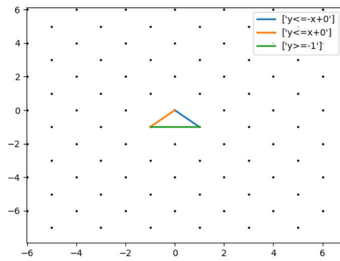


Figure 3.1: The representation of a meshed polytope

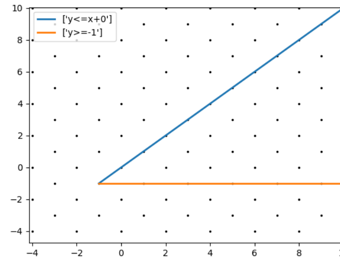


Figure 3.2: The representation of an unbounded meshed polygon

Figure 3.1 presents the Matplotlib figure representing a given meshed polytope. In case of unbounded meshed polygons, we should also need to retrieve the direction of the ray to represent a half-line in the figure. However, Matplotlib does not allow to easily represent half-lines. We then chose to set an arbitrary bound to the axis and to set a point at the intersection between the ray and the given bound. Figure 3.2 then represents a given unbounded meshed polygon.

The drawback of our representation is that it is not possible to visually distinguish a half-space from a (half-)line. Indeed, on the Matplotlib figure, both are represented as an edge. However, by looking at the face lattice of the DPDD data structure, or the constraints given in the figure, we can now easily compare the two meshed polygons and see the differences. When the meshed polygon consists of only a point, we set the color of the meshed polygon to another color than the points of the mesh in order to differentiate the given point from the mesh.

3.2 Operations

3.2.1 Point Membership

We performed a simple little modification to the algorithm of the *Point Membership* presented in Section 2.5.1. Now, the function does not only return whether the point given as input belongs to the meshed polygon, but also on which face of the face lattice of the meshed polygon the function was before ending. The state retrieved when performing the point membership will be used later in other functions of the prototype.

In addition to the point membership on a meshed polygon, we also created a version of the point membership that does not take into account the mesh of the meshed polygon. The function then returns whether a given point belongs to the set of constraints that determine the polygon of the meshed polygon.

We do not go any further in the explanation of these functions as they are only direct implementations of the algorithm given in Section 2.5.1.

3.2.2 Coloring

For the coloring process, we took advantage of the fact that we only work with polygons and that we store the types of the faces. We can then color first the nearest

faces to the initial state before coloring the direct descendants. We then repeat this procedure until coloring the last face of the meshed polygon. As the coloring process is based on the one explained in Section 2.5.2, we can use its properties to color faces based on the color of their direct ascendants. It is then the reason why we perform the coloring of faces in that order. The coloring process is adapted when we need to color a ray. As they are half-lines, we cannot use their (only) direct ascendant to decide a color. It then depends on the intersection of the ray with the given constraint and the direction of the ray.

Let us focus first on the case of a face that is a (half-)line. In order to know how to color the given face, we check whether the equation of the constraint and the one of the edge intersect each others. If it is the case, thanks to the *Sympy* solver, we can get the intersection point(s) between the two constraints. If there are several points as result, the face is colored Blue. If we did not find any intersection between the two equations, the edge is either fully within the given constraint or fully outside. To know which one it is, we just compare a point from each constraint by the symbol of the inequality of the given constraint. If we obtain a single point as intersection between the two constraints, we have several possibilities to color the face. The face is colored Yellow if the point belongs to the meshed polygon and that the last state visited during the *point membership* operation for the given point is the half-line. Otherwise, the face is colored in the same color as the color of the face of the vertex bounding the half-line.

Now that we can color faces that are (half-)lines, let us go through the function used to color any other face of the face lattice. In this function, we start by coloring each vertex. The coloration of vertices is pretty simple: we compute the point given by the face then check whether this point belongs to the equation of the constraint, or not. If the point is not on the constraint, then we determine the color by checking on which side of the constraint the point is. Once we have colored all the vertices, we can color all the edges whether by determining the color thanks to the color of the vertices bounding the edge, or thanks to the function explained before. Eventually for the last face, the plane of the polygon is colored only depending on the colors of the edges.

In order to easily check that the function correctly assigned the colors to the faces, we store the color directly in the structure of the face object. Each time the coloring function is used, we reset the color to avoid any conflict with previously assigned colors.

3.2.3 Insert Mesh

In order to calculate the mesh of a face, we take the set of meshes of its direct descendants as well as the set of constraints of the transitions going to the face. In this section, we go through the calculation of the submesh of a given face.

Let us first start with the easiest faces to compute their meshes: the vertices. Indeed, in order to find the mesh of a vertex, it is only required to check whether the point belongs to the set of submeshes of its direct descendants. If it is the case, its submesh is then equal to its own coordinates. Otherwise, the mesh of the vertex is empty.

The second easiest mesh to retrieve is the plane face. As the mesh is directly stored in our implementation of the DPDD data structure, we can simply use this mesh and just set the initial point to a point that is on the vertex face of the meshed polygon.

Eventually, let us talk about the edge face. An edge can either be a line segment, a half-line or just a line. We explain here how we operate for each type.

- When the edge is a line segment, we can easily find the initial point and the period for the canonical representation of the mesh:
  - If both vertices bounding the edge have a non-empty mesh, the coordinates of one of the vertices are used as the initial point of the mesh of the edge. The period can then be retrieved by computing the vector representing the distance between both vertices. However, there may be points of the mesh between the two vertices. We then need to divide the distance by the GCD of the combination of the vectors of the set of meshes given as input. Let $\vec{d}$ be the vector representing the distance between two points and $\vec{u_1}, \ldots, \vec{u_n}$ be the set of vectors of the set of meshes given as input. The vector $\vec{d}$ is a combination of $\vec{u_1}, \ldots, \vec{u_n}$: $\vec{d} = k_1\vec{u_1} + \ldots + k_n\vec{u_n}$, with $k_1, \ldots, k_n \in \mathbb{Z}$. Let g be the greatest common divisor of $k_1, \ldots, k_n$. Then, the period $\vec{p}$ of the mesh corresponds to: $\vec{p} = \frac{\vec{d}}{g}$.
  - If only one of the vertices bounding the edge has a non-empty mesh, the coordinates of this vertex are used as the initial point of the mesh of the edge. As the other vertex does not belong to the mesh, it cannot be used to create the mesh of the edge. We then first find another point of the mesh on the constraint of the edge. This point may actually not belong to the meshed polygon. We can then compute a distance vector, and then a possible period in the same way as previously mentioned. However, this period might not be the one we are looking for. Indeed, we need to compute two points of the mesh by adding once the period to the initial point and by subtracting once the period to the initial point. We then check whether one of the points belongs to the meshed polygon, and keep either the period or the negated period as the period of the mesh. It is however possible that neither of these points belong to the meshed polygon. Indeed, we could only have the initial point in the mesh of the edge. Therefore, if neither points belong to the meshed polygon, the mesh of the edge has an empty period.
  - If neither points have non-empty meshes, we use, as the initial point, the point of the mesh which is the nearest to one of the vertex, and that belongs to the meshed polygon while being on the constraint of the edge. In order to do so, we first find two arbitrary points of the mesh that are on the constraint of the edge. Once again, we compute the distance and then a period. This period and one of the points are then used to find the initial point. Indeed, we can retrieve any point of the mesh which is on the constraint of the edge with the current point and period. We then find the number of times we need to add the period to the point before reaching one of the vertices. Depending on where is the point based on the constraints of the transitions. We then need to round up or round down the number of times the period has been added to the point. Then, we do the same as when we have only a vertex that has a non-empty mesh to compute the period of the canonical representation of the mesh. Once again the mesh of an edge can be empty. We have to check that the nearest point belongs to the meshed polygon before using it as the initial point.

- When the edge is a half-line, we can actually operate in the same way as when we have a line segment with only one vertex with a non-empty mesh, or when we do not have any vertex with non-empty mesh.
- When the edge is a line, we compute two points of the mesh that are on the constraint of the edge and compute the period. We use as the initial point the nearest point, which is on the constraint of the edge, to the initial point given in the set of meshed given as input.

The following figures show the example of the computation of the mesh of an edge where no vertices bounding this edge have a non-empty mesh. The two examples represent two possible cases, depending on the first point found on the constraint. It then shows the importance of the transition as input of the algorithm.

Let us explain the steps performed on the figures. We first need to find a point on the constraint. This point is labeled p_0 on the figures. The distance is then computed by retrieving another point of the mesh on the constraint. The distance vector is labeled d in this example. Then, in Figure 3.4, as the point p_0 satisfies the constraint of the transition between the edge and the vertex point we are trying to reach, we round down the number of times the distance vector has been added to the point. The initial point of the mesh in this example is then the point labeled s_0. The period is the negation of the distance vector d. However, in Figure 3.5, as the point p_0 does not satisfy the constraint of the transition between the edge and the vertex point we are trying to reach, we round up the number of times the distance vector has been added to the point. The initial point is the same as in the previous example but the period is the distance vector this time.

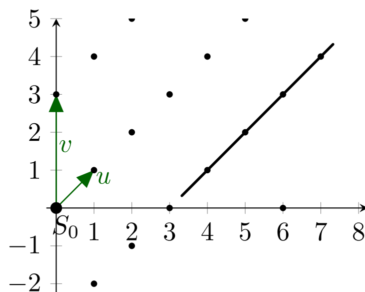


Figure 3.3: Edge on which the *Insert Mesh* function is performed.

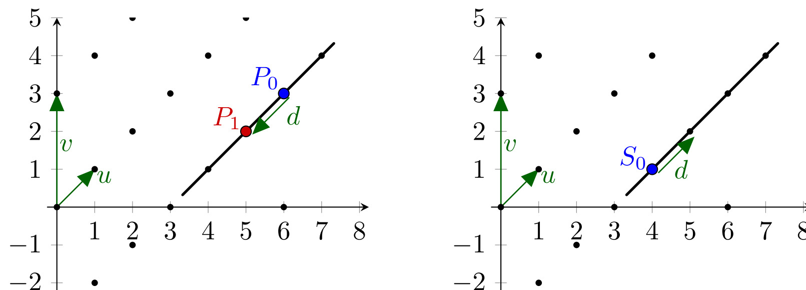


Figure 3.4: The *Insert Mesh* algorithm computing the mesh of the edge based on a point satisfying the transition.

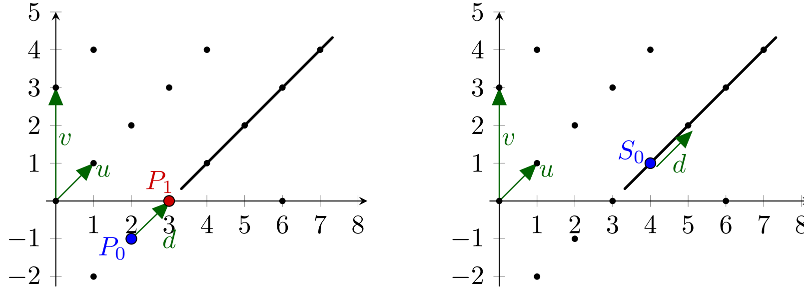


Figure 3.5: The *Insert Mesh* algorithm computing the mesh of the edge based on a point that does not satisfy the transition.

Thanks to these examples, we can also notice that the operation performed is in fact a change of basis to get the constraint as an axis and the point p_0 corresponds to the origin of the new basis.

All the meshes are computed and inserted from the plane state to the initial state. We compute them in that order to ensure that we have a set of mesh containing at least a submesh when the given face has a non-empty mesh.

3.2.4 Intersection

As we can now color the different faces based on a given constraint but also compute and insert meshes to each face, we use these functions to perform the intersection between a meshed polygon and a given constraint. We first color the faces depending on the constraint given to the *intersection* function. Then, by going through each face we easily retrieve the states that are still part of the new meshed polygon. Red faces are not kept while new faces are created when we have some Yellow faces. We use the properties stated in Section 2.5.2 in order to know what kind of face needs to be added. Labels of transitions are computed by retrieving the set of constraints that are in the parent face but not in the child face. After adding all faces, we check whether there already exists an initial state in the list of faces. If we have multiple vertices, a new face of type initial state has to be created. However, that is not the case if one or less vertex is present in the list of faces. In that case, we set the initial state of the data structure as the face that has the highest type.

Now that all the faces are created, we also need to create missing transitions between some faces. We can check that each transition is present by checking that faces whose label contains a subset of the constraints of a face of a direct higher type have a transition between them. We can therefore determine if all the transitions between initial state and vertices are present, as well as the transitions between vertices and edges, and edges and plane. If one is missing, we can retrieve the label of the missing transition by computing the subset of constraints that are not present in the label of a face but are in the face that has a direct higher type.

The last step to perform the intersection operation is to compute and insert the meshes of each face. We do that thanks to the *insert mesh* function explained in Section 3.2.3. We repeat the method for each face, and in the inverse hierarchical order, as we need the set of meshes of the direct descendants to compute the mesh of any face. As ascendants of faces may have been modified, and they impact the submesh

of their direct descendants through their transitions, it is needed to recompute the mesh of all the faces.

The *intersection* function then returns the new meshed polygon as an instance of the DPDD data structure.

3.2.5 Marking

To mark the faces depending on the set of constraints given, we will go through each edge face to check whether they contain one of the constraints given as input. If it is the case, the edge is colored Red and we set its direct ascendants in **Black**. When we are setting one of the faces as **Black**, we look at all its direct descendants. If they are all already colored in Red, then the vertex is also colored Red, unless the vertex has only one direct descendant.

We first reset all faces' color to prevent any conflict with previous iterations of the function or of the *coloring* function used for the *intersection* operation.

3.2.6 Add Point

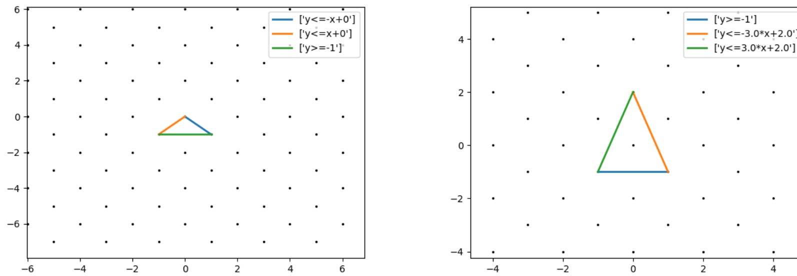
In order to add a new point to the meshed polygon, we first retrieve all the constraints that the given point does not satisfy. All these constraints that will not be present anymore in the new meshed polygon. These constraints are given as input to the *marking faces* function.

Once we have colored every faces, we retrieve only the vertices colored in **Black**. These vertices are kept in the new meshed polygon. We then compute the equation of the edge between the new point and the **Black** vertex. The newly created constraint, after having set the equation as an inequality, becomes a constraint of the new meshed polygon.

Once we have all the new constraints, we need to construct the new meshed polygon. To construct it, we retrieve all the constraints of the initial meshed polygon that were not used to mark its faces. The new meshed polygon then consists in the intersection of the mesh of the initial meshed polygon with the new constraints created in the previous step and the constraints of the initial meshed polygon that did not violate the point membership of the point to be added.

As the meshed polygon given needs to be in its canonical representation and as the given point needs to belong to the mesh of the initial meshed polygon, we do not need to round the resulting meshed polygon. The meshed polygon will already be in its canonical representation even if the *intersection* operation used is not the advanced one.

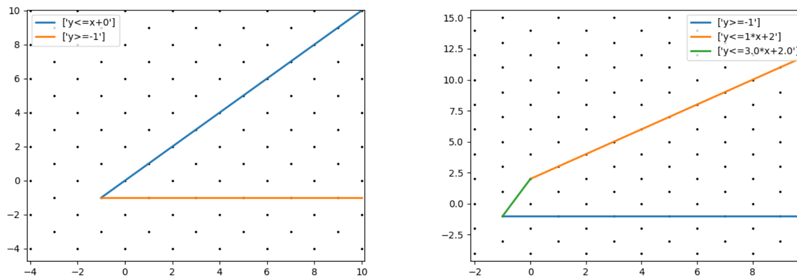
Figure 3.6 shows the example of a meshed polygon to which a point has been added. Figure 3.7 presents the example of a point added to an unbounded meshed polygon. Indeed, when a half-line is in the Red faces, a new constraint passing through the new point and with the same direction has to be created. As we can retrieve the direction of a half-line, we can easily compute this new constraint with the *Sympy* solver already used to compute every other new constraint.



(a) The representation of the initial meshed polytope.

(b) The representation after the point has been added.

Figure 3.6: Effect of the addition of the point (0,2) to the meshed polytope.



(a) The representation of the initial unbounded meshed polygon.

(b) The representation after the point has been added.

Figure 3.7: Effect of the addition of the point (0,2) to the unbounded meshed polygon.

3.2.7 Find Points

In order to find all the points within a given polytope, we create a rectangle surrounding the whole polygon. If one of the vertex belongs to the mesh of the meshed polygon, we can directly go to the second step. However, if it is not the case, we arbitrarily select an edge of the rectangle that we expand of one unit. We do so until one of the vertex is a point of the mesh.

Once we have this point, the second step is to use it to retrieve all the points of the mesh that are within the rectangle. This is validated by the definition of the mesh set, stating that any point of the mesh can be computed as a point of the mesh to which a combination of the vectors representing the period of the mesh is added.

The last step is then to retrieve the required points. We are able to retrieve all the points of the mesh within the rectangle created at the beginning of the method, by sequentially adding a combination of vectors from the vertex of the rectangle found that belongs to the mesh.

We then created a recursive function allowing to retrieve all these points by assigning at each iteration a different value to the combination variable of one of the vectors representing the period. However, all points of the mesh within the rectangle do not belong to the meshed polygon. We then have to check each point by using the *point membership* function explained in Section 3.2.1.

3.2.8 Translation

In order to reach a canonical representation of a meshed polygon, we have to round the meshed polygon to only have faces with non-empty meshes. When we work with unbounded meshed polygons which ray does not have non-empty mesh, the rounding operation consists in translating the ray to have the new edge passing through the nearest point of the meshed polygon to the initial ray.

In order to round a (half-)line we compute a new constraint corresponding to the translation of the given edge. To perform this operation, we have to compute the new value of b of the edge constraint $\vec{a} \cdot \vec{x} \leq b$. We do so thanks to the change of basis with the given mesh. We then retrieve the values of $\vec{a} \cdot \vec{x}$ and b. After, that, we re-write the left side of the equation as $k_1\vec{a} \cdot \vec{u_1} + \ldots + k_i\vec{a} \cdot \vec{u_i}$ and the right side as $b - \vec{a}v$, where $k_1, \ldots k_i \in \mathbb{Z}$, and v and $\vec{u_1}, \ldots, \vec{u_i}$ corresponds to the initial point and period of the mesh. Next, we just retrieve the GCD (g) of the left side elements and calculate the new value of b as $b - \left((b - \vec{a}v) - \left[\frac{b-\vec{a}v}{g}\right] g\right)$. As we need to round the value, we either choose to round down with the floor function or round up with the ceil function, depending on the sign of the constraint of the edge. The new b then replaces the initial value of the edge equation. The new equation corresponds to the new constraint of the translated edge. To see this modification, we just need to create an intersection with the new constraint and the meshed polygon.

3.2.9 Advanced Intersection

The *advanced intersection* operation consists in performing an intersection, between a meshed polygon in its canonical representation, and a constraint, and then rounding the resulting meshed polygon to retrieve the canonical representation of the resulting meshed polygon. Once the intersection has been performed, we retrieve all the vertices that do not have any non-empty mesh. We check, thanks to the canonical representation of a mesh and the function to *insert mesh*, if a vertex has a non-empty mesh. If we have vertices with empty mesh, we need to round the meshed polygon. For each vertex with empty mesh retrieved, we find the two nearest points on each side of the vertex belonging to the mesh that are on edges. Once the two points are retrieved, we create a new edge between the two. Doing so divides the meshed polygon in two. We can then add the points within the meshed polygon which still has the empty meshed vertex to the other meshed polygon to retrieve the canonical representation of the initial meshed polygon. This operation can be done if the meshed polygon is a meshed polytope or if the rays of the unbounded meshed polygon contain a non-empty mesh. If the meshed polygon is a (half-)line or if the unbounded meshed polygon has a ray with an empty mesh, then it is required to translate the (half-)line in order to retrieve the canonical representation of the meshed polygon. In the case of an unbounded meshed polygon, translating the ray to have a mesh does not ensure that the meshed polygon is now under its canonical form. Indeed as we can see in Figure 3.8, even after the translation of the ray, we can still have a vertex of the meshed polygon with an empty mesh. To avoid that, we perform once again

the *advanced intersection* operation to intersect the meshed polygon with the new constraint of its ray but also the rounding of the resulting meshed polygon.

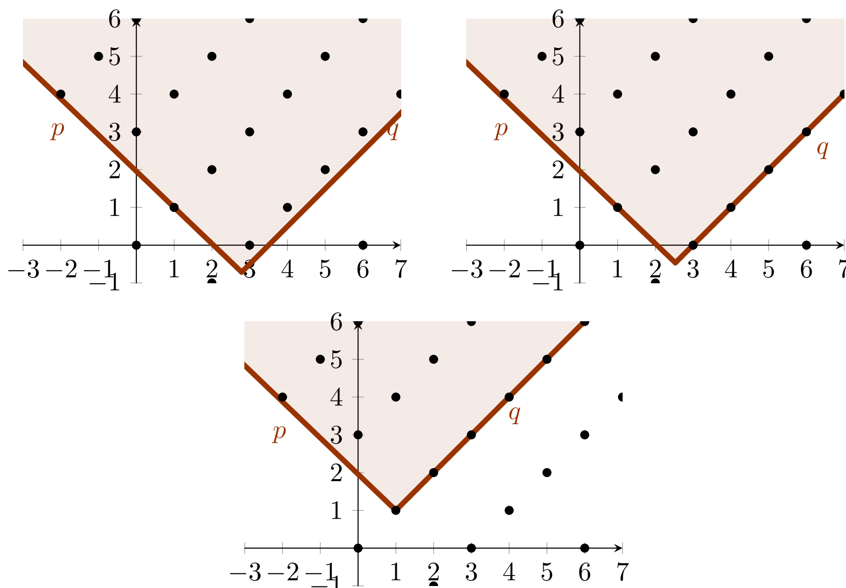


Figure 3.8: Rounding of a meshed polygon with a ray with an empty mesh.

Figures 3.9, 3.10, and 3.11 show examples of meshed polytopes on which the *advanced intersection* function has been performed. Figures 3.10, and 3.11 represent particular cases where the canonical representation of a meshed polygon results in an edge or a vertex.

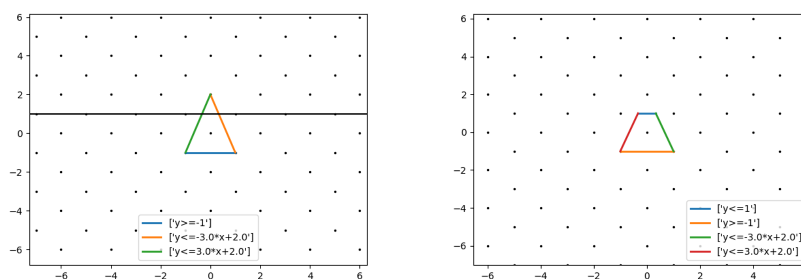


Figure 3.9: Advanced intersection on a meshed polytope.

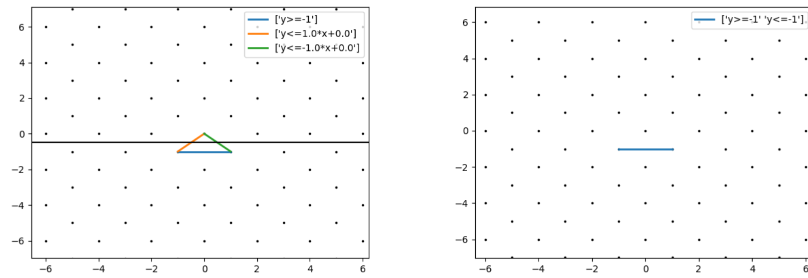


Figure 3.10: Advanced intersection resulting to a line segment.

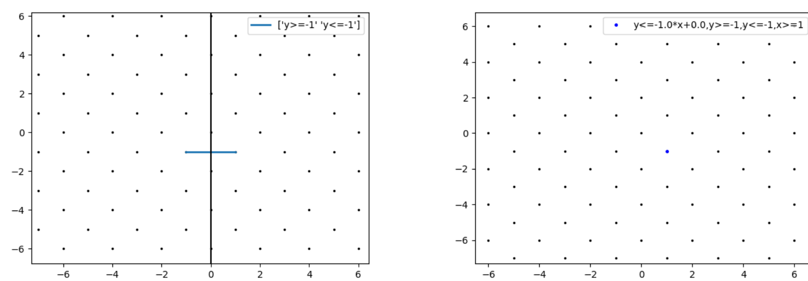


Figure 3.11: Advanced intersection resulting to an only vertex.

Chapter 4

Conclusions

This master's thesis introduced a new data structure called the *Discrete Polygon Decision Diagram* (DPDD), suited for representing meshed polygons. We provided algorithms in pseudo-code for all the major operations that can be performed on the data structure. This data structure is based on a representation of the face lattice of the polygon, in which a submesh is added to each face. Doing so allowed us to obtain more efficient algorithms.

Another contribution was to develop a Python prototype of the implementation of the data structure, as well as the manipulation operations, to prove the feasibility of the theoretical contributions this master's thesis brought.

The data structure does not discriminate on the type of meshed polygons. They can either be polytopes, i.e., bounded polyhedra, or unbounded polyhedra. The set of operations given in this master's thesis then fully works over meshed polytopes, or meshed polygons. Furthermore, we defined the canonical representation of meshes, polyhedra, and meshed polyhedra. The operations manipulating the meshed polygons output meshed polygons in their canonical representation, which simplifies comparison operations between meshed polygons.

The rounding process of polyhedra representing integer sets, which is essential to obtain a canonical representation of meshed polygons, has already been explored in the field of integer linear programming[22]. However, the needs for a symbolic representation of meshed polyhedra in optimization and computer-aided verification are different. The aim in optimization is to retrieve one solution to a given set of constraints. The best solution is based on the possible solutions defined by the meshed polyhedron, and the object functions that we want to either maximize, or minimize. In computer-aided verification, the goal is to represent every reachable state of the state-machine representing the program.

Generalizing the data structure and its operations to polyhedron defined in $\mathbb{R}^n$, for $n > 0$, instead of $\mathbb{R}^2$ would be the logical next step to this master's thesis. Doing so would not be trivial but the data structure can be kept as if, as it already allows to store meshed polyhedra of any dimension. The point membership operation also works with meshed polyhedra of any dimension. However, other operations would require to find more general properties to correctly round meshed polyhedra or add points to meshed polyhedra, as the algorithms given in this master's thesis take into account the type of each face to perform these operations.

In its current form, our data structure does not allow to compute intersections between meshed polygons and other meshed polygons or meshes. A quick improvement of this work would then be to find a way to compute efficiently this intersection.

Another way to improve this work would be to not only consider convex sets but also non-convex ones. Indeed, the set of reachable states of a system is generally a non-convex set. Another direction would be to extend the data structure to include non-linear constraints.

Bibliography

[1] Xavier Allamigeon, Ricardo D. Katz, and Pierre-Yves Strub. Formalizing the face lattice of polyhedra. *Logical Methods in Computer Science*, 18(2), 2022.

[2] Robert A. Beezer. *First course in linear algebra*, pages 1–44, 74–147, 257–339, 501–567. Robert Beezer, 2015.

[3] Hanspeter Bieri. *Nef polyhedra: A brief introduction.* Springer, 1995.

[4] Bernard Boigelot. Introduction to computer systems verification. University of Liège, Faculty of Applied Sciences, Montefiore Institute, 2022.

[5] Bernard Boigelot, Julien Brusten, and Jean-François Degbomont. Automata-based symbolic representations of polyhedra. In *Language and Automata Theory and Applications: 6th International Conference, LATA 2012, A Coruña, Spain, March 5-9, 2012. Proceedings 6*, pages 3–20. Springer, 2012.

[6] KC Border. Convex analysis and economic theory. California Institute of Technology, 2020.

[7] Stephen P. Boyd and Lieven Vandenberghe. *Convex optimization.* Cambridge University Press, 2021.

[8] Arne Brøndsted. *An introduction to convex polytopes.* Springer-Verlag, 1983.

[9] Karthik Chandrasekaran. Integer programming. University of Illinois Urbana-Champaign, 2021.

[10] Émilie Charlier. Mathématiques pour l'informatique 2. University of Liège, Faculty of Applied Sciences, Mathematics Institute, 2018.

[11] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* 01 2001.

[12] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[13] C.T.J Dodson and Tim Poston. *Tensor geometry: The geometric viewpoint and its uses*, page 18–56. Springer Berlin Heidelberg, 2013.

[14] W. Fenchel. *Convex cones, sets, and functions: From notes by D.W. Blackett of lectures at Princeton University, Spring Term, 1951.* Princeton University Department of Mathematics Logistics Research Project, 1953.

[15] Jean Gallier and Jocelyn Quaintance. Aspects of convex geometry polyhedra, linear programming, shellings, voronoi diagrams, delaunay triangulations. Department of Computer and Information Science, University of Pennsylvania.

[16] Branko Grünbaum. *Convex polytopes.* Springer, 2003.

[17] Bernd Gärtner and Emo Welzl. Geometry: Combinatorics & algorithms. Department of Computer Science, ETH Zürich, 2020.

[18] Rui-Juan Jing, Marc Moreno-Maza, and Delaram Talaashrafi. Complexity estimates for fourier-motzkin elimination. In *Computer Algebra in Scientific Computing: 22nd International Workshop, CASC 2020, Linz, Austria, September 14–18, 2020, Proceedings 22*, pages 282–306. Springer, 2020.

[19] I. Mainz. *Symbolic Representation of Convex Polyhedra in High-Dimensional Spaces*. PhD thesis, University of Liège, Faculty of Applied Sciences, Montefiore Institute, January 2020.

[20] Michel Rigo. Algèbre linéaire. University of Liège, Faculty of Applied Sciences, Mathematics Institute, 2010.

[21] Ralph Tyrrell Rockafellar. *Convex analysis*. Princeton University Press, 1970.

[22] Gerard Sierksma. *Linear and integer programming: Theory and practice*, pages 209–268. M. Dekker, 2002.

[23] John MacLaren Walsh. Optimization methods for engineering design: Polyhedral representation conversion. College of Engineering, Drexel University, 2014.