Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"
Corso di Laurea Magistrale in Matematica

Elaborato finale di Laurea Magistrale

Integer programming in the plane

Relatore: Prof. Michelangelo Conforti

Laureanda: Martina Gallato Matricola: 1179922

05 Luglio 2019
Anno Accademico 2018/2019

Contents

Introduction

Integer programming is the problem of maximizing a linear function over a set of integer vectors satisfying a set of linear constraints, namely:

$$\max\{c^t x : Ax \leq b, x \in \mathbb{Z}^n\},$$

with $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m\times n}, b \in \mathbb{Z}^m$.

Integer programming (IP) is a flourishing area of optimization, with applications that range from cryptography to production planning and telecommunication networks.

Although finite algorithms for bounded IPs were designed since the 50's and continuously improved upon, no algorithm is known for IP where the running time is bounded by a polynomial function of the encoding length of the problem (i.e. $m \times n \times \phi$, where $\phi = \log K$ and K is the largest entry in A, b, c). Indeed the problem belongs to the class of NP-complete problems for which no polynomial algorithm is known (and probably does not exist).

In this work we investigate structural and algorithmic questions in IP in fixed dimension, with a particular focus on the problem in dimension 2.

IP in fixed dimension arises when the dimension n (i.e. the number of variables) is fixed and it is accounted as a constant in the running time of an algorithm. IP in fixed dimension has been investigated for several decades and has many applications, mostly in computer science. The fundamental question is whether IP in fixed dimension can be solved efficiently, i.e. whether there is a polynomial algorithm. The answer is yes, however, the solution is highly non-trivial and uses several results in geometry of numbers.

The first one to prove polynomiality for IP in fixed dimension was Lenstra in 1983 (see [1], [2]). He showed in an elegant way that when n, the number of variables, is fixed, there is a polynomial algorithm to solve this problem. The idea behind his algorithm is to solve the problem recursively, splitting an n-dimensional problem into $f(n)$ $(n-1)$-dimensional subproblems, with $f(n)$ depending only on the dimension.

The fundamental theorem behind Lenstra's algorithm is Khinchine's Flatness Theorem. This theorem states that each n-dimensional polytope P containing no integer point must be thin in some integral direction c: that is, $\max\{c^t(x-y) : x, y, \in P\}$ is bounded by a function $f(n)$ which depends only on the dimension n.

A different proof for the polynomiality of IP in fixed dimension was given by Barvinok in 1994 (see [3], [4]). His approach is completely different from Lenstra's and it is based on the theory of generating functions.

The generating function of $P \cap \mathbb{Z}^n$ is defined as the Laurent series $g(P; z) = \sum_{\alpha \in P \cap \mathbb{Z}^n} z^\alpha$. Our aim is to identify it with a rational function, which will be called the *rational generating function* of $P \cap \mathbb{Z}^n$. Evaluating the rational generating function in $z = 1$ enables us to count the integer points in P quickly.

Barvinok proves that if the dimension n is fixed, there exists a polynomial algorithm for computing the rational generating function of a rational polyhedron $P \subseteq \mathbb{R}^n$. This result implies the existence of a polynomial algorithm for counting the number of integer points in a rational polyhedron $P \subseteq \mathbb{R}^n$. Therefore, using Barvinok's algorithm and binary search, one can solve an integer problem in fixed dimension in polynomial time.

IP in dimension 2 is linked to elementary algorithmic number theory. In particular, the problem of computing the greatest common divisor of 2 integers is a 2-dimensional IP (that clearly can be solved by the Euclidean Algorithm). Many results in IP in dimension 2 have their foundation in the theory of lattices and continued fractions.

IP in dimension 2 has been extensively studied and today many algorithms are known to solve an integer problem in dimension 2 in polynomial time. In this work we study the currently fastest algorithm which solves an integer problem in dimension 2, due to Eisenbrand, Rote and Laue (see [5] and [6]). The algorithm takes $O(m + \phi)$ arithmetic operations, where m is the number of constraints and ϕ is the maximum binary encoding length of the coefficients involved.

Chapter 1

Preliminaries

1.1 Euclidean Algorithm

The greatest common divisor of two given integral numbers a_0 and a_1 is $\max\{d \in \mathbb{N} : d|a_0,\ d|a_1\}$. The problem of finding the greatest common divisors of two integer numbers can be formulated as the following integer program in two variables:

$$\begin{aligned} \min xa_0 + ya_1 \\ s.t.\ xa_0 + ya_1 \geq 1 \\ x, y \in \mathbb{Z}. \end{aligned}$$

In other words, it holds that $\gcd(a_0, a_1) = \min\{xa_0 + ya_1 : x, y, \in \mathbb{Z}, xa_0 + ya_1 \geq 1\}$.

Indeed, let's call $m = \min\{xa_0 + ya_1 : x, y, \in \mathbb{Z}, xa_0 + ya_1 \geq 1\}$. If $d \mid a_0$ and $d \mid a_1$ then, $d \mid xa_0 + ya_1$ for every $x, y \in \mathbb{Z}$, hence $d \mid m$. So we have $\gcd(a_0, a_1) \leq m$. If $\gcd(a_0, a_1) < m$, then m is not a common divisor of a_0 and a_1. Therefore we can assume that $m \nmid a_0$. Then we can write $a_0 = qm + r$ with $q, r \in \mathbb{Z}$ and $1 \leq r < m$. If we call $\bar{x}$ and $\bar{y}$ the integers such that $m = \bar{x}a_0 + \bar{y}a_1$, we can write $r = a_0 - qm = a_0 - q(\bar{x}a_0 + \bar{y}a_1) = (1 - q\bar{x})a_0 - q\bar{y}a_1$, namely $r \in \{xa_0 + ya_1 : x, y, \in \mathbb{Z}, xa_0 + ya_1 \geq 1\}$. But this is a contradiction because $r < m$. Hence, we have $m = \gcd(a_0, a_1)$.

However, the greatest common divisor of two integer numbers can be computed also using the *Euclidean Algorithm* (*EA*).

Without loss of generality we can assume that a_0, a_1 are positive integers, since $\gcd(a_0, a_1) = \gcd(|a_0|, |a_1|)$. Anyway, the EA works correctly also with arbitrary integers, although it may return $-\gcd(a_0, a_1)$. We suppose that $a_0 > a_1$ (otherwise we switch them). The first iteration of the *EA* computes two integers q_0 and a_2 such that $a_0 = a_1 q_0 + a_2$ with $0 \leq a_2 < a_1$ and $q_0 \geq 1$. The i-th iteration computes q_{i-1} and a_{i+1} such that $a_{i-1} = a_i q_{i-1} + a_{i+1}$. We stop at k such that $a_{k+1} = 0$. Then, the greatest common divisor of a_0 and a_1 is a_k.

Indeed, it holds that $\gcd(a_0, a_1) = \gcd(a_1, a_2) = \gcd(a_{i-1}, a_i)\ \forall\, 1 \leq i \leq k$. This is true because the equality $a_0 = q_0 a_1 + a_2$ (where $q_0 \in \mathbb{Z}$) implies that the set of common divisors of a_0 and a_1 is exactly the set of common divisors of a_1 and a_2, and so on $\forall\, i \leq k$.

The running time of the *EA* is $O(\phi)$ where ϕ is the binary encoding length of a_0. Indeed, $a_0 = q_0 a_1 + a_2 > q_0 a_2 + a_2 \geq 2a_2 \Longrightarrow a_2 < a_0/2$.

We will be interested in knowing the two integers $x, y \in \mathbb{Z}$ such that $a_0 x + a_1 y = \gcd(a_0, a_1)$. In order to find them we need the *Extended Euclidean Algorithm*. We define

$$M^{(-1)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and then,

$$M^{(j)} = \begin{pmatrix} q_0 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_j & 1 \\ 1 & 0 \end{pmatrix} \qquad \text{for } 0 \leq j \leq k-1,$$

where the q_j are such that $a_j = a_{j+1} q_j + a_{j+2}$ for $0 \leq j \leq k-1$.

It holds that

$$M^{(j)} \begin{pmatrix} a_{j+1} \\ a_{j+2} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.$$

In particular, for $j = k-1$, it holds that

$$M^{(k-1)} \begin{pmatrix} a_k \\ a_{k+1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.$$

The matrix $M^{(k-1)}$ is nonsingular because $\det \begin{pmatrix} q_j & 1 \\ 1 & 0 \end{pmatrix} = -1$ for each j. Therefore we have:

$$\begin{pmatrix} a_k \\ a_{k+1} \end{pmatrix} = (M^{(k-1)})^{-1} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.$$

This implies that the coefficients in the first row of $(M^{(k-1)})^{-1}$ are x, y such that $a_0 x + a_1 y = \gcd(a_0, a_1)$. Also, since $\det(M^{(k-1)}) = (-1)^k$, x and y are integer numbers.

These two integers can be used, for instance, to find an integer point in a line. Suppose we have a line described by the equation $\{(x_1, x_2) \in \mathbb{R}^2 :$

$a_0x_1 + a_1x_2 = b\}$, where $a_0, a_1 \in \mathbb{Z}$ and $\gcd(a_0, a_1) = 1$. This assumption can be made without loss of generality since it is possible to put a general inequality in this form through a gcd-computing and a constant number of arithmetic operations.

Since $\gcd(a_0, a_1) = 1$, we know that there exist two numbers $x, y \in \mathbb{Z}$ such that $a_0x + a_1y = 1$. So, if we take $\hat{x} = bx, \hat{y} = by$ we get that $a_0\hat{x} + a_1\hat{y} = b$; therefore the point $(\hat{x}, \hat{y})$ belongs to the line. Then it is clear that for each $k \in \mathbb{Z}$ the point $(\hat{x} - ka_1, \hat{y} + ka_0)$ belongs to the line too.

1.2 Continued Fractions

A *continued fraction* is defined as

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \dots}},$$

where the $a_i \in \mathbb{R}\ \forall i \geq 0$. We will deal with terminating continued fractions with natural elements, that is to say:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{\ddots}{a_{n-1} + \cfrac{1}{a_n}}}}$$

with $a_0 \in \mathbb{Z}$ and $a_1, \dots, a_n \in \mathbb{N}_{>0}$. We use the notation $a = [a_0, \dots, a_n]$. We call fractions with $a_n > 1$ and the continued fraction $a = [1]$ *standard* continued fractions. There is a one-to-one correspondence between standard continued fractions and rational numbers.

Theorem 1.1. *To every rational number α there corresponds a unique standard continue fraction (whose value is α).*

Moreover, we can use the Euclidean Algorithm applied to a_0 and a_1 to find the continued fraction representing $\frac{a_0}{a_1}$. Indeed, the following proposition holds.

Proposition 1.2. *Let $a_0, a_1 \in \mathbb{Z}$, $a_1 > 0$. Let $q_0, \dots, q_{k-1}$ be the sequence of quotients generated by the application of the Euclidean Algorithm to a_0 and a_1. Then $\frac{a_0}{a_1} = [q_0, \dots, q_{k-1}]$.*

Proof. As a first step we prove that for each $j = 0, \dots, k-1$ it holds that $a_0/a_1 = [q_0, \dots, q_{j-1}, q_j + a_{j+2}/a_{j+1}]$. We observe that this is not necessarily a continued fraction because $q_j + a_{j+2}/a_{j+1}$ may not be integer. We prove this property by induction. For $j = 0$ we have that $a_0/a_1 = q_0 + a_2/a_1 = [q_0 + a_2/a_1]$, so the relation is true. Now let's assume it is true for $j-1$. We have $a_j/a_{j+1} = q_j + a_{j+2}/a_{j+1}$, so $a_0/a_1 = [q_0, q_1, \dots, q_{j-2}, q_{j-1} + a_{j+1}/a_j] = [q_0, q_1, \dots, q_{j-2}, q_{j-1}, a_j/a_{j+1}] = [q_0, \dots, q_{j-2}, q_{j-1}, q_j + a_{j+2}/a_{j+1}]$. Finally, taking $j = k-1$ we get the result. □

Definition 1.3. For $0 \leq j \leq n$, we call

$$c_j = [a_0, \dots, a_j]$$

the convergent of order j of $a = [a_0, \dots, a_n]$.

We denote by N_j and D_j the numerator and denominator of the irreducible fraction that represents c_j. For c_{-1} we put $N_{-1} = 1$ and $D_{-1} = 0$.

Theorem 1.4. *For $j \geq 1$ it holds that:*

$$\begin{aligned} N_j &= a_j N_{j-1} + N_{j-2}, \\ D_j &= a_j D_{j-1} + D_{j-2}. \end{aligned}$$

Proposition 1.5. *Let $a_0, a_1 \in \mathbb{Z}$, $a_1 > 0$. Let $q_0, \dots, q_{k-1}$ be the sequence of quotients generated by the Euclidean Algorithm applied to a_0 and a_1 and let $M^{(1)}, \dots, M^{(k-1)}$ be the matrices generated by the extended version of the EA. Then, for $j = -1, \dots, k-1$ the irreducible representation of the convergent of order j of a_0/a_1 c_j is $M^{(j)}_{11}/M^{(j)}_{21}$.*

Proof. The result clearly holds for $j = -1$. Then we have that $c_j = [q_0, \dots, q_j]$ by definition of convergent and by Proposition 1.2. For $j = 0$ we have $M^{(0)}_{11}/M^{(0)}_{21} = q_0 = c_0$ and $M^{(0)}_{12}/M^{(0)}_{22} = c_{-1}$; both of these fractions are irreducible. By induction let's suppose that $M^{(j-1)}_{11}/M^{(j-1)}_{21} = c_{j-1}$ and $M^{(j-1)}_{12}/M^{(j-1)}_{22} = c_{j-2}$ and that they are irreducible. In other words, we can write $M^{(j-1)} = \begin{pmatrix} N_{j-1} & N_{j-2} \\ D_{j-1} & D_{j-2} \end{pmatrix}$. Then, we get

$$M^{(j)} = \begin{pmatrix} N_{j-1} & N_{j-2} \\ D_{j-1} & D_{j-2} \end{pmatrix} \begin{pmatrix} q_j & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} N_j & N_{j-1} \\ D_j & D_{j-1} \end{pmatrix};$$

in the last equality we used the previous Theorem together with the Proposition 1.2. □

The following theorem holds:

Theorem 1.6. *The convergents of even order form an increasing sequence, whereas the convergents of odd order form a decreasing sequence. Furthermore, every even order convergent is less than a and every odd order convergent is greater than a.*

In other words, if n is even, it holds that

$$c_0 < c_2 < \cdots < c_n = a < c_{n-1} < \cdots < c_3 < c_1,$$

otherwise

$$c_0 < c_2 < \cdots < c_{n-1} < c_n = a < \cdots < c_3 < c_1.$$

Definition 1.7. Let a be a rational number and let $\bar{x}, \bar{y} \in \mathbb{Z}$ with $\bar{y} \geq 1$. We say that the fraction $\bar{y}/\bar{x}$ is a *best approximation of the second kind* of a if

$$|\bar{x}a - \bar{y}| < |xa - y|$$

for every $x, y \in \mathbb{Z}$ with $0 < x \leq \bar{x}$ such that $\bar{y}/\bar{x} \neq y/x$.

Theorem 1.8. *Let y/x be a best approximation of the second kind of a. Then y/x is a convergent of the standard continued fraction representing a.*

1.3 Lattices

A lattice is a set

$$\left\{ \sum_{i=1}^{k} \lambda_i b_i \mid \lambda_1, \ldots, \lambda_k \in \mathbb{Z} \right\}$$

where $b_1, \ldots, b_k \in \mathbb{R}^n$ are linearly independent vectors.

Lattices are integral combinations of linearly independent vectors. We can also say that they are a *discrete subgroup* of $\mathbb{R}^n$, meaning that there exists a small quantity ϵ such that all points in the lattice have at least distance ϵ from each other.

If $k = n$ then the lattice is said to be a *full-rank* lattice; since every lattice has full rank when restricted to $\operatorname{span}\{b_1, \ldots, b_k\}$ we will from now on assume that we are dealing with full rank lattices.

If we call B the matrix that has as column the vectors $b_1, \ldots, b_n$, then we can abbreviate the notation:

$$\Lambda(B) = \left\{ \sum_{i=1}^{n} \lambda_i b_i \mid \lambda_1, \ldots, \lambda_n \in \mathbb{Z} \right\}$$

The matrix B is called a *basis* of the lattice $\Lambda(B)$. It is clear that by adding an integral multiple of b_i to b_j for $j \neq i$ we are not changing the structure of the lattice. More formally:

Lemma 1.9. *Let $B_1, B_2 \in \mathbb{R}^{n\times n}$ non-singular. Then $\Lambda(B_1) = \Lambda(B_2)$ if and only if there is a unimodular matrix U such that $B_2 = B_1 U$.*

Proof. ($\Leftarrow$): U is unimodular, hence invertible. We need to observe that the map $f : \mathbb{Z}^n \longrightarrow \mathbb{Z}^n$ with $f(x) = Ux$ is a bijection on the integer lattice as $Ux \in \mathbb{Z}^n \; \forall x \in \mathbb{Z}^n$ and any vector $y \in \mathbb{Z}^n$ is such that $UU^{-1}y = y$. So we can write:

$$\Lambda(B_2) = \{B_2\lambda \mid \lambda \in \mathbb{Z}^n\} = \{B_1U\lambda \mid \lambda \in \mathbb{Z}^n\} = \Lambda(B_1).$$

($\Rightarrow$): $\Lambda(B_1) = \Lambda(B_2)$ means that any column of B_1 is an integral combination of columns in B_2 and vice versa. So we can find U and $V \in \mathbb{Z}^{n\times n}$ such that $B_2 = B_1U$ and $B_1 = B_2V$. Then

$$\det(B_1) = \det(B_2V) = \det(B_1UV) \Rightarrow \det(U), \det(V) \in \{-1, 1\}.$$

□

Given two matrix B_1 and B_2 one can find out in polynomial time if they generate the same lattice because the unimodular matrix U can be found in polynomial time using the Gauss elimination.

An $n \times n$ matrix A is in *Hermite normal form*[1] if A is an upper triangular non-negative matrix, with $a_{ii} > 0$, and $a_{ii} > a_{ij}$ for every $1 \leq i < j \leq n$.

It holds that every rational matrix Q with full rank can be brought into Hermite normal form through a unimodular matrix U, that is to say, there exists a unimodular matrix U such that $QU = H$, where H is in Hermite normal form. Furthermore, it holds that the Hermite normal form is unique.

From Lemma 1.9 it follows that every rational lattice has a unique basis in Hermite normal form. In other words, given a basis A of a rational lattice, there exists a unique H that is the Hermite normal form of A and it holds that $\Lambda(A) = \Lambda(H)$.

If we are dealing with a 2-dimensional rational lattice, its unique basis in Hermite normal form can be written as $\begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \in \mathbb{Q}^{2\times 2}$, where $c > 0$ and

[1]The Hermite normal form is defined also for rectangular matrices, but since we only deal with full-rank lattices, we will work only with square matrices; hence, we need the definition of Hermite normal form only for square matrices.

$a > b \geq 0$.

1.3.1 Shortest vector

A shortest vector of a lattice Λ is a nonzero vector $v \in \Lambda \setminus \{0\}$ of minimal norm $\|v\|$; this norm can be chosen among the ℓ_p-norms. The most studied case is the one with respect to the ℓ_2-norm. However, the natural norm for integer programming is the ℓ_∞-norm, and in particular, in the algorithm we are going to study, we will be interested in finding the shortest vector with respect to the ℓ_∞-norm. Namely, we are going to look for $v \in \Lambda \setminus \{0\}$ which minimizes $\|v\|_\infty = \max\{|v_i| : i = 1, 2\}$. There is a very quick way to find this vector in a 2-dimensional rational lattice.

Proposition 1.10. *Let $\Lambda \subseteq \mathbb{Q}^2$ be a rational lattice which is given by its Hermite normal form $\begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$. If neither $\begin{pmatrix} a \\ 0 \end{pmatrix}$ nor $\begin{pmatrix} b \\ c \end{pmatrix}$ are shortest vectors of Λ, then there exists a shortest vector of the form $\begin{pmatrix} -xa + yb \\ yc \end{pmatrix}$ where x/y is a best approximation of the second kind of b/a.*

Proof. First, observe that if $\begin{pmatrix} a \\ 0 \end{pmatrix}$ is not a shortest vector with respect to the ℓ_∞-norm, then we can assume that a shortest vector has positive second component; hence, the shortest vector can be written as $\begin{pmatrix} -xa + yb \\ yc \end{pmatrix}$ with $x \in \mathbb{N}_{\geq 0}$, $y \in \mathbb{N}_{>0}$. This fact is true also for any other norm that is invariant under the replacement of components by their absolute values; for instance, the ℓ_2-norm and the ℓ_1-norm.

Now, let

$$\begin{pmatrix} -xa + yb \\ yc \end{pmatrix}, \; x \in \mathbb{N}_{>0}, \; y \in \mathbb{N}_{>0}$$

be a shortest vector of Λ (with respect to the ℓ_∞-norm) with minimal ℓ_1-norm among all shortest vectors. We suppose that x/y is not a best approximation of the second kind of b/a. This means that there exist x', y' such that

$$x'/y' \neq x/y, \quad 0 < y' \leq y \quad \text{and} \quad |-x'a + y'b| \leq |-xa + yb|.$$

Since $\begin{pmatrix} -xa + yb \\ yc \end{pmatrix}$ minimizes the ℓ_1-norm, it holds that

$$y' = y \quad \text{and} \quad |-x'a + y'b| = |-xa + yb|.$$

This means that x and x' satisfy

$$|-xa+by| = |-x'a+by| = \min\{|-za+by| : z \in \mathbb{N}_{>0}\}.$$

Thus, assuming that $x' > x$ (the opposite case is analogous), it holds that $x' = x+1$. So we have

$$|-xa+by| = |-(x+1)a+by| \quad \text{with} \quad x/y \neq x'/y,$$

that is to say,

$$-xa+by = (x+1)a-by \Rightarrow by-ax = a/2.$$

Now, if $y > 1$ we can write $|b(y-1)-ax| = |a/2-b| \leq a/2$, but this would contradict the minimality of the ℓ_1-norm of $\begin{pmatrix} -xa+yb \\ yc \end{pmatrix}$. Therefore we have $y = 1$. Finally, $y = 1$, $b < a$ and $b - ax = a/2$ imply that $x = 0$, which is absurd because $\begin{pmatrix} b \\ c \end{pmatrix}$ is not a shortest vector by hypothesis. So x/y is a best approximation of the second kind of b/a. □

The following proposition will be of great utility.

Proposition 1.11. *It's given a lattice basis $A \in \mathbb{Q}^{2\times 2}$ and a sequence of positive rational numbers $\alpha_1, \ldots, \alpha_k$ which reveal themselves one after the other. A and each of the α_i have binary encoding length $O(\phi)$. We want to find a shortest vector with respect to the ℓ_∞-norm in each of the lattices $\Lambda(\begin{pmatrix} 1 & 0 \\ 0 & \alpha_i \end{pmatrix} A)$. After a preprocessing step that requires $O(\phi)$ arithmetic operations, each shortest vector query can be answered in $O(\log\phi)$.*

Proof. The preprocessing step consists in computing the Hermite normal form $\begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$ of the matrix A. This can be achieved, for instance, by using the extended Euclidean algorithm. Then we compute the convergents x_j/y_j of b/a with the Euclidean algorithm (see Proposition 1.2).

It holds that the sequence $|-x_j a + y_j b|$ is monotonously decreasing and the sequence $y_j c$ is monotonously increasing and non-negative. The Euclidean algorithm terminates in $O(\phi)$ steps, and there are as many convergents as those steps. Therefore, this preprocessing step requires $O(\phi)$ arithmetic operations.

In each query (i.e. for every $i = 1, \dots, k$) we have to determine the convergent x_j/y_j such that $\left\| \begin{pmatrix} -x_j a + y_j b \\ y_j \alpha_i c \end{pmatrix} \right\|_\infty$ is minimal. To do so, we look for j_i such that

$$|-x_{j_i} a + y_{j_i} b| \geq y_{j_i} \alpha_i c \quad \text{and} \quad |-x_{j_i+1} a + y_{j_i+1} b| < y_{j_i+1} \alpha_i c.$$

If $|-x_{j_i} a + y_{j_i} b| \geq y_{j_i} \alpha_i c$ holds for all convergents, then j_i will be the second-last position. If $|-x_{j_i+1} a + y_{j_i+1} b| < y_{j_i+1} \alpha_i c$ holds for all convergents, then j_i will be the first position. The shortest vector will be one of the following vectors:

$$\begin{pmatrix} a \\ 0 \end{pmatrix}, \begin{pmatrix} b \\ \alpha_i c \end{pmatrix}, \begin{pmatrix} -x_{j_i} a + y_{j_i} b \\ y_{j_i} \alpha_i c \end{pmatrix}, \begin{pmatrix} -x_{j_i+1} a + y_{j_i+1} b \\ y_{j_i+1} \alpha_i c \end{pmatrix}.$$

The j_i can be computed by binary search in $O(\log(\phi))$ steps. Therefore each query can be answered in time $O(\log(\phi))$. □

1.4 Flatness theorem

Definition 1.12. The width of a convex body K along an integral direction $c \in \mathbb{Z}^n$ is defined as follows:

$$w_c(K) := \max\{c^T x : x \in K\} - \min\{c^T x : x \in K\}.$$

Definition 1.13. The width of K is

$$w(K) := \min\{w_c(K) : c \in \mathbb{Z}^n\}.$$

It seems natural to think that if a certain body does not contain any integer point, then it has to be thin in some direction. This is true and the direction is exactly the direction we find searching for $c \in \mathbb{Z}^n$ that minimizes $w_c(K)$; it will be called a *flat direction* for K. More formally we have:

Theorem 1.14 (Flatness Theorem)**.** *There exists a constant $f(n)$ depending only on the dimension n, such that each full-dimensional convex body $K \subseteq \mathbb{R}^n$ containing no integer point has width less than $f(n)$.*

For any convex body it holds that $f(n) \leq O(n^{\frac{4}{3}} \cdot \log^{O(1)}(n))$. Hence, we can state that 2.5 is a good value for $f(2)$. Moreover, if K is a polytope, it holds that $f(n) \leq 2^{O(n^2)}$.

To prove the flatness theorem we need one of the most important results in convex geometry, which states that any convex body 'resembles' an ellipsoid. This result is known as the *John's theorem*.

Theorem 1.15 (John's theorem)**.** *For any convex body $K \subseteq \mathbb{R}^n$ there exists an ellipsoid $\mathcal{E}$ such that*

$$c + \mathcal{E} \subseteq K \subseteq c + n\mathcal{E},$$

where c is an appropriate translation.

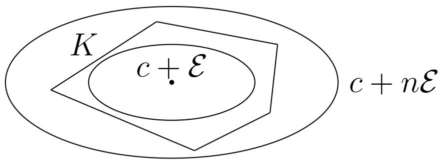


Figure 1.1: John's Theorem

As a first step, the flatness theorem is proved in the case of a unit ball. This is equivalent to proving the theorem for ellipsoids as an ellipsoid $\mathcal{E}$ becomes a unit ball through an appropriate linear transformation. More specifically, we have that $\mathcal{E} = \{x \in \mathbb{R}^n \mid \|H^{-1}x - H^{-1}a\|_2 \leq 1\} = \{a + Hy \mid \|y\|_2 \leq 1\}$ where a is the center of the ellipsoid and H is a non-singular matrix $\in \mathbb{R}^{n \times n}$. So to prove the flatness theorem for ellipsoids we apply the flatness theorem for balls to the ball $\mathcal{B}(H^{-1}a, 1)$ with the lattice $\Lambda(H^{-1})$.

Finally, since for any convex body K we can find an ellipsoid $\mathcal{E}$ such that $c + \mathcal{E} \subseteq K \subseteq c + n\mathcal{E}$, and since we know that the flatness theorem holds for an ellipsoid, it is possible to prove the flatness theorem also in the case of a general convex body K.

For the special case of a rational polytope, the ellipsoid $\mathcal{E}$ can be computed in polynomial time. Also, if the matrix H^{-1} and the center a defining the ellipsoid are rational, it is possible to compute either an integer point in the ellipsoid, or a flat direction for the ellipsoid, in polynomial time. Hence, the flatness theorem for rational polytopes can be reformulated as follows.

Theorem 1.16. *Let $K = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ be a rational polytope. Then in polynomial time one can find:*

1. *Either a point $x* \in K \cap \mathbb{Z}^n$;*
2. *Or a direction $c \in \mathbb{Z}^n$ with $w_c(K) \leq f(n)$ and $f(n) \leq 2^{O(n^2)}$.*

The application of the flatness theorem goes as follows: once we have found a flat direction c for K, we can have two cases. Either $w(K) \leq f(n)$ or $w(K) \geq f(n)$. In the first case, $K \cap \mathbb{Z}^n$ can be empty, whereas in the second case we know for sure that $K \cap \mathbb{Z}^n \neq \emptyset$. Moreover, in both cases, all the integer points contained in K lie in at most $w_c(K) + 1$ hyperplanes. These hyperplanes are of the form

$$K \cap \{x \in \mathbb{R}^n : c^T x = \delta\},$$

where

$$\delta \in \mathbb{Z} \cap [\min\{c^T x : x \in K\}, \max\{c^T x : x \in K\}].$$

The Flatness theorem plays an important role in the algorithm we are going to study. Our aim is to apply these ideas to the polygon $P \subseteq \mathbb{R}^2$ in which we are optimizing. Computing the flat direction and the width of a general polygon is not immediate; on the contrary, computing the flat direction and the width of a triangle is quite simple.

Therefore, instead of computing the width of the polygon P, we partition P into a certain number of polygons P_i. For each P_i, we find a triangle T_i included in P_i and such that P_i is included in a scaled copy (translated, if needed) of T_i. Namely: $T_i \subseteq P_i \subseteq kT_i + t$, with k homothety ratio and t a translation. We will see in details how to partition P and how to find these triangles. Once we have found them, we are going to study these triangles in order to bound their width between two values u and l (i.e. $l \leq w(T_i) \leq u$). In the next paragraph we explain this procedure. Finally, from $T_i \subseteq P_i \subseteq kT_i + t$ it holds that $w(T_i) \leq w(P_i) \leq kw(T_i)$; therefore, we have found an upper and a lower bound for $w(P_i)$, namely $l \leq w(P_i) \leq ku$, and we are ready to apply the Flatness Theorem.

1.4.1 Computing the width of a triangle

A triangle is the convex hull of three points. Since we are interested in computing its width and since the width is invariant under the effect of translations, we can assume that one of the vertices of the triangle is the origin. Therefore we can write $T = \operatorname{conv}(0, u, v)$ with $u, v \in \mathbb{R}^2$. It holds:

$$\begin{aligned} w_c(T) &= \max\{0, c^T u, c^T v\} - \min\{0, c^T u, c^T v\} \\ &= \max\{0, c^T u, c^T v\} + \max\{0, -c^T u, -c^T v\}. \qquad (1.1) \\ \Longrightarrow & \max\{\left|c^T u\right|, \left|c^T v\right|\} \leq w_c(T) \leq 2\max\{\left|c^T u\right|, \left|c^T v\right|\}. \end{aligned}$$

Now we define a matrix A_T associated with the triangle T: $A_T = \begin{pmatrix} u^T \\ v^T \end{pmatrix}$. The inequalities expressed in the last line of (1.1) can be rewritten as:

$$\|A_T c\|_\infty \leq w_c(T) \leq 2\|A_T c\|_\infty.$$

Taking the minimum over $c \in \mathbb{Z}^2$ yields

$$SV(\Lambda(A_T)) \leq w(T) \leq 2SV(\Lambda(A_T)),$$

where $\Lambda(A_T)$ is the lattice generated by the matrix A_T, namely $\Lambda(A_T) = \{A_T x : x \in \mathbb{Z}^2\}$ and $SV(\Lambda(A_T))$ is the length of the shortest vector in that lattice with respect to the ℓ_∞-norm. With this procedure, we also find a flat direction for T. Indeed, we have that the $c \in \mathbb{Z}^2$ such that $v = A_T c$ is the shortest vector in $\Lambda(A_T)$ is a flat direction fot the triangle T.

Chapter 2

A Fast Algorithm for IP in dimension 2

We are going to see an algorithm that solves the following problem:

$$\max\{c^T x : Ax \leq b, x \in \mathbb{Z}^2\},$$

where $c \in \mathbb{Z}^2, A \in \mathbb{Z}^{m\times 2}, b \in \mathbb{Z}^m$. This algorithm solves this problem in $O(m+\phi)$ where m is the number of constraints and ϕ is the maximal encoding length of the coefficients involved in the description of the problem (A, b, c).

We assume that the polygon $P = \{x \in \mathbb{R}^2 : Ax \leq b\}$ is bounded. We also assume that we are optimizing only one variable, namely x_2 (this can be done via an unimodular transformation). Therefore, our problem becomes

$$\max\{x_2 : Ax \leq b, x \in \mathbb{Z}^2\}.$$

2.1 Upper and Lower Polygons

We deal with two classes of polygons: upper and lower polygons. A lower polygon has a horizontal edge such that the whole polygon lays under it and through the endpoints of this edge we can draw two parallel lines enclosing the polygon.

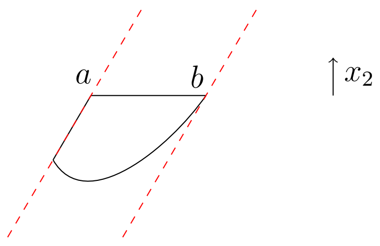


Figure 2.1: Lower polygon

An upper polygon has an horizontal edge such that the whole polygon lays above it and through the endpoints of this edge we can draw two parallel lines that enclose the polygon.

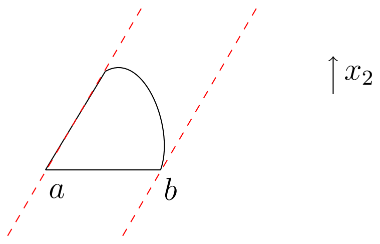


Figure 2.2: Upper polygon

2.2 Partitioning the Polygon

We want to work with these classes of polygons, therefore we partition P. First of all we look for the point e in P with maximum x_2 and the point f with minimum x_2 and draw a line ef. This line divides the polygon into two parts: a right part and a left part. In each of these parts we search for the vertex that maximizes the distance from the line ef and from it we draw an horizontal line. At this point we have two upper polygons that we will call Ul, Ur, namely *Upper left* and *Upper right*, and two lower polygons that we will call Ll, Lr, namely *Lower left* and *Lower right*. This partition can be done in $O(m)$.

We are going to solve the problem in each of these polygons. Actually, we see first how to solve the problem in one of the lower polygons, specifically Lr. Solving in Ll is analogous.

Before seeing how to solve the problem in an upper polygon, we see how to optimize in an upper triangle. Then, we briefly see the resolution in a polygon with a fixed number of constraints. Finally, taking into consideration all these assumptions, we will optimize in an upper polygon.

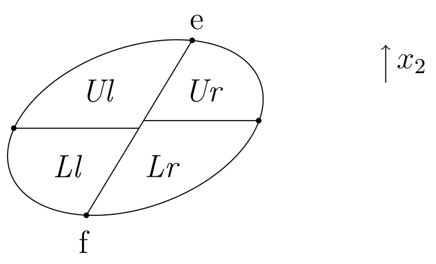


Figure 2.3: Partitioning of the polygon

2.3 Lower Polygons

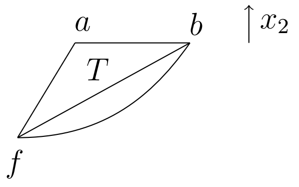


Figure 2.4: Lower-right polygon

We are now going to solve the problem in Lr. The first thing we are going to do is to find a triangle T such that $T \subseteq Lr \subseteq kT + t$, with k a certain homothety ratio and t an appropriate translation. Since Lr is a lower polygon, this triangle is abf, where a, b are the endpoints of the edge that lies above the polygon and f is the point with minimal x_2 in Lr. We have already found f in the partition of the original polygon; this point f is the vertex different from a of the edge that bounds the polygon on its left side.

This procedure works also in the case of a general lower polygon. In this case we actually have to search for the point with minimum x_2.

It holds that $T \subseteq Lr \subseteq 2T$; here we do not need any translation of the triangle. This is true because from b we can draw a line r (parallel to af) which enclose the polygon (we can do it because Lr is a lower polygon by construction) and f is the point with minimal x_2. Therefore, we have that the whole polygon is enclosed in the parallelepiped $abfv$ where v is the intersection between r and an horizontal line starting from f. When we draw $2T$ we double the length of the sides af and ab, getting two new points f', b'. The new triangle $ab'f'$ touches the parallelepiped in v and therefore the whole polygon is included in it (see Figure 2.5).

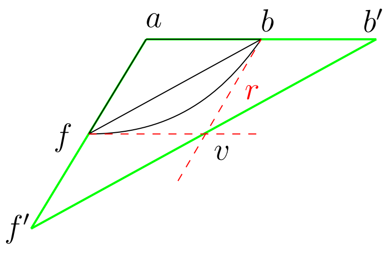


Figure 2.5: Building 2T

Because of the inclusions $T \subseteq Lr \subseteq 2T$, we have $w(T) \leq w(Lr) \leq 2w(T)$. To get an approximation of $w(T)$ we first find the matrix A_T as defined in section 1.4. Then we look for the shortest vector with respect to the ℓ_∞-norm in the lattice $\Lambda(A_T)$. This can be done using Proposition 1.10. Once we know the length of the shortest vector we can have two cases.

▶ First case: $SV(\Lambda(A_T)) \leq f(2)$.

In this case we have that $w(Lr) \leq 4f(2)$. This means that the integer points of Lr (if they exist) lie in at most $\lceil 4f(2) \rceil$ segments of the form:

$$Lr \cap \{x \in \mathbb{R}^2 : c^T x = \delta\},$$

where

$$\delta \in \mathbb{Z} \cap [\min\{c^T x : x \in Lr\}, \max\{c^T x : x \in Lr\}],$$

and c is a flat direction for T found through the query for the shortest vector in $\Lambda(A_T)$. We use the same c as a flat direction for Lr.

Some of these segments may be empty, meaning that there are no integer points laying on them. If Lr does not contain any integer point, then all of these segments will be empty. This case is possible because we don't know if $w(Lr) \geq f(2)$. In any case, we optimize in each of these segments; this procedure (which can be done for instance by using the Euclidean algorithm, as seen in section 1.1), requires $O(\phi)$ arithmetic operations for each segment. These segments are at most $\lceil 4f(2) \rceil = O(1)$, so the whole procedure costs $O(\phi)$. Finally, we take the optimum of the optima which we have found or, if none was found, we state that the problem is infeasible.

▶ Second case: $SV(\Lambda(A_T)) > f(2)$.

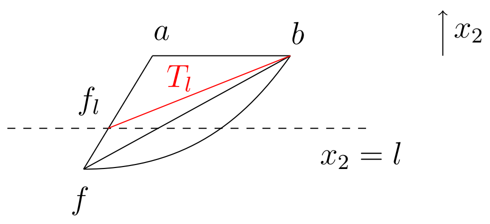


Figure 2.6: Truncated lower-right polygon

In this case we cannot conclude as quickly as in the first case. Indeed we have no upper bound on the width of Lr. So we are going to take a different approach. We look for a parameter l such that $f(2) \leq w(Lr_l) \leq 4f(2)$, where $Lr_l = Lr \cap \{x_2 \geq l\}$. If we can build a triangle T_l such that $w(T_l) \leq w(Lr_l) \leq 2w(T_l)$ (namely, $T_l \subseteq Lr_l \subseteq 2T_l + t$), this is equivalent to looking for l such that $SV(\Lambda(A_{T_l})) = f(2)$.

This T_l will be the triangle abf_l, where f_l is the intersection between the edge af and the line $x_2 = l$ (see Figure 2.6). We observe that T_l differs from the triangle T just for one vertex. Therefore we have that $A_{T_l} = \begin{pmatrix} 1 & 0 \\ 0 & \alpha_l \end{pmatrix} A_T$ for an appropriate $\alpha_l \in \mathbb{Q}$.

Our aim is to delete, through the introduction of the constraint $x_2 \geq l$, the lower part of the polygon, and we want this part to be as big as possible. In order to ignore this part, we must be sure that there still is at least one integer point in the upper part of the polygon.

This means that we want to find l as big as possible (equivalently α_l as small as possible) and such that $SV(\Lambda(A_{T_l}))$ is still greater than $f(2)$. As we have stated above, this exactly means looking for l such that $SV(\Lambda(A_{T_l})) = f(2)$.

Now, practically, we have to look for the shortest vector in $\Lambda(A_{T_l})$. From Proposition 1.10 we know how to find the shortest vector in $\Lambda(A_T)$ very quickly, once we have found the Hermite normal form of A_T.

Let's see how we find the shortest vector in $\Lambda(A_{T_l})$.

If the Hermite normal form of A_T is $\begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$, we compute the convergents x_j/y_j of b/a. Then we look for the first j such that $|-x_j a + y_j b| < f(2)$; α_l will be such that $\alpha_l y_j c = f(2)$, that is to say $\alpha_l = f(2)/y_j c$.

Indeed, if $|-x_j a+y_j b| < f(2)$ and $\alpha_l y_j c = f(2)$, then $|-x_j a+y_j b| < \alpha_l y_j c$. From the proof of Proposition 1.11 we know that the shortest vector in $\Lambda(A_{T_l})$

will be either $\begin{pmatrix} -x_j a + y_j b \\ \alpha_l y_j c \end{pmatrix}$ or $\begin{pmatrix} -x_{j+1} a + y_{j+1} b \\ \alpha_l y_{j+1} c \end{pmatrix}$. It can't be $\begin{pmatrix} a \\ 0 \end{pmatrix}$ because $SV(\Lambda(A_T)) > f(2)$.

Now,

$$| - x_{j+1}a + y_{j+1}b| < | - x_j a + y_j b| < \alpha_l y_j c \leq \alpha_l y_{j+1} c.$$

So,

$$f(2) = \left\| \begin{pmatrix} -x_j a + y_j b \\ \alpha_l y_j c \end{pmatrix} \right\|_\infty \leq \left\| \begin{pmatrix} -x_{j+1} a + y_{j+1} b \\ \alpha_l y_{j+1} c \end{pmatrix} \right\|_\infty .$$

With this α_l we have that $SV(\Lambda(AT_l)) = f(2)$. From α_l we can find the l we were looking for.

Once we have found this parameter l we have that $f(2) \leq w(Lr_l) \leq 4f(2)$. Since $w(Lr_l) \geq f(2)$, by the Flatness Theorem we know that the problem is feasible in Lr_l. And since $w(Lr_l) \leq 4f(2)$, then all the integer points contained in Lr_l lie on at most $\lceil 4f(2) \rceil$ segments included in Lr_l.

From the first observation and since we are maximizing x_2 we know that we can solve only in Lr_l, instead of solving in the entire polygon Lr. From the second observation we know that solving on Lr_l is equivalent to solving at most $\lceil 4f(2) \rceil$ 1-dimensional sub-problems. These sub-problems can each be solved in $O(\phi)$ arithmetic operations using, for instance, the Euclidean algorithm and they are $\lceil 4f(2) \rceil = O(1)$. So the problem can again be solved in $O(\phi)$ arithmetic operations.

2.4 Upper Triangles

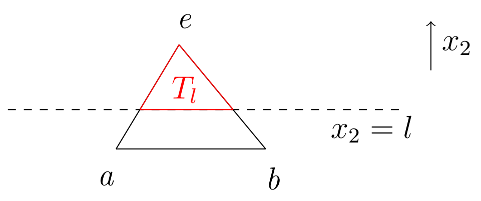


Figure 2.7: Upper triangle

We now move forward to the resolution of integer problems in upper triangles. We will use the same technique seen for lower polygons. First of all we compute the matrix A_T associated with the triangle T. Then we look for the shortest vector in $\Lambda(A_T)$. Now again we can have two cases.

▶ First case: $\mathrm{SV}(\Lambda(A_T)) \leq f(2)$.

In this case we know that $w(T) \leq 2f(2)$ and consequently all the integer points in T will lie in at most $\lceil 2f(2) \rceil$ segments of the form $T \cap \{x \in \mathbb{R}^2 : c^t x = \delta\}$ with $\delta \in \mathbb{Z} \cap [\min\{c^t x : x \in T\}, \max\{c^t x : x \in T\}]$. The vector $c \in \mathbb{Z}^2$ is a flat direction for T, and it is found through the query for the shortest vector in $\Lambda(A_T)$: c is the vector such that $v = A_T c$ is the shortest vector in $\Lambda(A_T)$.

▶ Second case: $\mathrm{SV}(\Lambda(A_T)) > f(2)$.

In this case we have no upper bound for $w(T)$, so we will proceed as in the previous case by looking for an l such that $f(2) \leq w(T_l) \leq 2f(2)$, where T_l is the truncated triangle $T \cap \{x_2 \geq l\}$, see Figure 2.7. As we have seen before, we can look for this l by searching for the shortest vector in $\Lambda(A_{T_l})$ and imposing that its length is equal to $f(2)$, namely $SV(\Lambda(A_{T_l})) = f(2)$. In this case, this is almost immediate because we see that T_l is exactly a scaled copy of T, i.e. there exists a constant β_l such that $A_{T_l} = \beta_l A_T$. Therefore $SV(\Lambda(A_{T_l})) = \beta_l SV(\Lambda(A_T))$.

Since we have already computed $SV(\Lambda(A_T))$ we find β_l by putting $\beta_l = f(2)/SV(\Lambda(A_T))$. From the computation of $SV(\Lambda(A_T))$ we find the flat direction c for the triangle T, which is a flat direction also for the triangle T_l since the matrices associated to the triangles are scaled copies. Therefore we can optimize only in T_l and we can do it by solving at most $\lceil 2f(2) \rceil = O(1)$ 1-dimensional problems.

2.5 Polygons with a fixed number of constraints

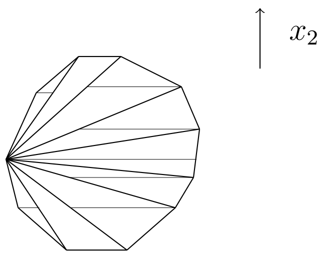


Figure 2.8: Triangulation of a polygon

In the case of a polygon defined by a fixed number of constraints, the first thing we are going to do is to triangulate the polygon, for example by drawing a line from one fixed vertex to all the others, except for the two adjacent vertices. In this way we get $O(1)$ triangles (they would be $O(m)$ but here m is constant).

We divide each triangle into an upper triangle and a lower triangle by drawing an horizontal line from one of its vertices (which has been properly selected). At this point we have a fixed number of upper and lower triangles.

We know how to solve in upper triangles and we know how to solve in lower polygons, therefore also in lower triangles. Each resolution can be done in $O(\phi)$ yielding a total cost of $O(\phi)$.

Observe that this algorithm works also for a general polygon, but its cost will be $O(m\phi)$, whereas the cost we strive to achieve is $O(m + \phi)$.

2.6 Upper Polygons

The last case we have to investigate is the case of an upper polygon.

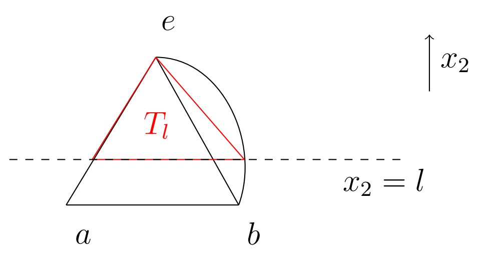


Figure 2.9: Upper-right polygon

Here the procedure will be slightly more complicated. First of all we approximate the polygon Ur with the triangle $T = abe$. It holds that $T \subseteq Ur \subseteq 2T$, with no translation required for the same reasons as in section 2.3. At this point we build the matrix A_T and we compute $SV(\Lambda(A_T))$. We can have two cases.

▶ First case: $SV(\Lambda(A_T)) \leq f(2)$.

In this case we know that $w(Ur) \leq 4f(2)$. Then we can solve the problem by optimizing over at most $\lceil 4f(2) \rceil$ 1-dimensional subproblems. This will take $O(\phi)$ operations.

▶ Second case: $SV(\Lambda(A_T)) > f(2)$.

In this case we have no upper bound for $w(Ur)$; we will look for an l such that $f(2) \leq w(Ur_l) \leq 4f(2)$, where $Ur_l = Ur \cap \{x_2 \geq l\}$. As in the other cases, this means looking for an l such that $SV(\Lambda(A_{T_l})) = f(2)$ where the triangles T_l are built as in Figure 2.9. When we build the triangles T_l, we observe that two vertices are changing. Therefore we have that $A_{T_l} = \beta_l \begin{pmatrix} 1 & 0 \\ 0 & \alpha_l \end{pmatrix} A_T$, with $\beta_l, \alpha_l \in \mathbb{Q}$.

In the case of a lower polygon and in the case of an upper triangle we could find l quickly, whereas here we can't. Here we look for l by trials and we need to limit the number of trials, so that our algorithm is still efficient.

We achieve this goal by pruning constraints while we look for l; we will prune constraints such that the solution of the problem remains invariant. To do so, we add two new constraints that will somehow take the place of the constraints that will be deleted. These constraints are of the form $v \leq x_2 \leq u$.

Observe that if we delete a certain number of constraints at each iteration, for example $1/n$ of them with n a fixed integer, then in $O(\log m)$ we will end with a polygon defined by a fixed number of constraints (and therefore we will know how to solve the problem quickly, see Section 2.5).

In particular our algorithm discards $1/4$ of the constraints at each iteration, therefore we will end with a polygon described by 4 constraints in $O(\log m)$ steps. Indeed, at the i-th iteration we are left with $\left(\frac{3}{4}\right)^i m$ constraints. If we stop at k such that $\left(\frac{3}{4}\right)^k m = 4$ it holds that $k = O(\log m)$.

At the beginning of the algorithm we have $v = x_2$-coordinate of the edge ab, and $u = x_2$-coordinate of the vertex e. Observe that adding these two constraints does not change the polygon.

Also, observe that, a part from the edges ab and ae, all the other edges, from left to right, have slopes which start from maximum 0, decrease to a minimum of $-\infty$ and then possibly decrease again from $+\infty$ until reaching minimum the slope of ae. This is true because e was the point maximizing the x_2-coordinate, and because b was the point maximizing the distance from ef.

In the first step of the algorithm we pair up the m original constraints defining the polygon and we intersect them, getting $m/2$ intersection points. We compute the median of the x_2-coordinates of these points and we call this value l_{med}. This is a candidate for the parameter l we are looking for. We can have three cases.

1. First case: $l_{med} \leq v$.

Since l_{med} is the median of the x_2-coordinates, we have $m/4$ intersection points that lie under the line $x_2 = l_{med}$. We use these points to select the constraints to prune.

Each of these points is defined by the intersection of a pair of constraints. We look at the slopes of these two constraints. If both the slopes are negative we prune the constraint with bigger slope in absolute value; if both the slopes are positive then we prune the constraint with smaller slope in absolute value; if one is positive and the other negative we prune the constraint with positive slope.

Visually we are pruning the constraints which lie lower; because of the constraint $x_2 \geq v$, the removal of these constraints does not change the solution of the problem.

In this operation we are removing 1/4 of the constraints. Since $l_{med} < v$, we do not change the bounds $v \leq x_2 \leq u$.

2. Second case: $l_{med} \geq u$.

This case is analogous to the previous one. Here we have $m/4$ points lying above the line $x_2 = l_{med}$ and therefore above the line $x_2 = u$. These points are each defined by a pair of constraints.

If the slopes of the two constraints are both negative we prune the constraint with smaller slope in absolute value; if the slopes are both positive we prune the constraint with greater slope; if one slope is negative and the other is positive we prune the constraint with negative slope.

Also in this case no changes are made to the constraints $v \leq x_2 \leq u$.

3. Third case: $v < l_{med} < u$.

This is the most interesting case; here we modify the constraints $v \leq x_2 \leq u$. We narrow them by replacing either v or u with the value of l_{med} .

To decide how to proceed we first compute the triangle $T_{l_{med}}$. Then, we compute the matrix associated to it:

$$A_{T_{l_{med}}} = \beta_{l_{med}} \begin{pmatrix} 1 & 0 \\ 0 & \alpha_{l_{med}} \end{pmatrix} A_T$$

for certain $\beta_{l_{med}}, \alpha_{l_{med}} \in \mathbb{Q}$. We compute the shortest vector in $\Lambda(A_{T_{l_{med}}})$. We can have three cases.

- First case: $SV(\Lambda(A_{T_{l_{med}}})) < f(2)$.

In this case we replace the value of u with the value of l_{med}. The constraint $x_2 \leq u$ becomes $x_2 \leq l_{med}$. Our algorithm always modifies the values of u in order to have $SV(\Lambda(A_u)) < f(2)$. This makes sense because it means that we are ignoring a flat part of the polygon: $Ur \cap \{x_2 \geq u\}$; in this part of the polygon we can always solve the problem in $O(\phi)$. A part from changing this constraint we will also remove 1/4 of the original constraints defining the polygon. We will look at the points laying above $x_2 = l_{med}$ and we will prune constraints as in 2.

• Second case: $SV(\Lambda(A_{T_{l_{med}}})) > f(2)$.

In this case we replace the value of v with the value of l_{med}. The constraint $x_2 \geq v$ becomes $x_2 \geq l_{med}$. The value of v is always modified in order to have $SV(\Lambda(A_v)) > f(2)$. This inequality enables us to ignore the lower part of the polygon $Ur \cap \{x_2 \leq v\}$. Indeed, by the flatness theorem, the problem is feasible in the upper part $Ur \cap \{x_2 \geq v\}$; since we are maximizing x_2 we can solve the problem only in this part. Also in this case we will have $m/4$ points laying under the line $x_2 = l_{med}$. We will prune constraints as in 1.

• Third case: $SV(\Lambda(A_{T_{l_{med}}})) = f(2)$.

In this case l_{med} is exactly the l we were looking for from the beginning. Therefore the algorithm stops and we solve the problem in $Ur_{l_{med}}$. We have that $f(2) \leq w(Ur_{l_{med}}) \leq 4f(2)$.

So the problem is feasible in $Ur_{l_{med}}$ and we have to solve at most $\lceil 4f(2) \rceil$ 1-dimensional sub-problems. Specifically we have to optimize in the segments $\{x \in \mathbb{R}^2 : c^T x = \delta\} \cap Ur_{l_{med}}$ with c flat direction for $Ur_{l_{med}}$ and $\delta \in \mathbb{Z} \cap [\min\{c^T x : x \in Ur_{l_{med}}\}, \max\{c^T x : x \in Ur_{l_{med}}\}]$. The flat direction c was actually a flat direction for the triangle $T_{l_{med}}$, and it was found through the search for the shortest vector in $\Lambda(A_{T_{l_{med}}})$.

This is the procedure of one step of our algorithm. If we didn't stop, namely $SV(\Lambda(A_T)) > f(2)$ and we didn't end up in case (c) ($SV(\Lambda(A_{T_{l_{med}}})) = f(2)$), then we are going to repeat the procedure again with the new set of constraints that we are left with at the end of this iteration. We are going to repeat this procedure until we end in case (c) or we are left with a fixed number of constraints defining the intermediate part of the polygon $Ur \cap \{v \leq x_2 \leq u\}$. If we terminate by ending up in case (c) we have already seen how to solve the problem. In the other case we have to solve the problem in

$$\big(Ur \cap \{v \leq x_2 \leq u\}\big) \cup \big(Ur \cap \{x_2 \geq u\}\big).$$

For the first term of this union we have a description with a fixed number of constraints, and so we are going to solve the problem using the method described in section 2.5. The problem is solved in $O(\phi)$. Regarding the second term of this union, we know that it is flat by construction. Indeed, u was modified in order to keep this part of the polygon flat. So in $O(\phi)$ we can find out whether this part contains or not integer points and if so, we can find a flat direction by using the triangle T_u and then solve the problem in at most $\lceil 4f(2) \rceil$ 1-dimensional subproblems.

2.7 Computational Analysis

We have seen that partitioning the polygon costs $O(m)$, and that solving on a lower polygon, on an upper triangle and on a polygon with a fixed number of constraints costs each $O(\phi)$. So the last step we have to make is studying the computational cost of the resolution in an upper polygon.

We have already seen that we are sure that the algorithm ends in at most $O(\log m)$ steps. But how much does each iteration cost?

Let's say that at the i-th iteration we are left with m_i constraint. In this iteration we have to pair them up, compute their intersections, compute the median and then find the shortest vector in the lattice $\Lambda(A_{T_{l_{med}}})$. Let's ignore for one moment the query for the shortest vector. The remaining cost is $O(m_i)$ at each iteration. Now at every step of the algorithm we are removing 1/4 of the constraints, so in fact we have $m_i = \left(\frac{3}{4}\right)^i m$. Therefore the total cost (ignoring the query for the shortest vector) is:

$$\sum_{i=1}^{log(m)} \left(\frac{3}{4}\right)^i m = m \left(\frac{1 - \frac{3}{4}^{log(m)+1}}{1 - \frac{3}{4}} \right) = O(m).$$

Concerning the costs for the query for the shortest vector, Proposition 1.11 comes in our help. Indeed we can see A_T as the matrix A of the Proposition; moreover it holds that the shortest vector of $\Lambda\big(\beta_l \begin{pmatrix} 1 & 0 \\ 0 & \alpha_l \end{pmatrix} A_T\big)$ is β_l times the shortest vector of $\Lambda\big(\begin{pmatrix} 1 & 0 \\ 0 & \alpha_l \end{pmatrix} A_T\big)$. The α_l are rationals which reveal themselves one after the other and so the queries for the shortest vector are exactly as the ones described in Proposition 1.11. We are iterating maximum $\log m$ times, therefore the total cost for the queries is $O(\phi + \log\phi \log m)$. With this last analysis we can finally state the following theorem.

Theorem 2.1. *A two-variable integer programming problem* $\max\{c^t x : Ax \leq b, x \in \mathbb{Z}^2\}$ *with* $A \in \mathbb{Z}^{m \times 2}$*,* $b \in \mathbb{Z}^m$*,* $c \in \mathbb{Z}^2$ *involving coefficients of maximal binary encoding length* $O(\phi)$*, can be solved with* $O(\phi + m)$ *arithmetic operations.*

Chapter 3

SVP in ℓ_∞-norm

3.1 Gaussian Algorithm

It is now clear how strongly connected is the shortest vector problem in ℓ_∞-norm with integer programming. For this reason, and many others, the problem of finding the shortest vector in a lattice has been extensively studied. One approach to the problem is to find a *reduced* basis for the given lattice, where a reduced basis satisfies some specific properties which allow us to find the shortest vector, or at least an approximation of it.

The first algorithm for lattice basis reduction was developed by Gauss and it works in dimension 2. It was originally built to work with the ℓ_2-norm; however, we are going to see an extended version of this algorithm which works with any efficiently computable norm. Hence, this version works also with the ℓ_∞-norm.

It takes as input a pair of vectors a, b which are the basis of the lattice. It returns a', b' where a' is the shortest vector in the lattice with respect to the chosen norm $\|\cdot\|$.

This algorithm can be seen as an extension of the Euclidean algorithm, as it subtracts integer multiples of the shorter vector between a and b from the larger one, reducing its length. This step resembles the division with remainder in the Euclidean algorithm.

We give some definitions.

Definition 3.1. Let $[a, b]$ be a lattice basis. This basis is *reduced* if

$$\|a\|, \|b\| \leq \|a + b\|, \|a - b\|.$$

Definition 3.2. We define the *i-th successive minimum* as the value λ_i such that

$$\lambda_i(\Lambda) := \min\{r \geq 0 \mid \dim(\operatorname{span}(\mathcal{B}(0,r) \cap \Lambda)) \geq i\},$$

meaning that we have i many linearly independent vectors of length at most λ_i. Of course it holds that $\lambda_1 = SV(\Lambda)$.

The following theorem holds:

Theorem 3.3. *Let $[a,b]$ be a lattice basis and let λ_1 and λ_2 be the successive minima of the lattice generated by a, b. Then, $[a,b]$ is reduced if and only if a and b have norm λ_1 and λ_2.*

To prove this theorem we need a lemma:

Lemma 3.4. *Suppose we have three vectors on a line $x, x+y, x+\alpha y$ with $\alpha \in (1,\infty)$. If $\|x\| \leq \|x+y\|$, then $\|x+y\| \leq \|x+\alpha y\|$ and if $\|x\| < \|x+y\|$, then $\|x+y\| < \|x+\alpha y\|$.*

Proof. We define $\delta = 1/\alpha$. Then, we can write

$$x + y = (1-\delta)x + \delta(x + \alpha y).$$

This implies

$$\|x+y\| \leq (1-\delta)x + \delta(x+\alpha y).$$

For the hypothesis of the Lemma, we have

$$\|x+y\| < (1-\delta)\|x+y\| + \delta(x+\alpha y),$$

which implies

$$\delta\|x+y\| < \delta\|x+\alpha y\|.$$

By definition $\delta > 0$ so we can divide by δ and conclude. $\square$

We are now ready to prove the theorem:

Proof. ($\Longleftarrow$) First let's assume, without loss of generality, that $\|a\| = \lambda_1$. This implies that $\|a-b\|, \|a+b\| \geq \|a\|$. Since a, b were linearly independent vectors, also $a-b$ and $a+b$ are linearly independent from a. So by definition of λ_2 we have that

$$\lambda_2 \leq \max\{\|a\|, \|a-b\|\} = \|a-b\|$$

and

$$\lambda_2 \leq \max\{\|a\|, \|a+b\|\} = \|a+b\|.$$

This implies

$$\|a\|, \|b\| \leq \|a+b\|, \|a-b\|.$$

($\Longrightarrow$) Now assume that $\|a\|, \|b\| \leq \|a+b\|, \|a-b\|$ and without loss of generality that $\|a\| \leq \|b\|$. Let us build an arbitrary lattice vector $ra+sb$ where $r, s \in \mathbb{Z}$. We want to show that

$$\|a\| \leq \|ra+sb\| \qquad \forall\ (r,s) \neq (0,0)$$

and

$$\|b\| \leq \|ra+sb\| \qquad \forall\ s \neq 0.$$

There are three cases:

- If $s=0$: $\|a\| \leq \|ra\| = \|ra+sb\|$ proving the first inequality;
- If $r=0$: $\|a\| \leq \|b\| \leq \|sb\| = \|ra+sb\|$ proving both inequalities;
- If $r \neq 0$ and $s \neq 0$ let's assume $r \geq s \geq 0$. We can write

  $$\|(r/s)a+b\| = \|\frac{ra+sb}{s}\| \leq \|ra+sb\|.$$

  Notice that $\|b\| \leq \|b+a\|$ and $r/s \geq 1$, therefore by the previous lemma,

  $$\|a\|, \|b\| \leq \|a+b\| \leq \|b+(r/s)a\| \leq \|ra+sb\|.$$

$\square$

We give another definition.

Definition 3.5. A basis $[a,b]$ is well ordered if $\|a\| \leq \|a-b\| < \|b\|$.

The first part of the algorithm tries to transform the input basis into a well ordered one. It could happen that already in this phase we find a reduced basis. In this case, we return it and we are done. Otherwise, with a well ordered basis, we enter the second part of the algorithm, which is a loop that we exit when we have found a reduced basis.

- First part:

```
if $\|a\| > \|b\|$ then swap$(a,b)$;
if $\|a-b\| > \|a+b\|$ then let $b := -b$;
if $\|b\| \leq \|a-b\|$ then return $[a,b]$;
if $\|a\| \leq \|a-b\|$ then go to the second part;
if $\|a\| = \|b\|$ then return$[a, a-b]$;
let $[a,b] := [b-a, -b]$;
```

- Second part (loop):
  Find $\mu \in \mathbb{Z}$ such that $\|b - \mu a\|$ is minimal;
  if $\|a - b\| > \|a + b\|$ then let $b := -b$;
  if $[a, b]$ is reduced:
  then return $[a, b)$;
  else swap(a, b) and go to loop.

Thanks to the first two lines we can assume that $\|a\| \leq \|b\|$ and that $\|a - b\| \leq \|a + b\|$. Then if $\|b\| \leq \|a - b\|$, the basis $[a, b]$ is already reduced and we are done. So it holds $\|b\| > \|a - b\|$. If $\|a\| \leq \|a - b\|$, the basis is well-ordered and we can access the second part of the algorithm. So assume $\|b\| > \|a - b\|$ and $\|a\| > \|a - b\|$. We can have that $\|a\| < \|b\|$ or that $\|a\| = \|b\|$. In the first case, (i.e. $\|a\| < \|b\|$), the basis $[a - b, -b]$ is well-ordered. Indeed, $\|b - a\| \leq \|a\| < \|b\|$; therefore we can access the loop with this basis. In the second case, (i.e. $\|a\| = \|b\|$), the basis $[a, a - b]$ is reduced and we return it; indeed $\|a - (a - b)\| = \|b\| = \|a\|$, $\|a + a - b\| = \|2a - b\| \geq |\|2a\| - \|b\|| = \|a\| = \|b\|$ and $\|a - b\| \leq \|2a - b\|$ because $\|a - b\| < \|a\|$ and $\|a\| \leq \|2a - b\|$, whereas $\|a - b\| \leq \|b\|$ comes directly from the hypothesis of this case.

In the second part of the algorithm we have a well-ordered basis $[a, b]$ and the first thing we are going to do is to make b as short as possible by subtracting an integer multiple of a. Then we make sure that it holds $\|a - b\| \leq \|a + b\|$, by changing the sign of b if needed. If the basis is reduced we return it, otherwise we swap the roles of a and b and we enter again the loop.

Let's see in details how we find $\mu \in \mathbb{Z}$ such that $\|b - \mu a\|$ is minimal. The following lemma tells us that we can find such a μ efficiently if the basis $[a, b]$ is well-ordered.

Lemma 3.6. *Let a and b be two vectors such that $\|b\| > \|b - a\|$. Then, one can efficiently find an integer μ such that $\|b - \mu a\|$ is minimal. Furthermore, it holds that $1 \leq \mu \leq 2\frac{\|b\|}{\|a\|}$.*

Proof. Let us define $c := \left\lceil 2\frac{\|b\|}{\|a\|} \right\rceil$. It holds that

$$\|b - ca\| \geq |\|b\| - c\|a\|| \geq c\|a\| - \|b\| \geq \|b\|,$$

and so by Lemma 3.4 it holds that $\|b - ca\| \leq \|b - (c+1)a\|$. Therefore we have that the following inequality

$$\|b - ka\| \leq \|b - (k+1)a\|$$

is true for $k = c$, but it is false for $k = 0$. We can find, using for instance binary search, an integer $1 \leq \mu \leq c$ such that

$$\|b - (\mu - 1)a\| > \|b - \mu a\| \leq \|b - (\mu + 1)a\|.$$

This value minimizes $\|b - \mu a\|$. Indeed, by Lemma 3.4, it holds that $\|b - \mu a\| \leq \|b - (\mu+1)a\| \leq \|b - ka\|$ for every $k \geq \mu + 1$. And in the same way it holds also that $\|b - \mu a\| < \|b - (\mu - 1)a\| < \|b - ka\|$ for every $k \leq \mu - 1$. □

We have seen that, in order to find μ efficiently, it is sufficient to have a well-ordered basis at the beginning of each iteration. Therefore we need to prove the following:

Lemma 3.7. *In any execution of the Gauss algorithm, at the beginning of each iteration the basis $[a, b]$ is well-ordered.*

Proof. We know that the first time we enter the loop the basis is well-ordered. We have to see that at the end of each iteration the basis is either reduced or well-ordered. If $[a, b]$ is the basis which enters the loop, we call $[a', b']$ the basis which exits the loop. It holds that $a' = \pm(b - \mu a)$ and $b' = a$. We know that $\|a' - b'\| \leq \|a' + b'\|$ and also that $\|a' - b'\| = \| \pm (b - \mu a) - a\| = \|b - (\mu \pm 1)a\| \geq \|b - \mu a\| = \|a'\|$. So we have that $\|a'\| \leq \|a' - b'\| \leq \|a' + b'\|$. At this point we can have two cases: either $\|b'\| \leq \|a' - b'\|$ and then $[a', b']$ is reduced, or $\|b'\| > \|a' - b'\|$ and then $[a', b']$ is well-ordered. □

Observe that at each iteration we are shortening the length of the vectors by a constant factor, therefore the algorithm will certainly terminates. More precisely:

Theorem 3.8. *For any choice of two linearly independent vectors $[a, b]$, the algorithm always terminates and correctly computes a reduced basis for the lattice Λ generated by a and b.*

Proof. We already know that if the algorithm terminates then the basis $[a, b]$ is reduced. Moreover, we are subtracting lattice vectors from each other, therefore $[a, b]$ is still a basis of the original lattice. We need to see that the algorithm does not loop forever. This cannot happen because we know that at the beginning of each iteration, $\|b - a\| < \|b\|$. Therefore, b is replaced by a new vector which is strictly shorter. This implies that the algorithm must stop after a finite number of iterations. □

The last thing we want to prove is the fact that the number of iterations is polynomial in the size of the input. Let us call this number k and let $[a_k, a_{k+1}]$ be the well-ordered basis at the beginning of the loop. Then the reduced basis will be $[a_1, a_2]$. It holds that:

Lemma 3.9. *For every $i \geq 3$, $\|a_i\| < 1/2\|a_{i+1}\|$.*

Proof. Let us call the subsequence $(a_{i-1}, a_i, a_{i+1}) = (a, b, c)$. It holds that both $[a, b]$ and $[b, c]$ are well-ordered, $\|a\| < \|b\| < \|c\|$ and $a = \pm(c - \mu b)$. If we call $\epsilon = \pm 1$, we get that $c = \epsilon a + \mu b$. Our goal is to prove that $\|c\| > 2\|b\|$.

- Case $\mu = 1$. It would mean $\|c - b\| = \|a\| < \|b\|$, but this is not possible as the basis $[b, c]$ is well-ordered. Hence, this case is excluded.

- Case $\mu = 2, \epsilon = -1$. In this case we have $\|c - b\| = \|-a + b\|$ and this is not possible because it also holds that $\|a - b\| < \|b\|$ and $\|b\| \leq \|b - c\|$.

- Case $\epsilon = -1,\ \mu > 2$. We have $\|c\| = \|-a + \mu b\| \geq \mu\|b\| - \|a\| > (\mu - 1)\|b\| \geq 2\|b\|$.

- Case $\mu \geq 2, \epsilon = 1$. In this case $\|c\| = \|a + \mu b\|$. First, $\|b - a\| < \|b\|$ as $[a, b]$ is well-ordered. This implies that $\|b\| < \|b + a\|$. Then, $\|a\| \leq \|a - b\|$ implies $\|a\| < \|b + a\|$, and so

  $$\|a\| < \|a + b\| < \|a + 2b\| < \|a + \mu b\|.$$

  So we have just proved that $\|c\| = \|a + \mu b\| \geq \|a + 2b\|$. We need to prove that $\|a + 2b\| > 2\|b\|$. It holds that

  $$\|2b - a\| \leq \|b\| + \|b - a\| < \|b\| + \|b\| = 2\|b\|.$$

  This implies

  $$\|2b - a\| < \|2b\| < \|2b + a\|,$$

  and therefore $\|c\| > 2\|b\|$.

$\square$

By induction, this lemma implies that

$$\|a_i\| \geq 2^{i-3}\|a_3\|,$$

and in particular

$$2^{k-2} \leq 2^{k-2}\|a_3\| \leq \|a_{k+1}\| \leq \|a\| + \|b\|.$$

Therefore, $k \leq 2 + \log_2(\|a\| + \|b\|)$, that is to say, the running time of the algorithm is polynomial in the size of the input. Equivalently, the shortest vector problem in dimension 2 can be solved in polynomial time.

3.2 A $2^{O(n)}$-time algorithm for the SVP

We describe the randomized algorithm from Ajtai, Kumar and Sivakumar which solves the shortest vector problem in dimension n in time $2^{O(n)} \cdot \text{poly(input)}$ (see [2], [12], [13]). Similarly to the Gaussian algorithm, this algorithm uses the idea of iteratively subtract lattice vectors from each other in order to reduce their length. A random factor is needed in order to avoid the scenario in which the algorithm ends with only zero vectors.

The problem is the following: it's given a lattice $\Lambda(B)$ with $B \in \mathbb{Q}^{n\times n}$, and the goal is to find a vector $x \in \Lambda(B) \setminus \{0\}$ which minimizes $\|x\|_2$. The norm has been chosen as the ℓ_2-norm, but we will prove that the algorithm works also with the ℓ_∞-norm.

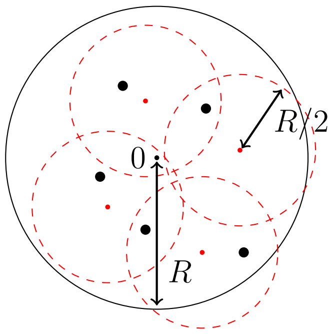
3.2.1 Some useful geometric insights

First, we assume to work with a lattice $\Lambda(B)$ such that $2 \leq SV(\Lambda(B)) \leq 3$. Indeed the following lemma holds.

Lemma 3.10. *Given an algorithm A that finds a shortest non-zero vector in lattices $\Lambda(B)$ for which $2 \leq SV(\Lambda(B)) \leq 3$, we can find a shortest non-zero vector in any lattice in time that is greater by a factor of at most $O(n)$.*

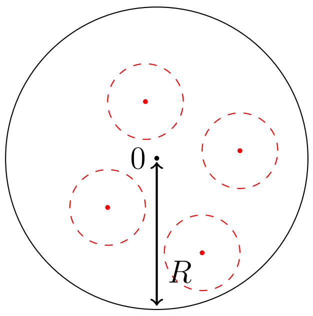
We give a series of useful results.

Lemma 3.11. *Let $X \subseteq \mathcal{B}(0, R) \subseteq \mathbb{R}^n$ be a finite set of points. Then, one can find a subset of centres $C \subseteq X$ with $|C| \leq 5^n$ so that $d(x, C) \leqslant \dfrac{R}{2}$ for all $x \in X$.*



Proof. We start with $C := \emptyset$ and we greedily add a point x from X to C if $d(x, C) > R/2$. In this way we get a set of clusters such that $d(x, C) \leqslant \dfrac{R}{2}$

for all $x \in X$; we need to prove that $|C| \le 5^n$. It holds that $\|c - c'\| > R/2$ for $c, c' \in C, c \neq c'$. Hence, $\mathcal{B}(c, \frac{R}{4}) \cap \mathcal{B}(c', \frac{R}{4}) = \emptyset$ for all $c, c' \in C, c \neq c'$.



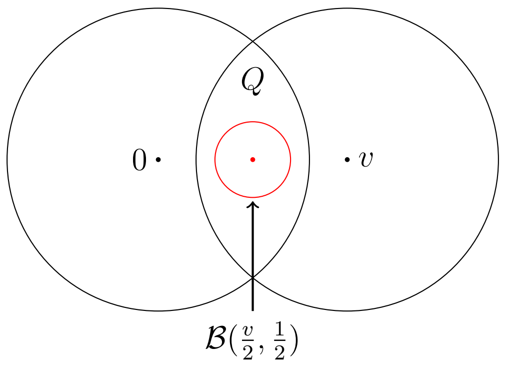
All these balls are fully contained in $\mathcal{B}(0, \frac{5}{4}R)$ and it holds that

$$\frac{\operatorname{vol}(\mathcal{B}(0, \frac{5}{4}R))}{\operatorname{vol}(\mathcal{B}(0, \frac{R}{4}))} = 5^n.$$

Therefore there are maximum 5^n centres. □

Lemma 3.12. *Let $v \in \mathbb{R}^n$ be a vector of length $2 \le \|v\|_2 \le 3$. Let $Q := \mathcal{B}(0,2) \cap \mathcal{B}(v,2)$. Then*

$$\frac{\operatorname{vol}(Q)}{\operatorname{vol}(\mathcal{B}(0,2))} \ge 2^{-2n}$$



Proof. If a ball $\mathcal{B}(\frac{v}{2}, \frac{1}{2})$ is contained in Q, then the volume ratio is at most 2^{2n} because

$$2^{-2n} = \frac{\operatorname{vol}\big(\mathcal{B}(\frac{v}{2}, \frac{1}{2})\big)}{\operatorname{vol}\big(\mathcal{B}(0,2)\big)} \le \frac{\operatorname{vol}\big(Q\big)}{\operatorname{vol}\big(\mathcal{B}(0,2)\big)}$$

Let's see that $\mathcal{B}(\frac{v}{2},\frac{1}{2}) \subseteq Q$, that is to say, if x is such that $d(x,\frac{v}{2}) \leq \frac{1}{2}$, then x satisfies $d(x,0) \leq 2$ and $d(x,v) \leq 2$.

This is true because $d(x,0) \leq d(x,\frac{v}{2}) + d(\frac{v}{2},0) \leq \frac{1}{2}+\frac{3}{2} = 2$ and $d(x,v) \leq d(x,\frac{v}{2}) + d(\frac{v}{2},v) \leq \frac{1}{2}+\frac{3}{2} = 2$.

□

The following lemma implies that there are not too many short vectors in a lattice.

Lemma 3.13. *Let $\Lambda \subseteq \mathbb{R}^n$ be a lattice with $SV(\Lambda) \geq 2$. Then $|\Lambda \cap \mathcal{B}(0,8)| \leq 2^{4n}$.*

Proof. Since the shortest vector has length greater than two, we can put a ball of radius 1 around each lattice point in $\Lambda \cap \mathcal{B}(0,8)$ and these balls will not overlap. All these balls are contained in $\mathcal{B}(0,9)$, therefore they cannot be more than $9^n < 2^{4n}$. □

For the sake of completeness, we give an interesting extension of the previous lemma, even though it is not used in the algorithm.

Lemma 3.14. *Let $\Lambda \subseteq \mathbb{R}^n$ be a lattice and let λ be the length of its shortest vector. It holds that*

$$|\Lambda \cap \mathcal{B}(0,k\lambda)| \leq (2k+1)^n \quad \text{for any } k \leq 1.$$

Proof. We can put a ball of radius $\frac{\lambda}{2}$ around each lattice point in $\Lambda \cap \mathcal{B}(0,k\lambda)$ and these balls will not overlap. It also holds that all these balls are contained in $\mathcal{B}(0,k\lambda+\frac{\lambda}{2})$. It holds that:

$$\frac{\operatorname{vol}(\mathcal{B}(0,k\lambda+\frac{\lambda}{2}))}{\operatorname{vol}(\mathcal{B}(0,\frac{\lambda}{2}))} = \frac{\left(k\lambda+\frac{\lambda}{2}\right)^n}{\left(\frac{\lambda}{2}\right)^n} = \frac{\lambda^n\big(k+\frac{1}{2}\big)^n}{\lambda^n\frac{1}{2^n}} = \big(2k+1\big)^n,$$

therefore, there cannot be more than $(2k+1)^n$ lattice points in $\mathcal{B}(0,k\lambda)$. □

3.2.2 The algorithm

The initialization of the algorithm goes as follows:

- First compute $R_0 := n \cdot \max_{i=1,\dots,n} \|b_i\|_2$, where b_i are the columns of the matrix B which generates the lattice $\Lambda(B)$. Set $R := R_0$.
- Sample $N := 2^{8n}\log(R_0)$ random points $x_1, \dots, x_N$ from $\mathcal{B}(0,2)$.
- For each x_i compute $z_i \in \Lambda(B)$.

The computation of z_i can be done in polynomial time. To do so, let's recall the definition of fundamental parallelepiped of $\Lambda(B)$:

$$\mathcal{P}(B) := \{\sum_{i=1}^{n} \lambda_i b_i \mid 0 \le \lambda_i \le 1 \; \forall \, i \in [n]\}.$$

For $x \in \mathbb{R}^n$ we define the remainder of x as $\mathrm{rem}_B(x) \in \mathcal{P}(B)$ such that

$$x - \mathrm{rem}_B(x) \in \Lambda(B).$$

Let's see that we can compute $\mathrm{rem}_B(x)$ in polynomial time.

Lemma 3.15. *For any point $x \in \mathbb{R}^n$, the remainder $\mathrm{rem}_B(x)$ can be computed in polynomial time.*

Proof. For any point $x \in \mathbb{R}^n$, there is a unique choice of λ_i such that $\sum_{i=1}^n \lambda_i b_i = x$, as the b_i are n linearly independent vectors. This linear combination can be computed in polynomial time, for instance using the Gaussian elimination. Then, we have that $\mathrm{rem}_B(x) = \sum_{i=1}^n (\lambda_i - \lfloor \lambda_i \rfloor) b_i$. □

For $i = 1, \dots, N$ we call y_i the remainder of x_i, i.e. $y_i := \mathrm{rem}_B(x_i)$. Then the points $z_i \in \Lambda(B)$ will be computed as $z_i = y_i - x_i$. So, at the beginning of our algorithm we start with a list $(x_1, z_1), \dots, (x_N, z_N)$.

This list satisfies two invariants:

- $z_i \in \Lambda(B)$ for all $i = 1, \dots, N$;
- $\|y_i\|_2 \le R$ for all $i = 1, \dots, N$;

This is true because $z_i \in \Lambda(B)$ for all $i = 1, \dots, N$ by construction, and $\|y_i\|_2 \le R_0$ for all $i = 1, \dots, N$ because $\|y_i\|_2 = \|\mathrm{rem}_B(x_i)\|_2 \le \sum_{j=1}^n \|b_j\|_2 \le R_0$.

The algorithm will modify the list $(x_1, z_1), \dots, (x_N, z_N)$ in order to keep these two invariants valid. The idea is to iteratively subtract the lattice vectors from each other in order to reduce their length. This happens because, since we have taken $N = 2^{8n} \log R_0 >> 5^n$, there must exist some points y_i, y_j such that $\|y_i - y_j\|_2 \leq \frac{R}{2}$ for Lemma 3.11. This implies

$$\|z_i - z_j\|_2 = \|y_i - x_i - (y_j - x_j)\|_2 \leq \|y_i - y_j\|_2 + \|x_j - x_i\|_2 \leq \frac{R}{2} + 4.$$

At the beginning we had

$$\|z_i\|_2 = \|y_i - x_i\|_2 \leq \|y_i\| + \|x_i\| \leq R + 2.$$

Hence putting $z_i := z_i - z_j$, we are shortening the length of z_i.

Notice that we may incur in the problem of having only zero vectors at the end of our algorithm. To avoid this scenario we will introduce the random factor.

The sieving algorithm is the following:

- Initialize a list Z of N points satisfying both invariants for $R = R_0$.
- While $R > 6$:
  - Perform a clustering as described in Lemma 3.11 for $y_i := x_i + z_i$ and call C the set of cluster centers. Call $\sigma(i)$ the index such that $\|y_i - y_{\sigma(i)}\|_2 \leq \frac{R}{2}$.
  - Delete from the initial list Z the points associated with the cluster centers.
  - For each remaining pair, set $z_i := z_i - z_{\sigma(i)}$.
  - Set $R := \frac{R}{2} + 2$.
- Return the shortest non-zero vector among all pairs $z_i - z_j$.

First of all let's prove that the two invariants are maintained:

1. $z_i \in \Lambda(B)$ since we are always adding or subtracting lattice vectors.
2. We have to check that $\|y_i'\|_2 \leq \frac{R}{2} + 2$, where $y_i' = x_i + z_i' = x_i + z_i - z_{\sigma(i)} = y_i - z_{\sigma(i)}$.

$$\|y_i'\|_2 = \|y_i - z_{\sigma(i)}\|_2 \leq \|y_i - y_{\sigma(i)}\|_2 + \|x_{\sigma(i)}\|_2 \leq \frac{R}{2} + 2.$$

Now we explain how we avoid ending only with zero vectors (with high probability). Let's call $v \in \Lambda(B)$ the shortest lattice vector. We define the regions

$$C_1 := \mathcal{B}(0,2) \cap \mathcal{B}(v,2) \quad \text{and} \quad C_2 := \mathcal{B}(0,2) \cap \mathcal{B}(-v,2).$$

We define a 'flipping' map $\tau : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ as follows:

$$\tau(x) = \begin{cases} x+v & \text{if} \quad x \in C_2 \\ x-v & \text{if} \quad x \in C_1 \\ x & \quad \text{otherwise} \end{cases}$$

Observe that in the initialization we take the points x_i uniformly at random from $\mathcal{B}(0,2)$. This procedure is equivalent to choosing the points uniformly at random from $\mathcal{B}(0,2)$ and then, with probability $\frac{1}{2}$ (for instance tossing a coin) flipping each x_i, that is to say, substituting x_i with $\tau(x_i)$. We use this trick only to analyse the algorithm, therefore the fact that we do not know v is not a problem. We imagine to change the algorithm and to perform this 'tossing procedure' exactly before the first time that it actually matters whether x_i is flipped or not. With this in mind, we can prove the following.

Lemma 3.16. *With high probability, the shortest vector v is among the pairs $z_i - z_j$ for some surviving indices i, j.*

Proof. Observe that $\text{rem}_B(x) = \text{rem}_B(x+v) = \text{rem}_B(x-v)$; thus, the y_i do not depend on the 'flipping' procedure. On the other hand, z_i depends directly on x_i; therefore we need to know whether x_i was flipped or not in order to work with z_i. This means that as long as the algorithm can work with the y_i we do not need any information about the flipping of x_i.

In the clustering we work with the y_i, but when we update the list we are performing $z_i' = z_i - z_{\sigma(i)}$, which can be written as $y_i' = y_i - z_{\sigma(i)}$. This means that we need to know the side of $x_{\sigma(i)}$. This is the reason why we get rid of the cluster centers at each iteration. In this way we can go on working with points for which we do not need any information.

We call a point *good* if it belongs to $C_1 \cup C_2$. At the beginning of the algorithm we have at least $2^{6n-1} \log R_0$ good points with high probability. Indeed, thanks to Lemma 3.12, each point in $\mathcal{B}(0,2)$ is good with probability greater than $p := 2^{-2n}$. Thus, the expected number of good points is $pN = 2^{-2n}2^{8n} \log R_0 = 2^{6n} \log R_0$ and the variance of this number is at most pN.

By Chebyschev's inequality[1], the probability that there are less than $pN/2$ points is at most $4/pN$ which is a very small probability.

During the entire algorithm we remove at most $O(\log R_0)2^{5n}$ points, since the number of iterations is $O(\log R_0)$ and in each iteration we remove at most $5^n < 2^{5n}$ points. Thus, when we exit the while-loop we still have more than $(2^{6n-1} - 2 \cdot 5^n) \log R_0 > 2^{5n}$ good points for which we have not decided the side.

Before being tossed, these z_i satisfy $\|z_i\|_2 \leq \|y_i\|_2 + \|x_i\|_2 \leq 6 + 2 = 8$. For Lemma 3.13 it holds that $|\Lambda \cap \mathcal{B}(0, 8)| \leq 2^{4n}$; therefore there exists $w \in \Lambda \cap \mathcal{B}(0, 8)$ such that $z_i = w$ for at least $\dfrac{2^{5n}}{2^{4n}} = 2^n$ points. So, when flipping these points z_i, with probability $\dfrac{1}{2}$ some of them will remain w, and with probability $\dfrac{1}{2}$ some others will become $w + v$ or $w - v$. Therefore, when we take the differences between them, with high probability we will find the shortest vector v.

□

3.2.3 Proofs for ℓ_∞-norm

We want to show that this algorithm works also with the ℓ_∞-norm. In this case it's given a lattice $\Lambda(B)$ with $B \in \mathbb{Q}^{n \times n}$ and the goal is to find a vector $x \in \Lambda(B) \setminus \{0\}$ which minimizes $\|x\|_\infty$. We will see that the results displayed in Sections 3.2.1 and 3.2.2 can be proved equivalently with respect to the ℓ_∞-norm.

First of all, also in this case we can assume to work with lattices $\Lambda(B)$ such that

$$2 \leq SV_\infty(\Lambda(B)) < 3,$$

where $SV_\infty(\Lambda(B))$ denotes the length of the shortest vector with respect to the ℓ_∞-norm. Indeed, if Lemma 3.10 holds for the shortest vector in ℓ_2-norm, then a similar result has to hold for the ℓ_∞-norm. Let's see how we can get some information about the shortest vector in ℓ_∞-norm from the the shortest vector in ℓ_2-norm.

[1]If we have a random variable X with $E[X] = \mu$ and $Var(X) = \sigma^2$ the probability that this variable assumes value in the interval $[\mu - \lambda\sigma, \mu + \lambda\sigma]$ is greater than $1 - \dfrac{1}{\lambda^2}$, where λ is a positive real parameter. In this case we apply this inequality to the random variable X representing the number of good points, $E[X] = pN$, $Var(X) \leq PN$ and we take $\lambda = \dfrac{\sqrt{pN}}{2}$.

We know that

$$\|\cdot\|_\infty \leq \|\cdot\|_2 \leq \sqrt{n}\|\cdot\|_\infty,$$

and from these inequalities we get that

$$SV_\infty(\Lambda(B)) \leq SV_2(\Lambda(B)) \leq \sqrt{n}SV_\infty(\Lambda(B)).$$

Indeed, let's call v_2 the shortest vector in $\Lambda(B)$ with respect to the ℓ_2-norm and v_∞ the shortest vector in $\Lambda(B)$ with respect to the ℓ_∞-norm. We know that

$$\|v_2\|_\infty \leq \|v_2\|_2 \leq \sqrt{n}\|v_2\|_\infty,$$

and

$$\|v_\infty\|_\infty \leq \|v_\infty\|_2 \leq \sqrt{n}\|v_\infty\|_\infty.$$

It also holds that

$$\|v_\infty\|_\infty \leq \|v_2\|_\infty \quad \text{and} \quad \|v_2\|_2 \leq \|v_\infty\|_2$$

So we get

$$\|v_\infty\|_\infty \leq \|v_2\|_\infty \leq \|v_2\|_2 \Longrightarrow SV_\infty(\Lambda(B)) \leq SV_2(\Lambda(B))$$

and

$$\|v_2\|_2 \leq \|v_\infty\|_2 \leq \sqrt{n}\|v_\infty\|_\infty \Longrightarrow SV_2(\Lambda(B)) \leq \sqrt{n}SV_\infty(\Lambda(B)).$$

So, we assume to work with lattices $\Lambda(B)$ such that

$$2 \leq SV_\infty(\Lambda(B)) < 3.$$

We need to see that the other results hold.

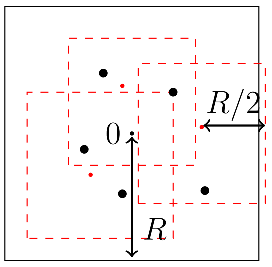
In the proof of Lemma 3.11 we use the definition and the properties of a distance d, which don't change if we are referring to the ℓ_∞-norm instead of the ℓ_2-norm. Furthermore, the ratio between the volume of two balls is the same for any ℓ_p-norm. In particular,

$$V_n^\infty(R) := (2R)^n \quad \text{and} \quad V_n^2(R) := \frac{\pi^{\frac{n}{2}}R^n}{\Gamma(\frac{n}{2}+1)},$$

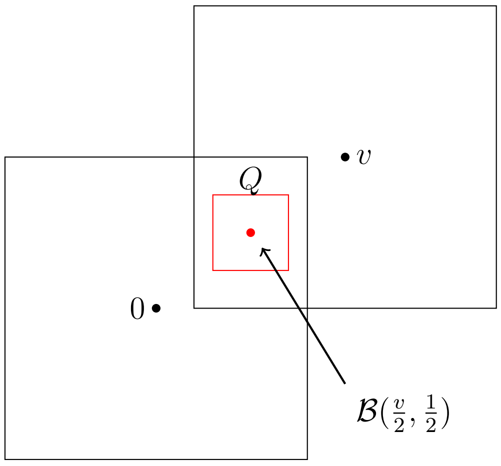
so we have that

$$\frac{V_n^\infty(R_1)}{V_n^\infty(R_2)} = \frac{2^n \cdot R_1^n}{2^n \cdot R_2^n} = \left(\frac{R_1}{R_2}\right)^n \quad \text{and} \quad \frac{V_n^2(R_1)}{V_n^2(R_2)} = \frac{\pi^{\frac{n}{2}}R_1^n}{\Gamma(\frac{n}{2}+1)} \cdot \frac{\Gamma(\frac{n}{2}+1)}{\pi^{\frac{n}{2}}R_2^n} = \left(\frac{R_1}{R_2}\right)^n.$$

Therefore, the proof is the same. Of course we cannot visualize the balls as in the previous case. In the case of the ℓ_∞-norm, the balls can be visualized as squares. An instance is displayed in the following figure.



Concerning Lemma 3.12, we have already proved that $\mathcal{B}(\frac{v}{2}, \frac{1}{2}) \subseteq Q$ using only the triangle inequality which is valid for any distance, including the distance with respect to the ℓ_∞-norm. Then, we use again an argument involving the volume ratio between two balls and we have just seen that the ratio doesn't change if we change the norm. Therefore also this proof is still valid. We can visualize one instance as in the following picture.



For the same reasons, also the proof of Lemma 3.13 is valid with respect to the ℓ_∞-norm.

The algorithm that finds the shortest vector with respect to the ℓ_∞-norm can be defined as follows.

Initialization:

- Compute $R_0 := n \cdot \max_{i=1,\dots,n} \|b_i\|_\infty$, where b_i are the columns of the matrix B which generates the lattice $\Lambda(B)$. Set $R := R_0$.
- Sample $N := 2^{8n}\log(R_0)$ random points $x_1, \dots, x_N$ from $\mathcal{B}(0,2)$ (where this ball is defined with respect to the ℓ_∞-norm).
- For each x_i compute $z_i \in \Lambda(B)$

The observations about the last step of this initialization and the computing of the remainder still hold.

The invariants will be:

1. $z_i \in \Lambda(B)$ for all $i = 1, \dots, N$;
2. $\|y_i\|_\infty \leq R$ for all $i = 1, \dots, N$;

After the initialization, the first one is satisfied by construction; the second one is true because $\|y_i\|_\infty = \|\mathrm{rem}_B(x_i)\|_\infty \leq \sum_{j=1}^n \|b_j\|_\infty \leq R_0$.

The sieving algorithm is the following:

- Initialize a list Z of N points satisfying both invariants for $R = R_0$.
- While $R > 6$:
  - Perform a clustering for $y_i := x_i + z_i$ and call C the set of cluster centers. Call $\sigma(i)$ the index such that $\|y_i - y_{\sigma(i)}\|_\infty \leq \dfrac{R}{2}$.
  - Delete from the initial list Z the points associated with the cluster centers.
  - For each remaining pair, set $z_i := z_i - z_{\sigma(i)}$.
  - Set $R := \dfrac{R}{2} + 2$.
- Return the shortest non-zero vector among all pairs $z_i - z_j$.

The success of this algorithm, following the proofs for the previous one, can be showed using the triangle inequality and Lemmas 3.11, 3.12, 3.13. Therefore this algorithm finds with high probability the shortest vector with respect to the ℓ_∞-norm.

Chapter 4

Ideas for IP in dimension 3

We start seeing how to use the ideas developed in the last sections to solve the problem in dimension 3. Namely, we would like to solve

$$\max\{c^t x \mid Ax \leq b, x \in \mathbb{Z}^3\},$$

where

$$c \in \mathbb{Z}^3,\ A \in \mathbb{Z}^{m\times 3},\ b \in \mathbb{Z}^m.$$

First of all, let's recall Doignon's theorem and a corollary (see [15]).

Theorem 4.1 (Doignon's Theorem). *Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ be a system such that $P \cap \mathbb{Z}^n = \emptyset$. Then, there exists a subset of at most 2^n inequalities which are already integer infeasible.*

Corollary 4.2. *Let $Ax \leq b$ be a system of linear inequalities in n variables, and let $c \in \mathbb{Q}^n$. If $\max\{c^t x \mid Ax \leq b, x \in \mathbb{Z}^n\}$ is finite, then there exists a subsystem of at most $2^n - 1$ inequalities $A'x \leq b'$ such that*

$$\max\{c^t x \mid Ax \leq b, x \in \mathbb{Z}^n\} = \max\{c^t x \mid A'x \leq b', x \in \mathbb{Z}^n\}.$$

This means that the optimum is determined by a smaller subset (whose size cannot exceed $2^n - 1$) of inequalities.

We can use Clarkson's algorithm (see [14]) to reduce our problem to a problem with a fixed number of constraints. Therefore, from now on, we will assume to work with problems with a fixed number of constraints.

Clarkson claims that an integer program defined by m constraints can be solved with $O(m)$ arithmetic operations and $O(\log m)$ calls to an algorithm which solves an integer program defined by a fixed size subset of constraints. Hence, if we can solve the problem with a fixed number of constraints in $O(\phi)$

arithmetic operations, where ϕ is the maximum binary encoding length of the input data, then the original problem with m constraints can be solved in $O(m + \log m\phi)$.

We are given an integer program

$$\max\{c^t x \mid Ax \le b, x \in \mathbb{Z}^3\}.$$

We can assume that $P = \{x \in \mathbb{R}^3 | Ax \le b\}$ is bounded and full-dimensional, and that the objective is to maximize one component, let's say the second.

We are going to partition the polygon P in order to work only with two-layer simplices. A simplex is a full-dimensional polytope $\Sigma \subseteq \mathbb{R}^n$ with $n+1$ vertices. A two-layer simplex is a simplex whose vertices can be partitioned into two sets V and W such that for all $v, v' \in V$ we have $v_2 = v'_2$ and for all $w, w' \in W$ we have $w_2 = w'_2$, namely, the second components of the elements in V and in W agree.

To do this partition, we list the second components of the vertices of P in decreasing order: $\alpha_1, \dots, \alpha_l$. Then, we partition P into the polygons P_i described by:

$$P_i = P \cap \{x_2 \le \alpha_i\} \cap \{x_2 \ge \alpha_{i+1}\} \quad i = 1, \dots, l-1.$$

Thanks to Caratheodory's theorem[1], each P_i can be partitioned into two-layer simplices, which are spanned by the vertices of P_i.

Indeed, for each P_i it holds that

$$P_i = \text{conv}(X),$$

where $X \subseteq \mathbb{Q}^3$ denotes the set of vertices of P_i; this set contains the vertices v of P such that $v_2 = \alpha_i$ or $v_2 = \alpha_{i+1}$ and it may contain some other points arising from the intersection of P with the plane $x_2 = \alpha_i$ or with the plane $x_2 = \alpha_{i+1}$. So, if we take $p \in P_i$, it holds that $p \in \text{conv}(X)$.

By Caratheodory's theorem, there exists a set $S \subseteq X$ of at most 4 points such that $p \in \text{conv}(S)$. This means that for each point $p \in P_i$ there exists a simplex Σ (which will be a two-layer simplex because of the properties of P_i) such that $p \in \Sigma$; equivalently, this means that P_i can be partitioned into two-layer simplices.

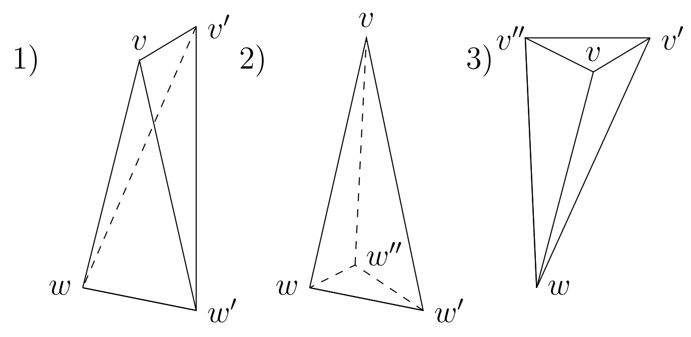
At this point, our problem has become

$$\max\{x_2 \mid x \in \Sigma \cap \mathbb{Z}^3\},$$

[1]If $S \subseteq \mathbb{R}^n$ and $x \in \text{conv}(S)$, then x is the convex combination of at most $n+1$ points in S, see also [15].

where $\Sigma = \{x \in \mathbb{R}^3 \,|\, Ax \leq b\}$, with $A \in \mathbb{Z}^{4\times 3}, b \in \mathbb{Z}^4$.

We can have three cases, which can be visualized as follows:



Let's consider case 1). Similarly to what we have seen for a triangle, it holds that $\Sigma = \text{conv}(v, v', w, w')$. We want to study the width of this simplex, so we can translate it and assume that $v = 0$. Then, we have $\Sigma = \text{conv}(0, v', w, w')$. So,

$$\begin{aligned} w_c(\Sigma) &= \max\{0, c^T v', c^T w, c^T w'\} - \min\{0, c^T v', c^T w, c^T w'\} \\ &= \max\{0, c^T v', c^T w, c^T w'\} + \max\{0, -c^T v, -c^T w, -c^T w'\}. \\ \Longrightarrow & \max\{\left|c^T v'\right|, \left|c^T w\right|, \left|c^T w'\right|\} \leq w_c(\Sigma) \leq 2\max\{\left|c^T v\right|, \left|c^T w\right|, \left|c^T w'\right|\}. \end{aligned} \tag{4.1}$$

Now we define a matrix A_Σ associated with the simplex Σ: $A_\Sigma = \begin{pmatrix} v'^T \\ w^T \\ w'^T \end{pmatrix}$.

The inequalities expressed in the last line of (4.1) can be rewritten as:

$$\|A_\Sigma c\|_\infty \leq w_c(\Sigma) \leq 2\|A_\Sigma c\|_\infty.$$

Taking the minimum over $c \in \mathbb{Z}^3$ yields

$$SV(\Lambda(A_\Sigma)) \leq w(\Sigma) \leq 2SV(\Lambda(A_\Sigma)),$$

where $\Lambda(A_\Sigma)$ is the lattice generated by the matrix A_Σ, namely $\Lambda(A_\Sigma) = \{A_\Sigma x : x \in \mathbb{Z}^2\}$ and $SV(\Lambda(A_\Sigma))$ is the length of the shortest vector in that lattice with respect to the ℓ_∞-norm. With this procedure, we also find a flat direction for Σ. Indeed, we have that the $c \in \mathbb{Z}^3$ such that $v = A_\Sigma c$ is the shortest vector in $\Lambda(A_\Sigma)$, is a flat direction for the simplex Σ.

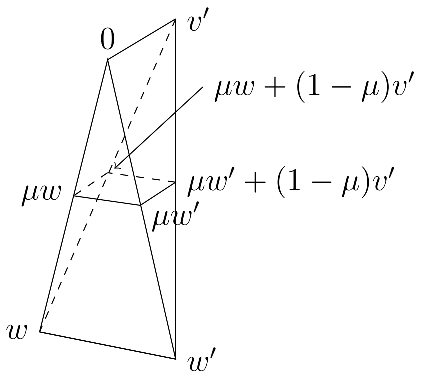
We would like to have an upper bound on the width of Σ. Namely,

$$w(\Sigma) \leq kf(3),$$

with k some constant.

The first thing we are going to check is if $SV(\Lambda(A_\Sigma)) \leq \frac{k}{2}f(3)$. If this inequality holds, than $w(\Sigma) \leq kf(3)$, and so we can solve the problem by solving at most $\lceil kf(3) \rceil$ 2-dimensional sub-problems with a fixed number of constraints, which can be solved in $O(\phi)$ arithmetic operations each, as we have seen in section 2.5.

In the other case, that is to say if $SV(\Lambda(A_\Sigma)) > \frac{k}{2}f(3)$, we don't have an upper bound for the width of Σ. Therefore, we will look for a parameter π such that $f(3) \leq w(\Sigma_\pi) \leq kf(3)$, where $\Sigma_\pi = \Sigma \cap \{x_2 \geq \pi\}$.



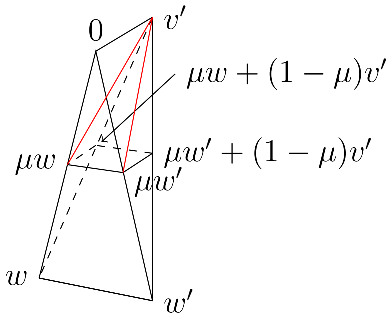
We can see that

$$\Sigma_\pi = \text{conv}(0, v', \mu w, \mu w', \mu w + (1-\mu)v', \mu w' + (1-\mu)v'),$$

where $\mu = \pi/w_2$. If we define $\Sigma'_\pi = \text{conv}(0, v', \mu w, \mu w')$ we can prove that $\Sigma'_\pi \subseteq \Sigma_\pi \subseteq 2\Sigma'_\pi$. More generally, it holds the following.

Lemma 4.3. *Let* $\Sigma = \text{conv}(V \cup W) \subseteq \mathbb{R}^3$ *be a two-layer simplex with* $0 \in V$ *and* $w_2 < 0$ *for all* $w \in W$ *and let* $0 \geq \pi \geq w_2$ *for* $w \in W$. *It holds that the truncated simplex* $\Sigma_\pi = \Sigma \cap \{x_2 \geq \pi\} \subseteq 2\text{conv}(V \cup \mu W)$, *where* $\mu = \pi/w_2$, *for* $w \in W$. *Therefore,*

$$w(\text{conv}(V \cup \mu W)) \leq w(\Sigma_\pi) \leq 2w(\text{conv}(V \cup \mu W)).$$



Let's call $\Sigma_{V,\mu W} := \operatorname{conv}(V \cup \mu W)$. We will study its width. We want to find $\mu \in [0,1]$ such that

$$f(3) \leq w(\Sigma_{V,\mu W}) \leq \frac{k}{2} f(3).$$

If $\Sigma = \operatorname{conv}(0, w, w', w'')$ we have that Σ_π is already a simplex and it is a scaled copy of Σ. In particular it holds that $A_{\Sigma_\pi} = \mu A_\Sigma$. This means that $SV(\Lambda(A_{\Sigma_\pi})) = \mu SV(\Lambda(A_\Sigma))$. So, we simply take $\mu = f(3)/SV(\Lambda(A_\Sigma))$ and we are done. Here, the shortest vector can be computed with the algorithm seen in section 3.2.

In the other cases ($\Sigma = \operatorname{conv}(0, v', v'', w)$ or $\Sigma = \operatorname{conv}(0, v', w, w')$) there is at least one row of A_Σ which remains the same. So we cannot conclude as quickly as in the previous case. We have to look for μ by trials and we need to find a way to bound the number of trials.

Bibliography

[1] Jr. Lenstra, H. W.: Integer programming with a fixed number of variables, Mathematics of Operations Research, 8(4):pp. 538–548, 1983.

[2] Rothvoss, T.: Integer Optimizations and Lattices, University of Washington, Spring 2016.

[3] A. I. Barvinok, A. I.: Polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed, Math. Oper. Res., 19:769–779, 1994.

[4] De Loera, J. and Hemmecke, R. and Köppe, M.: Algebraic and Geometric Ideas in the Theory of Discrete Optimization, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2012.

[5] Eisenbrand, F., Rote, G.: Fast 2-variable integer programming. In: Integer Programming and Combinatorial Optimization, IPCO 2001, K. Aardal, B. Gerards (eds.), vol. 2081 of LNCS, Springer, pp. 78-89, 2001.

[6] Eisenbrand, F., Laue, S.: A linear algorithm for integer programming in the plane. Math. Program.. 102. 249-259, 2005.

[7] Eisenbrand, F.: Short vectors of planar lattices via continued fractions. Inf. Proc. Lett. 79 (3), 121–126, 2001.

[8] Khinchin, A.: Continued Fractions, Dover Publications, 1997.

[9] Chrystal, G.: Algebra - An Elementary Text-Book - Part II, Adam and Charles Black, Edinburgh, 1889.

[10] Micciancio, D. and Goldwasser, S. : Complexity of Lattice Problems: a cryptographic perspective, Kluwer Academic Publishers, Boston, Massachusetts, 2002.

[11] Dadush, N. D. : Integer Programming, Lattice Algorithms, and Deterministic Volume Estimation, H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, May 2012.

[12] Oded, R.: Lecture notes on lattices, 2009.

[13] Ajtai, M., Kumar, R., Sivakumar, D. : A Sieve Algorithm for the Shortest Lattice Vector Problem, Almaden Research Center, San Jose, 2001.

[14] Clarkson, Kenneth L. : Las Vegas Algorithms for Linear and Integer Programming when the Dimension is Small, J. ACM, Vol. 42 (2), pp. 488-499, New York, March 1995.

[15] Schrijver, A. : Theory of Linear and Integer Programming, John Wiley & Sons, Inc., New York, 1986.