

MASARYKOVA UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA
ÚSTAV MATEMATIKY A STATISTIKY

Bakalářská práce

BRNO 2024

VIKTORIE BLAHOVÁ

MASARYKOVA
UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA
ÚSTAV MATEMATIKY A STATISTIKY

LLL a kvadratické formy

Bakalářská práce

Viktorie Blahová

Vedoucí práce: Mgr. Marek Sýs, Ph.D. Brno 2024

Bibliografický záznam

Autor:	Viktorie Blahová Přírodovědecká fakulta, Masarykova univerzita Ústav matematiky a statistiky
Název práce:	LLL a kvadratické formy
Studijní program:	Matematika
Studijní obor:	Obecná matematika
Vedoucí práce:	Mgr. Marek Sýs, Ph.D.
Akademický rok:	2023/2024
Počet stran:	<i>xiv</i> + 49
Klíčová slova:	LLL algoritmus; mřížka; problém nejkratšího vektoru; Gramova matice; Semaeův algoritmus

Bibliographic Entry

Author:	Viktorie Blahová Faculty of Science, Masaryk University Department of Mathematics and Statistics
Title of Thesis:	LLL and quadratic forms
Degree Programme:	Mathematics
Field of Study:	Mathematics
Supervisor:	Mgr. Marek Šýs, Ph.D.
Academic Year:	2023/2024
Number of Pages:	<i>xiv</i> + 49
Keywords:	LLL algorithm; lattice; shortest vector problem; Gram matrix; Semaev's algorithm

Abstrakt

Tato práce řeší problém nejkratšího vektoru mřížky hledáním minima kvadratických forem. Analyzujeme případy, kdy samotné minimum kvadratické formy nestačilo k najetí nejkratšího vektoru. Dále analyzujeme Semaevův exaktní algoritmus pro dimenzi 3 a navrhuje jeho modifikované rozšíření na libovolnou dimenzi. Jednu iteraci této modifikace pak ve zkrácené verzi aplikujeme na vstupní bázi LLL algoritmu a porovnáváme jeho rychlost oproti základní verzi LLL. Naše modifikace zvládla zkrátit počet iterací v dimenzi 20 o 18 %.

Abstract

This thesis is solving the shortest lattice vector problem by finding the minimum of quadratic forms. We analyze cases where the minimum of the quadratic form alone is not sufficient to find the shortest vector. We also analyze Semaev's exact algorithm for dimension 3 and propose a modified extension for an arbitrary dimension. We then apply a single iteration of this modification to the input basis of the LLL algorithm and compare its speed against the base LLL version. Our modification managed to reduce the number of iterations in dimension 20 by 18 %.

ZADÁNÍ
BAKALÁŘSKÉ PRÁCE

Akademický rok: 2023/2024

Ústav:	Ústav matematiky a statistiky
Studentka:	Viktorie Blahová
Program:	Matematika
Specializace:	Obecná matematika

Ředitel ústavu PŘF MU Vám ve smyslu Studijního a zkušebního řádu MU určuje bakalářskou práci s názvem:

Název práce:	LLL and quadratic forms
Název práce anglicky:	LLL and quadratic forms
Jazyk závěrečné práce:	angličtina

Oficiální zadání:

Well-known LLL algorithm can be viewed as greedy algorithm for solving shortest vector problem. The algorithm can be changed so it will work with Gram matrix (of dot products) of basis vectors b_i . Norm of each lattice vector can be expressed as quadratic form with Gram matrix as its associated matrix. For fixed non-zero x_j , it is easy to find analytically shortest vector $v_{\min,j} = (x_1, \dots, x_k) \cdot (b_1, \dots, b_k)^T$. It suffices to minimize quadratic form $(x_1, \dots, x_k) \cdot G(x_1, \dots, x_k)$. Unfortunately, resulted vector $v_{\min}$ is not (in general) part of the lattice. On the other hand this $v_{\min}$ should be close to some small vector of the lattice. The goal of the thesis is to analyze quadratic forms defined by Gram matrix and propose alternative approach to LLL algorithm that will be based on quadratic forms. Student will: 1. generate many random lattices, find exact SVP using enumeration or sieving, and find corresponding coefficients $SV = (X_1, \dots, X_k) \cdot (b_1, \dots, b_k)^T$. 2. analyze distances between $v_{\min,j}$ (for $x_j = X_j$) and SV. 3. based on statistics or information gained from 3. propose alternative approach or tweak of the LLL algorithm that will use quadratic forms.

Vedoucí práce:	Mgr. Marek Sýs, Ph.D.
Datum zadání práce:	3. 10. 2023
V Brně dne:	5. 11. 2023

Zadání bylo schváleno prostřednictvím IS MU.

Viktorie Blahová, 23. 10. 2023

Mgr. Marek Sýs, Ph.D., 30. 10. 2023

RNDr. Jan Vondra, Ph.D., 30. 10. 2023

Poděkování

Na tomto místě bych chtěla poděkovat vedoucímu Mgr. Marku Sýsovi, Ph.D. za cenné připomínky a návrhy a dále Ondřeji Sojkovi za jeho podporu.

Prohlášení

Prohlašuji, že jsem svoji bakalářskou práci vypracovala samostatně pod vedením vedoucího práce s využitím informačních zdrojů, které jsou v práci citovány.

Brno 5. května 2024

.....
Viktorie Blahová

Contents

Introduction	1
Chapter 1. Theory	3
1.1 Matrices and the Gram matrix	3
1.2 Lattices	4
Chapter 2. The LLL algorithm	7
Chapter 3. Alternative approach to the SVP	9
3.1 Lagrange-Gauss algorithm	9
3.2 The cube algorithm	11
Chapter 4. Patterns in failing cases	13
4.1 Generating the failing cases	14
4.2 Orthogonality of the basis	15
4.3 Proximity to a wall	18
4.4 Comparing the closest and smallest vectors of the cube	22
4.5 Descent	24
Chapter 5. Semaev's algorithm	27
5.1 Exact algorithm for 3-dimensional lattice	28
5.1.1 Semaev with rounding	32
5.2 Generalizing for higher dimensions	34
5.2.1 Naive approach	34
5.2.2 Modified algorithm for dimension 4	36
5.2.3 The rounding modification for higher dimensions	38
Chapter 6. The proposed algorithm	41
6.1 Analysis of the LLL algorithm	41
6.2 Semaev-enhanced LLL	43
6.3 Results	43
Conclusion	45
Bibliography	47

Attachments	49
--------------------------	-----------

Introduction

A key area of post-quantum cryptography is lattice-based cryptography. One of the fundamental lattice problems is the shortest vector problem (SVP), which consists of finding the shortest non-zero vector in a given lattice. Exact algorithms for solving SVP have exponential time complexity, so in practice, approximation algorithms are used more often. The state-of-the-art Lenstra-Lenstra-Lovász (LLL) algorithm is an approximation method based on basis reduction.

This work builds upon thesis [12], which introduced an alternative method called the cube algorithm, to find the shortest vector using quadratic forms and omitting the basis reduction. However, in higher dimensions the cube algorithm only manages to find a vector equal to or shorter than LLL in 70 % of the cases [12, p. 55].

The objective of this thesis is to analyze the instances where the cube algorithm fails to find the shortest vector. We generate random matrices and sort them into two categories, based on whether the cube algorithm manages to find the shortest vector. The goal is to attempt to identify common properties of both categories that can predict the probability of the cube algorithm's failure, or to improve the algorithm so that it returns the shortest vector more frequently.

In the second part of the thesis, we present Semaev's algorithm [10], an exact algorithm for dimension 3 that also uses quadratic forms. We analyze this algorithm and provide several generalizations to higher dimensions. Finally, we propose a new algorithm that combines our modification of Semaev's algorithm with the LLL and evaluate its performance.

Chapter 1

Theory

This chapter's purpose is to introduce definitions and theorems used in the work.

1.1 Matrices and the Gram matrix

Definition 1. The Gram matrix of vectors $v_1, \dots, v_n$ is a square matrix G , where the entries of G are defined by the dot products $g_{ij} = v_i \cdot v_j$ for $i, j = 1, \dots, n$.

Given a matrix A , whose rows are vectors, we obtain the corresponding Gram matrix by multiplying $G = A \cdot A^T$. The Gram matrix is symmetric. The Gram matrix serves as a tool to work with norms of the basis vectors in the matrix B . The diagonal $g_{i,i}$ represents the square of the Euclidean norm $\|b_i\|^2$. On the other hand, the non-diagonal elements $g_{ij} = b_i \cdot b_j$ can be represented also as:

$$b_i \cdot b_j = \|b_i\| \cdot \|b_j\| \cdot \cos(\theta),$$

where θ is the angle between the two vectors b_i, b_j . The Gram matrix serves as a useful tool for efficiently accessing the norms of the vectors in matrix A and computing the angles between pairs of these vectors.

Theorem 1 (Gram-Schmidt orthogonalization). Let V be an Euclidean space, $\{u_1, \dots, u_k\}$ basis of its subspace $U \subseteq V$. Then there exists an orthogonal basis $\{v_1, \dots, v_k\}$ such that v_i is in the linear span of $(u_1, \dots, u_k)$ for every $1 \leq i \leq k$.

The proof can be found in [2, p. 201]. Further on, we shall be working with **quadratic forms**. Let us recall the definition:

Definition 2. A mapping $q : U \rightarrow \mathbb{R}$ is called a quadratic form on V if there exists a symmetric bilinear form f on V such that $q(u) = f(u, u)$.

We will work with quadratic forms primarily in their matrix form, i.e., using a symmetric matrix A :

$$q_A(x_1, \dots, x_n) = \sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j = \mathbf{x} \cdot A \cdot \mathbf{x}^T.$$

1.2 Lattices

Let $v_1, \dots, v_n \in \mathbb{R}^n$ be linearly independent vectors. Denote by $L(v_1, \dots, v_n)$ the set of all integer linear combinations of the vectors v_i :

$$L(v_1, \dots, v_n) = \left\{ a_1 \cdot v_1 + \dots + a_n \cdot v_n \mid a_1, \dots, a_n \in \mathbb{Z} \right\}.$$

The vectors $B = \{v_1, \dots, v_n\}$ make up the basis generating the lattice L . It's important to note that a lattice can have multiple different bases. The dimension of the lattice, denoted by $\dim(L)$ or simply n , is defined by the number of vectors in its basis. Familiar concepts from vector spaces, such as the norm, dot product, and angle, also apply to lattices.

In the theory of vector spaces, it is a known theorem that every finite-dimensional vector space has a basis [2, p. 41]. The same property extends to lattices as well. Furthermore, as a result of the Gram-Schmidt process, we know that every finite-dimensional Euclidean space has an orthonormal basis. This naturally leads to the question of whether a similar concept exists within lattice theory. However, as illustrated by Figure 1.1, it becomes apparent that such equivalence generally does not exist. The goal of lattice reduction, which is a core principle used in known algorithms such as LLL or BKZ, is to find a basis that is as short as possible [7].

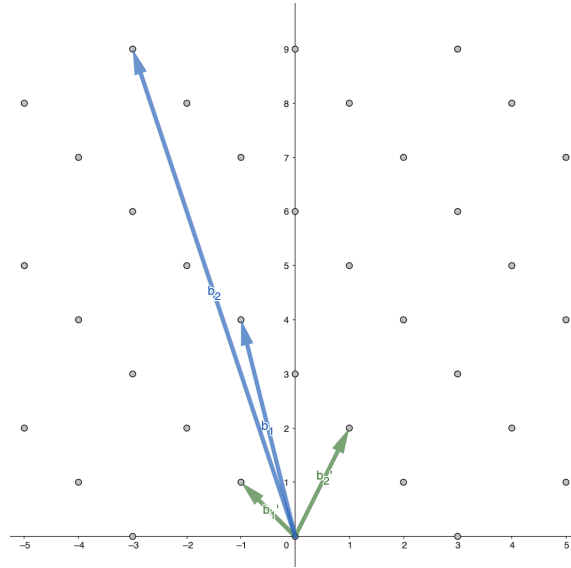


Figure 1.1: Two different bases of the same lattice. The basis (b'_1, b'_2) green contains the shortest possible basis vector.

In the study of lattices, we study so-called lattice problems. The most well-known focus on finding a particular vector within the given lattice. The closest vector problem (CVP for short) aims to find the closest lattice vector to a given point. Assuming a given lattice L and a target vector t , the objective is to find a lattice vector $v \in L$ that minimizes the Euclidean distance $\|v - t\|$.

In contrast, the shortest vector problem (SVP) focuses on finding a non-zero lattice vector, that is, a non-zero vector of minimal Euclidean norm. It can be viewed as a geometric generalization of the greatest common divider computations: the Euclid algorithm technically computes the smallest nonzero linear combination of two integers. The Bézout's identity is formulated as

$$\gcd(a, b) = ax + by \text{ where } a, b \in \mathbb{N}, x, y \in \mathbb{Z}.$$

We can imagine replacing the integers a and b by an arbitrary number of vectors $b_1, \dots, b_n$ with integer coordinates.

In lower dimensions, the shortest vector problem is solvable relatively easily, but with increasing dimension, the problem turns out to be NP-hard [1]. Consequently, there are two basic types of algorithms for solving SVP: exact and approximate. Later, we will focus in-depth on one of the latter, the Lenstra–Lenstra–Lovász (LLL) algorithm. Due to their computational difficulty, these problems have significant applications in lattice-based cryptography, which are further described in [14].

Chapter 2

The LLL algorithm

Let us first note, that it is not the aim of this thesis to give the reader a full understanding of the LLL algorithm. For our purposes, it is sufficient to grasp the basic idea behind it. The Lenstra–Lenstra–Lovász algorithm, first introduced in 1982 solves the approximate form of the SVP in polynomial time [5]. For an input lattice basis B , it computes its Gram-Schmidt orthogonal basis B^* and Gram-Schmidt coefficients $\mu_{ij} = \frac{b_i \cdot b_j^*}{\|b_j^*\|^2}$. The resulting basis B' is so called LLL-reduced, which means it satisfies the following conditions [5, p. 516]:

1. (Size reduction) For every $1 \leq j < i \leq n : |\mu_{i,j}| \leq \frac{1}{2}$, where $\mu_{i,j}$ are the Gram-Schmidt coefficients.
2. (Lovász condition) For $1 < k \leq n : \|b_k^*\|^2 \leq (\delta - \mu_{k,k-1}^2) \cdot \|b_{k-1}^*\|^2$, with a parameter $\delta \in (\frac{1}{4}, 1)$, usually chosen as $\delta = \frac{3}{4}$.

The LLL algorithm guarantees an upper bound for the first vector of the reduced basis b'_1 :

$$\|b'_1\| \leq 2^{\frac{n-1}{4}} \cdot \sqrt[n]{\det(L)}$$

for $\delta = \frac{3}{4}$ [5, p. 517]. This bound does not guarantee that b'_1 is the shortest vector of the lattice, however, it gives us a reasonable approximation. The LLL algorithm is described in a simple form in Algorithm 1 [12, p. 17].

The algorithm takes the input basis $B = (b_1, \dots, b_n)$ and gradually tweaks it toward the reduction conditions. Let us note that the LLL algorithm keeps track of several variables:

- the current lattice basis vectors $b_1, \dots, b_n$,
- the Gram-Schmidt vectors $b_1^*, \dots, b_n^*$, that is a real-valued basis of orthogonal vectors,
- the coefficients of the Gram-Schmidt orthogonalization μ_{ij} ,
- the index k of the vector b_k we are currently reducing.

Algorithm 1 LLL algorithm**Require:** A lattice basis $B = (b_1, \dots, b_n)$, parameter δ

```

1: function LLL( $b_1, \dots, b_n$ )
2:    $k = 2$ 
3:   while  $k \leq n$  do
4:     for  $j$  from  $k - 1$  to 1 do
5:       if not SIZE CONDITION( $k, j$ ) then ▷ uses  $\mu_{k,j}$ 
6:          $b_k \leftarrow b_k = \text{ROUND}(\mu_{k,j}) \cdot b_j$ 
7:         UPDATE GRAM SCHMIDT( $b_1, \dots, b_n$ )
8:       if not LOVÁSZ CONDITION( $k, \delta$ ) then ▷ uses  $b_k^*, b_{k-1}^*, \mu_{k,k-1}$ 
9:          $k \leftarrow k + 1$ 
10:      else
11:        SWAP( $b_k, b_{k-1}$ )
12:        UPDATE GRAM SCHMIDT( $b_1, \dots, b_n$ )
13:         $k \leftarrow \text{MAX}(k - 1, 2)$ 
    return ( $b_1, \dots, b_n$ )

```

The algorithm initiates at $k = 1$ and then loops until k reaches n . That is until all n vectors are reduced. Both the size-reduction and Lovász conditions are checked in each iteration of the algorithm. If one of them is not met, the basis vectors are modified, the tracked variables are updated, and the loop iterates until the vectors form an LLL-reduced basis. [12, p. 16]

Chapter 3

Alternative approach to the SVP

To find the shortest vector of the lattice, the LLL algorithm relies on basis reduction. In Chapter 3.2, we describe the cube algorithm, which offers an alternative method for finding the shortest vector s without the basis reduction. The cube algorithm uses the original non-reduced basis B and computes its Gram matrix G . Let c be the vector of coefficients of an arbitrary vector of the lattice. Then the quadratic form $Q(c) = c \cdot G \cdot c^T$ expresses the norm of vector $c \cdot B$ of the basis B . Using this quadratic form, we will move from considering the shortest vector problem as finding integer coefficients $c \in \mathbb{Z}^n$ to a problem of finding the minimum of the quadratic form Q in $\mathbb{R}^n$. To accomplish this, we will need to fix one of the coefficients to 1. (Otherwise, the solution would always be a zero vector c .)

As an introduction, we first discuss the Lagrange algorithm, as it also fixes one coefficient as 1 and finds an optimal value of the other. The chapter then focuses on the cube algorithm itself.

3.1 Lagrange-Gauss algorithm

The Lagrange-Gauss algorithm aims to transform the original basis (b_1, b_2) of a 2-dimensional lattice to a new basis (b'_1, b'_2) , whose vectors are as shortest as possible and consequently nearly orthogonal. Assuming that $\|b_2\| > \|b_1\|$, the key part consists of constructing a new vector b_2 by subtracting b_1 from b_2 as long as we are still reducing the norm of b_2 . In other words, to find a k such that $b_2 - k \cdot b_1$ is minimal.

One way to reach the minimum is using the derivative, minimizing the function $f(k) = \|b_1 - k \cdot b_2\|^2$. Solving the problem by finding a local minimum, we receive that $k = \frac{b_1 \cdot b_2}{\|b_2\|^2}$. However, this result is not an integer, so we have to round the coefficient k , to ensure that the new vector is still part of the lattice.

$$\begin{aligned}
f(k) &= \|b_1 - k \cdot b_2\|^2 = \|b_1\|^2 - 2k \cdot b_1 \cdot b_2 + k^2 \cdot \|b_2\|^2 \\
f'(k) &= -2b_1 \cdot b_2 + 2k\|b_2\|^2 = 0 \\
k &= \left\lfloor \frac{b_1 \cdot b_2}{\|b_2\|^2} \right\rfloor
\end{aligned}$$

The algorithm iterates this process until the basis vectors (b_1, b_2) cannot be shortened any further and returns the reduced basis (b'_1, b'_2) . The Lagrange-Gauss algorithm can be also seen as an analogy to the Euclidean algorithm for the greatest common divisor.

Algorithm 2 Lagrange-Gauss algorithm

Require: A basis (b_1, b_2) of a two-dimensional lattice L

```

1: function GAUSS( $b_1, b_2$ )
2:   if  $\|b_1\| < \|b_2\|$  then
3:     SWAP( $b_1, b_2$ )
4:   while  $\|b_1\| > \|b_2\|$  do
5:      $k \leftarrow \text{ROUND}(\frac{b_1 \cdot b_2}{\|b_2\|^2})$ 
6:      $r \leftarrow b_1 - k \cdot b_2$ 
7:      $b_1 \leftarrow b_2$ 
8:      $b_2 \leftarrow r$ 
9:   return  $b_1, b_2$ 

```

Example

Let's show an example of the algorithm with a basis $B = ((1, 2), (5, 7))$. Since $\|b_1\| < \|b_2\|$, we need to swap the two vectors, so $b_1 = (5, 7)$ and $b_2 = (1, 2)$. Now computing $k = \left\lfloor \frac{5+14}{5} \right\rfloor = 4$, and the auxiliary variable r gets $r = (1, -1)$, following with update of the basis vectors $b_1 = (1, 2)$ and $b_2 = (1, -1)$. Checking the condition again, now $\|b_1\| = \sqrt{5}$ and $\|b_2\| = \sqrt{2}$. Since $\|b_1\| > \|b_2\|$, we continue with another iteration. Now $k = \left\lfloor \frac{1-2}{2} \right\rfloor = -1$, giving us an auxiliary $r = (2, 1)$ and the basis vectors update to $b_1 = (1, -1)$ and $b_2 = (2, 1)$. We can see that this step essentially only swapped b_1 and b_2 , keeping the norms unchanged. Now, the condition $\|b_1\| > \|b_2\|$ does not hold, hence the algorithm ends and returns a basis $B' = ((1, -1), (2, 1))$, where the shortest vector is $b'_1 = (1, -1)$. Let us note that the vector b'_1 is the result of linear combination $-4 \cdot (1, 2) + 1 \cdot (5, 7)$ of vectors b_1, b_2 of the original basis B .

3.2 The cube algorithm

The idea is to express the norm of any lattice vector $v = c \cdot B$ as a quadratic form and find its non-zero minimum. We find the minimum using real analysis, resulting in real-valued coefficients c_R , and assume that the coefficients of the shortest vector of the lattice are close to c_R . For a given basis $B = (b_1, \dots, b_n)$, the objective is to find a linear combination $v = c \cdot B^T$ such that $c_i = 1$ for some i and the norm $\|v\|$ is minimized. The shortest vector s of the lattice L generated by the basis B can be expressed as

$$s = c_s \cdot B,$$

where c_s is the vector of integer coordinates. Instead of finding the vector s , we shall focus on finding its coefficients c_s . Let $c_R = (x_1, \dots, x_n)$. Assume the quadratic form

$$Q(c_R) = \|c_R \cdot B\| = c_R \cdot B \cdot B^T \cdot c_R^T = c_R \cdot G \cdot c_R^T.$$

We shall be looking for the minimum of this quadratic form, given that we fix a single component of c_R . The vector $c_R = (\dots, 1, \dots)$ is a vector of real-valued coefficients, except the i -th position with value 1.

Assuming the integer coefficients in c_s are close to those in c_R , we declare a c_c , whose components are formed by the ceiling or floor of the values in c_R , as a candidate for the coefficients of the shortest vector. [12]

How to obtain c_R ?

We are looking for the minimum of the norm function, now represented by the quadratic form $Q(c_R)$. Without the loss of generality, fixing the first component of c_R for an arbitrary integer k $c_R = (1, x_2, \dots, x_n)$. The quadratic form then is in the form

$$Q(c_R) = \begin{pmatrix} 1 & x_2 & \dots & x_n \end{pmatrix} \cdot \begin{pmatrix} g_{11} & g_{12} & g_{13} & \dots & g_{1n} \\ g_{21} & g_{22} & g_{23} & \dots & g_{2n} \\ g_{31} & g_{32} & g_{33} & \dots & g_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_{n1} & g_{n2} & g_{n3} & \dots & g_{nn} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

By matrix multiplication we obtain

$$\begin{aligned} Q(c_R) &= 1^2 \cdot g_{11} + 1 \cdot x_2 \cdot g_{21} + \dots + 1 \cdot x_n \cdot g_{1n} + \\ &+ 1 \cdot x_2 \cdot g_{12} + x_2^2 \cdot g_{22} + \dots + x_2 \cdot x_n \cdot g_{2n} + \\ &+ 1 \cdot x_3 \cdot g_{13} + x_2 \cdot x_3 \cdot g_{23} + \dots + x_3 \cdot x_n \cdot g_{3n} + \\ &+ \dots \dots \dots + x_n^2 \cdot g_{nn}. \end{aligned}$$

We shall find the partial derivatives of this expression

$$\frac{\partial(Q(c_R))}{\partial x_i} = 2 \cdot 1 \cdot g_{i1} + 2 \cdot x_2 \cdot g_{i2} + \cdots + 2 \cdot x_n \cdot g_{in,i}.$$

We shall put all of these derivatives equal to 0 as we are looking for extreme in every coordinate. We shall now solve this system of linear equations with respect to variables x_i . By subtracting the first column consisting only of scalars, we obtain

$$\begin{pmatrix} g_{22} & g_{23} & \cdots & g_{2n} \\ g_{32} & g_{33} & \cdots & g_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n2} & g_{n3} & \cdots & g_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = -1 \cdot \begin{pmatrix} g_{21} \\ g_{31} \\ \vdots \\ g_{n1} \end{pmatrix} \quad (3.1)$$

Solving this system gives us the resulting $c_R = (1, x_2, \dots, x_n)$.

How to obtain c_c ?

Going through the components of c_R , we will ceil respectively floor each of the real-valued numbers, creating in total 2^{n-1} new vectors, which create the vertices of a single cell in the whole lattice. We shall call the set of all of them the *cube*. Our final coefficients c_c are simply the coefficients corresponding to the vector s_c with the shortest norm $\|s_c\| = c_c \cdot G \cdot c_c^T$.

As the aim of this thesis is to generate the cases, for which the cube algorithm does not find the shortest vector, we use a modified version of the cube algorithm. The method for generating the failing cases is described in Chapter 4, Algorithm 3. The original version of the cube algorithm along with examples can be found in thesis [12].

Chapter 4

Patterns in failing cases

According to the experiments done in [12], the cube algorithm is able to find the same vector as the LLL algorithm in only about 75 % of cases in dimensions 15 to 20. One of the objectives of this thesis is to identify common characteristics that could predict the algorithm's failure in advance. Furthermore, we explore the possibility of estimating the shortest vector of the lattice when the algorithm yields an incorrect result and we propose a method to compute the shortest vector in these "failure cases".

In Chapter 4.1 we generate random bases (dimensions ranging from 3 to 10) and run both an exact and the cube algorithm. We compare the two results and sort the bases into two categories, based on whether the cube algorithm managed or failed to find the shortest vector of the lattice. On these matrices, we then run experiments to test our hypotheses.

Chapters 4.2 and 4.3 test two certain hypotheses on these two sets of bases. Chapter 4.2 is inspired by an example, where it was difficult to solve a 2-dimensional variant of the closest vector problem, when the initial basis vectors formed an extremely acute angle. We test the prevalence of extreme angles in both sets of failing and non-failing matrices. Chapter 4.3 on the other hand, focuses on those coefficients c_c of the cube algorithm containing a component, that is nearly an integer. It then examines whether rounding this component yields the corresponding component of c_s . The rounding works reliably on bases where the cube yields the correct result and we test the same for the other set of matrices.

We move on to investigate, how the norm decreases in the cube. Chapter 4.4 focuses on analyzing the number of cases where the c_c of the cube algorithm are also the closest coefficients c_{cl} to the exact coefficients c_s . We conclude this part with Chapter 4.5, where we attempt to search for the c_s beyond the cube in its close neighborhood.

Although the information gained in these chapters does not provide us with a tool to reliably determine, whether the cube algorithm will manage to find the shortest vector beforehand, we have discovered, that the presence of an extreme angle in the base increases the probability of failure roughly tenfold.

4.1 Generating the failing cases

For the purpose of this chapter, we have generated various numbers of random bases with dimensions 3, 4, 6, and 10 and verified whether the cube algorithm successfully identifies the shortest vector. We compare the cube's coefficients c_c with the coefficients c_s of the exact shortest vector found by SageMath's built-in function `shortest_vector()`. Because the computation time of the exact algorithm has exponential time complexity, we have not generated examples for higher dimensions.

In this chapter, the method of generating the random matrices B involves simply generating a random integer within the range $\langle -1000, 1000 \rangle$ for each of its components and ensuring that $\text{rank}(B) = \text{dim}(B)$.

Subsequently, we move on to assert whether the cube algorithm finds the shortest vector for a particular instance of a randomly generated matrix B . We first compute its exact coefficients c_s and check, whether at least one of the coefficients is ± 1 . Had we selected any other integer, we would be searching for the shortest vector on a proper sublattice, and we would frequently obtain failing results.

After the check, we save the indices of the 1s and -1s. For each of these indices i , we solve the system of linear equations 3.1 with x_i fixed as 1, obtaining the real-valued minimum c_R . Subsequently, we generate all 2^{n-1} integer neighbors of c_R by rounding its coefficients up and down to the nearest integers. We compute the Euclidean norms of these neighbors and store the coefficients of the vector with the smallest norm as c_c .

In the final step, we verify whether c_c is equal to the exact coefficients c_s up to a factor of -1 . If the equality holds, the cube algorithm has successfully found the shortest vector. Otherwise, we proceed to the next index i . If c_s does not equal c_c in any iteration, we conclude that the cube algorithm has failed to find the shortest vector.

Algorithm 3 Assert whether the cube algorithm works on a matrix**Require:** matrix $B = (b_1, \dots, b_n)$

```

1: function ASSERT CUBE( $B$ )
2:    $s \leftarrow \text{SHORTEST VECTOR}(B)$   $\triangleright$  SageMath function shortest_vector()
3:    $c_s \leftarrow s \cdot B^{-1}$   $\triangleright$  Finds the exact coefficients in basis  $B$ 
4:   if (1 or -1) not in  $c_s$  then
5:     return None  $\triangleright$  We only test matrices with coefficient  $c_i = \pm 1$ 
6:    $G \leftarrow \text{GRAM MATRIX}(B)$ 
7:    $\text{unit\_indices} = \{i \mid c_{s,i} = \pm 1\}$ 
8:   for  $i$  in  $\text{unit\_indices}$  do
9:      $c_R = \text{FIND REAL MINIMUM}(G, i)$   $\triangleright$  Solves the system 3.1
10:     $c_c = \text{SHORTEST COEFF IN CUBE}(c_R, B)$ 
11:    if  $c_c = \pm c_s$  then
12:      return True  $\triangleright$  Cube finds exact sv
13:  return False  $\triangleright$  Cube doesn't find exact sv

```

4.2 Orthogonality of the basis

In certain cases a pair of basis vectors form an angle that is either too acute or too obtuse. Without orthogonalization, it may be more difficult to solve the lattice problems for such bases. Our approach was inspired by an example from [8], although it is focused on the closest vector problem. Let us consider a lattice L defined by two different bases B_1 and B_2 .

$$B_1 = \begin{pmatrix} 5 & 1 \\ -2 & 8 \end{pmatrix} \quad \text{and} \quad B_2 = \begin{pmatrix} 37 & 41 \\ 103 & 113 \end{pmatrix}$$

Let us note that the angle between the vectors in B_1 is approximately 92° , while the angle of the basis vectors in B_2 is approx. 0.29° . We shall look for a vector u of the lattice L closest to the point $[27, 8]$. The vector u can be expressed as $u = a_1 \cdot b_1 + a_2 \cdot b_2$, where a_1, a_2 are integer coefficients and b_1, b_2 are basis vectors. Using the basis B_1 , we get the system of equations

$$\begin{aligned} 5a_1 - 2a_2 &= 27 \\ 1a_1 + 8a_2 &= 8 \end{aligned} \quad \text{yielding} \quad \begin{aligned} a_1 &\cong 5,524 \\ a_2 &\cong 0,310 \end{aligned}$$

Simply rounding this result gives us the coefficients $(a_1, a_2) = (6, 0)$, which corresponds to a lattice vector $u = (30, 6)$, which indeed is the closest vector of the lattice L to the point $[27, 8]$. However, if we apply the same process on the second basis B_2 of the same lattice L , we get a

$$\begin{cases} 37a_1 - 103a_2 = 27 \\ 41a_1 + 113a_2 = 8 \end{cases} \Rightarrow \begin{cases} a_1 = 0.461 \\ a_2 = -0.970 \end{cases}$$

The resulting coefficients $a_1, a_2 = (0, -1)$ give us the presumed closest vector $u = (-103, -113)$, which is obviously not close at all to the point $[30, 6]$.

The question is whether this is just an isolated case, or if we could attempt to demonstrate some general conclusions. In the case of a short and nearly orthogonal basis, there was no issue finding the closest vector. On the contrary, with a basis having an extremely small angle between its vectors, this simple method led us completely off course. We are searching for an answer to several questions:

- Does the presence of an extreme angle in the basis imply that the cube algorithm will fail to reliably identify the shortest vector? We expect the probability of it to be significantly high.
- Conversely, if the cube algorithm couldn't find the shortest vector, does this mean, that the basis contains a pair of vectors with an extreme angle?

Methods and results

n	$c_c = c_s$			$c_c \neq c_s$		
	at least one extreme angle	no extreme angles	% of matrices with extreme angles	at least one extreme angle	no extreme angles	% of matrices with extreme angles
3	82	9868	0.8 %	45	513	8 %
4	1	976	0.1 %	0	250	0
6	0	94	0	0	872	0
10	0	16	0	0	23	0

Table 4.1: Prevalence of extreme angles.

After testing several possibilities, we have settled on a benchmark angle $\alpha = 5^\circ$. Greater α might be colluding with our vague idea of "an extreme" angle. However, for a lower value of α , we would need to generate a much larger amount of random bases, as these angles are quite rare.

We have worked with sets of bases of dimensions 3, 4, 6, and 10 and divided them into two sets depending on whether the coefficients returned during the cube algorithm, c_c equaled the coefficients c_s of the shortest vector in the lattice. For both of these sets, we have examined the number of matrices containing at least one angle either less than α or greater than $180^\circ - \alpha$. The findings are presented in Table 4.1.

In dimensions 4, 6, and 10, we did not observe any instances of extreme angles in either set of matrices. This is likely due to the smaller number of generated

cases. In dimension 3, we found an extreme angle in 0.8 % of non-failing matrices (i.e., matrices where the cube algorithm finds the shortest vector) and 8 % of failing matrices.

The data suggests that extreme angles indeed occur approximately 10 times more frequently in failing matrices; however, they are far from being a sufficient condition for classifying a matrix as failing.

4.3 Proximity to a wall

In this chapter we test the hypothesis, that whenever there is a coefficient c_i of c_R such that c_i is *nearly* an integer (e.g., 0.999999 or 0.000001), it would be sufficient to simply round c_i to obtain the corresponding component for the exact coefficients c_s . To test this hypothesis, we concentrate on cases where at least one component of the c_R is *close to* an integer. We might consider that simply rounding the value up or down could lead us to the correct integer value in the c_s .

From a geometric perspective, this scenario corresponds to a situation where the cube point is in close proximity to a face of the cube. We would anticipate c_R to lie on this face, or at least in the same direction. This property could potentially halve the number of possibilities for where the exact coefficients lie.

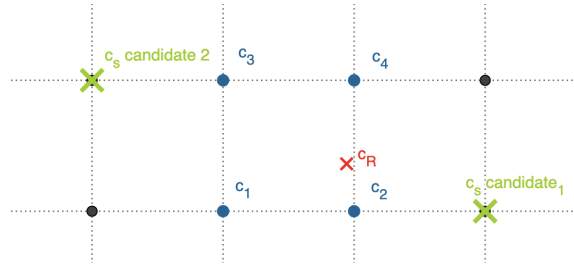


Figure 4.1: The grid of coefficients, c_R marked in red, cube coefficients marked in blue. In this 2-dimensional example, we would expect that since the real-valued coefficients c_R are close to the right wall of the cube, the coefficients c_s of the shortest vector would more likely be on the right (candidate 1) than in the opposite direction (candidate 2).

Methods and results

We have defined several benchmarks $\varepsilon \in \{0.05, 0.005, 0.0010, 0.0005, 0.0001\}$.

Then for each component c_{Ri} of c_R , we decide whether it is "almost an integer" by checking $|\lfloor c_{Ri} \rfloor - c_{Ri}| < \varepsilon$. Then, it simply verifies whether the rounded number is equal to the corresponding component of c_s . We have decided to add another aspect of the analysis: whether the "direction" of the rounding at least points to the "direction" of the correct coefficients c_e . For example, if we round up, whether it holds that $c_{si} > c_{Ri}$. We check using the directional condition *dir*:

$$dir : |c_{Ri} - c_{si}| > |\lfloor c_{Ri} \rfloor - c_{si}|. \quad (4.1)$$

The results of our tests are presented in Table 4.2. If no cases were found for a given ε , it was omitted from the results. In lower dimensions 3 and 4, the rounding success rate for $\varepsilon = 0.05$ is very low, on the order of single-digit percentages, and for other ε values, we did not find enough examples. In dimensions 6 and 10, the rounding success rates for $\varepsilon = 0.5$ were 30.7 % and 40.3 %, respectively, which is still not a certainty.

ε	$c_{Ri} = c_{si}$	$c_{Ri} \neq c_{si}$	SR	dir True	dir True or $c_{Ri} = c_{si}$
dimension 3					
0.05	0	157	0 %	139	88.5 %
dimension 4					
0.05	4	98	3.9 %	73	75.4 %
0.005	0	10	0 %	8	80 %
dimension 6					
0.05	16	36	30.7 %	25	78.8%
0.005	0	3	0 %	3	100 %
dimension 10					
0.05	131	194	40.3 %	119	76.9%
0.005	15	19	44.1 %	11	76.4%
0.001	4	4	50 %	2	75 %
0.0005	1	1	50 %	0	50 %
0.0001	0	1	0 %	0	0 %

Table 4.2: Results of testing the rounding hypothesis on bases containing a component c_{Ri} close to an integer, where the cube algorithm **fails to** find the shortest vector. The fourth column presents the percentage of bases with components for which rounding successfully yielded $c_{Ri} = c_{si}$. The fifth column shows the number of bases satisfying the directional condition 4.1. The final column displays the percentage of bases for which either rounding succeeded or condition 4.1 holds.

As for the analysis of the rounding direction, the success rate was relatively high, the highest being in dimension 3 at 88.5 %. In dimensions 6 and 10, the success rate was on the order of 60-70 %. Overall, in only 23 % of the total number of cases in dimension 10, the rounding occurred in the wrong direction. For $\varepsilon = 0.005$ and dimension 10, the results remain roughly similar, but for lower ε values, we did not have enough generated cases to draw any conclusions.

We can conclude that in around 75 % of cases, the rounding of a nearly-integer coefficient c_{Ri} leads to the correct direction where the exact coefficient c_{si} lies.

Rounding functioning matrices

It is also worth looking at our assumption from the other point of view. Let us have a basis B such that the cube algorithm does find the shortest vector correctly.

The results of the same analysis run of functioning matrices can be seen in Table 4.3. In this case, the success rates of rounding are nearing 100 % in all dimensions and values of ε .

Limits	$c_{Ri} = c_{si}$	$c_{Ri} \neq c_{si}$	SR
	dimension 3		
0.05	212	1	99.5 %
0.005	24	0	100 %
0.001	3	0	100 %
0.0005	3	0	100 %
0.0001	1	0	100 %
	dimension 4		
0.05	24	2	92.3 %
0.005	2	0	100 %
0.001	1	0	100 %
	dimension 6		
0.05	427	15	96.6 %
0.005	46	1	97.8 %
0.001	11	0	100 %
0.0005	6	0	100 %
	dimension 6		
0.05	54	7	88.5 %
0.005	5	2	71.4 %
0.001	2	0	100 %

Table 4.3: Results for components c_{Ri} that are close to an integer, varying dimensions. Run on matrices for which the cube algorithm **does** find the shortest vector of the lattice.

4.4 Comparing the closest and smallest vectors of the cube

As in the previous chapter, where we experimented with rounding and its direction, we now attempt to deduce information about the direction in which the coefficients c_s lie relative to c_R . In this chapter, we ask the question of whether coefficients of the shortest vector of the cube c_c are also the closest coefficients c_{cl} to the coefficients c_s . The optimal result would be for the shortest and closest vectors to be equal in all cases.

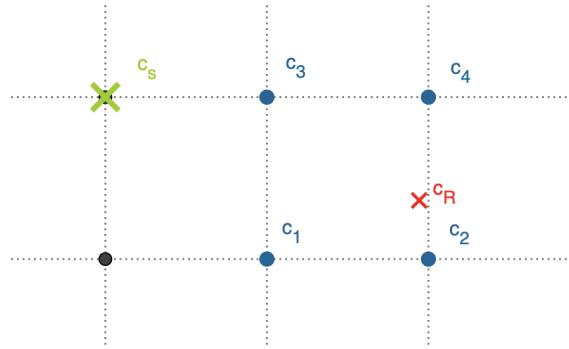


Figure 4.2: The grid of coefficients, c_R marked in red, cube coefficients marked in blue. The coefficients c_3 are closest one to the exact coefficients c_s . The question is whether $c_3 \cdot B$ is the shortest vector of the cube.

Methods and results

The test itself computes the two vectors in question. The shortest vector of the cube is computed via the cube algorithm. The closest combination c_{cl} can be determined by comparing each component of c_R and c_s and rounding it depending on whether $c_{R,i} > c_{s,i}$ or vice-versa.

Assuming a uniform distribution, the probability of a single set of coefficients in an n -dimensional cube possessing a unique property is $\frac{1}{2^{n-1}}$. For instance, in a 3-dimensional cube, the probability is 25 %. If the results manage to show that the measured probabilities of $c_c = c_{cl}$ are higher than this uniform distribution benchmark, we might call the result a partial success.

We ran the experiment on the generated bases of varying dimensions with the following results in Table 4.4. The outcomes surpassed the benchmark in nearly all dimensions, with the exception of dimension 10, where the sample size was insufficient. In the case of dimension 6, considering that the cube has 32 potential coefficient vectors, a significant contrast is evident, as the c_c have a 4.5-fold higher probability of being the closest coefficients to c_s . Hence, the closest vector is more likely to be closer to exact coefficients, than if the closeness had been an uniformly distributed property.

n	$c_c = c_{cl}$	$c_s \neq c_{cl}$	% of $c_c = c_{cl}$	benchmark $\frac{1}{2^{n-1}}$	relative comparison
3	176	387	31.3 %	25 %	1.25
4	60	190	24 %	12.5 %	1.92
6	125	747	14.3 %	3.125 %	4.5
10	0	23	0 %	0.19 %	0

Table 4.4: Results of testing the hypothesis $c_c = c_{cl}$. The last column shows the ratio between the success rate of $c_c = c_{cl}$ (fourth column) and the benchmark success rate (fifth column), and the ratio of both rates (last column).

Nearest norms of c_R

To get a picture of how the norm function behaves in the neighbourhood of c_R , we have plotted the norms of the neighbouring coefficients for two matrices B , for which the cube algorithm fails to find the shortest vector.

The grid we see in Figure 4.3 is not the lattice itself, but the grid of coefficients. These visualizations have been generated from 3-dimensional matrices B , as the quadratic form we minimize is then represented by a function of two variables and is easy to plot. For higher-dimensional matrices, we would have to omit at least one of the components for this process and we might lose some valuable information. In the figure, we might notice a "valley" of relatively low norms, that is most likely a consequence of the quadratic form of the norm function, which results in a parabolic cylinder when plotted in three dimensions.

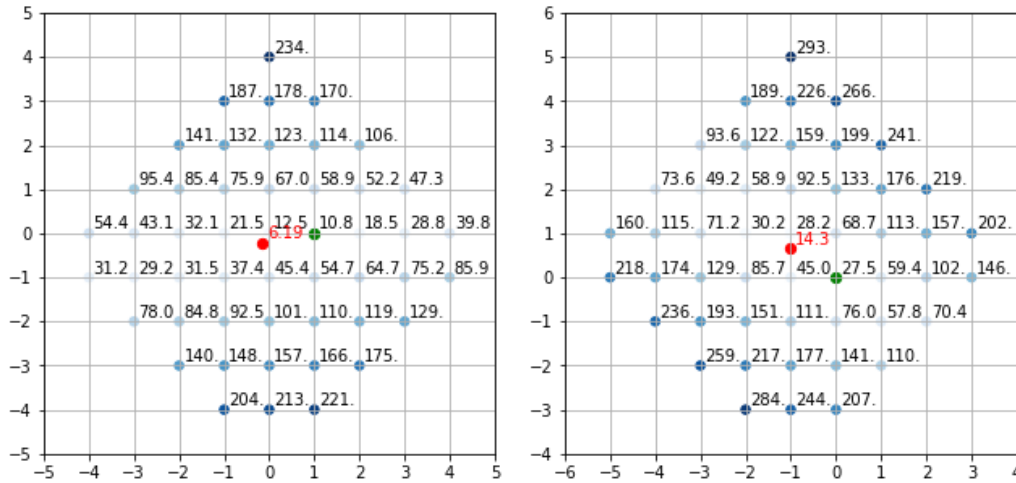


Figure 4.3: The grid of coefficients close to c_R and norms of the corresponding vectors. The real-valued c_R and its norm are marked in red and do not lie on the grid. Marked in green, we can see the coefficients found by the exact algorithm. The blue dots are the coefficients of the nearby lattice vectors. The four closest to the red c_R are the cube vectors whose values we process during the cube algorithm. Generated on matrices of dimension 3.

4.5 Descent

The Figure 4.3 suggests that even though the exact coefficients c_s may not lie directly in the cube, we might look for it in the “valley” hinted in the figure. The rounding experiment in Chapter 4.3 gave satisfying results regarding the approximate direction of the exact coefficients c_s from the c_R and the hypothesis about distance in Chapter 4.4 gave us a hint, that the c_s is more likely to lie in the direction of the shortest vector of the cube.

We shall combine the two approaches in the following Descent algorithm with the pseudocode in Algorithm 4. The algorithm also tries to find the shortest vector without reduction of the basis, but rather descending towards the shortest vector by gradually going through the points of the cube.

In simplicity, the algorithm starts at the c_R , finds the coefficient corresponding to the minimum of the cube c_c and selects these coefficients as a new center c_{center} . Subsequently, it generates coefficients *bigCube* nearby the center¹ and finds those coefficients $c_{BC\ min}$, which correspond to the shortest vector among them. It tests, whether the norm of $c_{BC\ min} \cdot B$ is shorter than the norm of $c_{center} \cdot B$. If so, it sets $c_{BC\ min}$ as a new center and the algorithm loops until it cannot find a shorter vector. The process is described in Algorithm 4.

¹The *bigCube* is generated as a set of all possible additions and subtractions of 1 for every coordinate of the c_{center} coefficients.

Algorithm 4 Descent Algorithm**Require:** matrix B , coefficients of the cube c_R

```

1:  $c_{center} \leftarrow \text{SHORTEST COEFF IN CUBE}(c_R, B)$ 
2:  $norm_{min} \leftarrow c_{center} \cdot B$ 
3: loop
4:    $bigCube \leftarrow \text{GENERATE BIG CUBE POINTS}(c_{center})$ 
5:    $c_{BC\ min} \leftarrow \text{FIND SHORTEST}(bigCube, B)$ 
6:    $norm_{BC\ min} = ||c_{BC\ min} \cdot B||$ 
7:   if  $norm_{BC\ min} \leq norm_{min}$  then
8:      $norm_{min}, c_{center} \leftarrow norm_{BC\ min}, c_{BC\ min}$ 
9:   continue
10: return  $c_{center}$ 

```

▷ Cannot descend further

Results

We have tested the Algorithm 4 on the same failing bases as in Chapter 4.4. Even though the shortest coefficients were in the correct "direction" in 31.3 % of matrices of dimension 3 in the previous chapter (Table 4.4), the descent algorithm found the shortest vector in only 23 %. The success rate steadily decreased in higher bases, reaching 0 % at dim 10, probably due to the lack of examples.

The Algorithm 4 examines at most 2^n possible new points at each iteration at line 5 and does not have a promising success rate. For these reasons, we have not examined the descend approach any further.

dimension	$center = c_s$	$center \neq c_s$	SR
3	132	431	23 %
4	44	206	17.6 %
6	97	775	11.1 %
10	0	23	0 %

Table 4.5: Number of bases for which the descent algorithm's result $center$ was equal to the shortest vector's coefficients c_s . The last column shows the success rate for each dimension.

Chapter 5

Semaev's algorithm

Although the cube algorithm was not able to efficiently find the shortest vector, the same principles are used by Semaev, in his article [10]. Semaev introduces an algorithm for dimension 3 that also reduces the basis, in the sense described in [10]. Semaev's algorithm computes the shortest vector of the cube c_c , but this time replaces the longest vector of its current basis with vector $c_c \cdot B$ and iterates this procedure until the basis can no longer be shortened. Semaev even showed that this algorithm in dimension 3 indeed finds the shortest vector of any basis. In this chapter, we first study the 3-dimensional case and use the gained information to modify the algorithm for higher dimensions.

In Chapter 5.1, we present Semaev's algorithm and analyze its performance on various types of matrices of dimension 3. In Section 5.1.1, we propose a modified version of this algorithm, which, although it does not guarantee finding the shortest vector, significantly reduces the computational complexity by omitting over 2^{n-1} arithmetic operations.

In Chapter 5.2, we discuss strategies for extending Semaev's algorithm to dimensions greater than 3. The main result of this chapter is Algorithm 5.1.1, that although it cannot find the shortest vector reliably in higher dimensions, it is sufficiently fast and serves as a foundation for the algorithm we propose in Section 6.2.

Parameters of the experiments

The matrices used in the previous chapter were unsuitable for the following experiments, as Semaev's algorithm consistently took only 2-3 iterations for an arbitrarily large range of input values. For this reason, we decided to use matrices of the form

$$\begin{pmatrix} d_1 & 1 & 0 & \dots & 0 \\ d_2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{n-1} & 0 & 0 & \dots & 1 \\ d_n & 0 & 0 & \dots & 0 \end{pmatrix}$$

where d_i are randomly generated integers from range $(-10^k, 10^k)$ for various values of k . We shall call these matrices the d -matrices and reference the values of the first column as d_i .

These matrices are derived from the subset-sum problem [9] and are designed to present particularly challenging instances for basis reduction algorithms.

5.1 Exact algorithm for 3-dimensional lattice

The Semaev's algorithm pseudo-code can be seen in the Algorithm 5.1. The algorithm is proven to find the shortest vector and a reduced basis (in the sense described in [10]) for matrices of the form $B = (b_1 \ b_2 \ b_3)^T$.

Semaev's algorithm works with basis B ordered such that $|b_1| \leq |b_2| \leq |b_3|$. The two shortest vectors are reduced using the Lagrange-Gauss algorithm and used to reduce the remaining longest vector b_3 by finding the real-valued minimum of the expression

$$|b_3 + x_2 b_2 + x_1 b_1|$$

over the variables x_1, x_2 . The minimum is computed using the cube algorithm described before. The real-valued minimum, respectively its coefficients $c_c = (x_1, x_2, 1)$ are used to find the shortest of its close integer coefficients, thus one of the combinations.

$$(\lfloor x_1 \rfloor, \lfloor x_2 \rfloor, 1), (\lfloor x_1 \rfloor, \lceil x_2 \rceil, 1), (\lceil x_1 \rceil, \lfloor x_2 \rfloor, 1), (\lceil x_1 \rceil, \lceil x_2 \rceil, 1)$$

For each of these four coefficients, it calculates the norm of its corresponding vector and chooses the shortest one to replace the vector b_3 . It re-orders the basis by size and repeats the process as long as the largest vector can still be shortened.

Algorithm 5 Semaev for matrices of dimension 3, original version from [10]**Require:** B matrix of a basis, dimension n , its vectors are sorted by norm

```

1: function SEMAEV3( $B = (b_1, \dots, b_n)$ )
2:   loop
3:      $b_1, b_0 \leftarrow \text{GAUSS}(b_1, b_0)$ 
4:      $G \leftarrow \text{GRAM MATRIX}(B)$ 
5:      $c_R \leftarrow \text{REAL MINIMUM}(G, n - 1, 1)$   $\triangleright$  Fixes the last vector to its
1-multiple
6:      $\text{cube\_shortest\_lc} \leftarrow \text{SHORTEST LC IN CUBE}(B, c_R)$ 
7:      $\text{candidate} \leftarrow (\text{cube\_shortest\_lc}) \cdot B$ 
8:      $\text{longest\_norm} \leftarrow g_{n,n}$ 
9:     if  $\|\text{candidate}\| > \text{longest\_norm}$  then
10:      break
11:      $b_n = \text{candidate}$ 
12:      $B \leftarrow \text{SORT BY NORM}(B)$ 
13:   return  $B$ 

```

Line	Operation	Multiplications	Additions	Divisions
3	Gauss reduction	19 per it	11 per it	1 division per it
4	Gram matrix	27	18	-
5	Real minimum	4	4	3
6	Shortest lc in cube	48	36	
7	candidate	9	6	
9	$\ \text{candidate}\ $	3	2	
1	sort by norm	9	6	
Total		$19 \cdot it + 100$	$11 \cdot it + 72$	$it + 3$

Table 5.1: Number of individual arithmetic operations of the Semaev algorithm.

Results

We have examined how many iterations does the Semaev's algorithm take for various inputs, and how fast does it reduce the lengths of the basis vectors. As it can be seen in Table 5.2, the average number of iterations gradually increases evenly with increasing range for d_i . The average number of iterations for range $(-10^k, 10^k)$ is approximately $1.8 \cdot k$.

	number of iterations			
range for d_i	average	min	max	exponent:iterations
10^5	4.62	3	7	1.11
10^{10}	6.95	5	9	1.69
10^{15}	9.2	5	13	1.80
10^{20}	11.16	7	15	1.78
10^{25}	13.72	8	18	1.87
10^{30}	15.66	10	21	1.94
10^{35}	18.96	15	25	1.77
10^{40}	21.28	15	28	1.86

Table 5.2: Number of iterations of the Semaev's algorithm depending on the range of d_i . The last column is the ratio of the exponent of the range to the average number of iterations. The results of each line were computed on 100 random matrices.

However, this gives us little to no information about how complex each of these iterations is. In Table 5.1 we have computed how many arithmetic operations are performed by every major part of the algorithm. The two longest ones are the Gauss reduction on line 3 and determining the shortest vector in the cube on line 6. The latter is especially problematic, as the number of computations depends exponentially on the dimension of the basis, as it has to go through 2^{n-1} coefficients of the cube. When we want to optimize or generalize Semaev's algorithm, it will be effective to focus on the exponentially-hard part in particular.

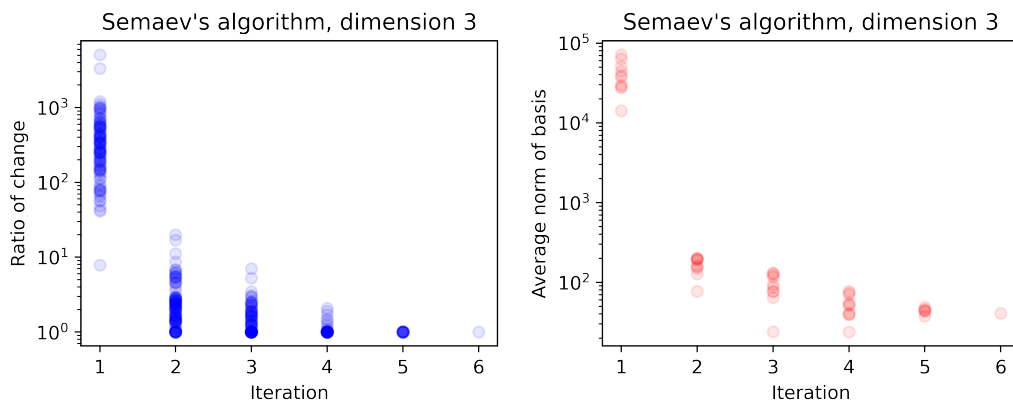


Figure 5.1: Left: The ratio q of the norms of the shortest vectors of two consecutive iterations of the Semaev's algorithm. Computed on 100 matrices of dimension 3 with range for d_i being 10^5 .

Right: Average norms of the basis during the run of the Semaev's algorithm Computed on 10 matrices of dimension 3 with range for d_i being 10^5 .

Another interesting metric might be the rate of shortening during the algorithm's lifetime. Does the shortest norm degrade rapidly, or gradually? The same can be said about the average norm of the basis vectors. It is far more informative to study the rate of shortening, rather than the length of the shortest vector itself.

The graph on the left in Figure 5.1 shows the ratio $\varrho = \frac{\|b_1\|}{\|b'_1\|}$, indicating the relative change of the norm of two shortest vectors b_1, b'_1 of two bases of consecutive loops of Semaev's algorithm. The graph shows that the rate of change in the first iteration ranges from 10^2 to 10^3 , however, in the subsequent iterations, the rate of change is much lower, ranging from 1 to 10. The second graph in Figure 5.1 also shows that the change is most rapid in the first iteration, average norm dropping from approx. 10^5 to 10^2 , and the later iterations being much less effective in terms of reducing the size. The high values of both metrics during the first iterations suggest that the first iteration of Semaev's algorithm is the most significant in terms of reducing the size of the basis.

5.1.1 Semaev with rounding

The most straightforward solution to the computationally exhaustive step on line 6 of the original Semaev's Algorithm 5.1 is, to simply not search for the shortest vector in each case. As we've found out in Chapter 4.4, all of the vectors in the cube are relatively short; thus, the basis should gradually reduce in length, although this may require additional iterations. However, the complexity of each iteration would be simpler, as we would eliminate the exponentially complex step. It may suffice to *approximate* the shortest vector in each step, rather than determining it precisely. The downside of this approach is, that we lose the guarantee of finding the exact shortest vector in dimension 3.

There are several options available on how to find these approximations: for example flooring, ceiling, or rounding the values of the real-valued coefficients $(x_1, x_2, 1)$. The algorithm implemented here simply rounds the components to the closest integer.

Algorithm 6 Semaev with rounding

Require: B matrix of a basis, dimension n , its vectors are sorted by norm

```

1: function SEMAEV_ROUND( $B = (b_1, \dots, b_n)$ )
2:   loop
3:      $b_1, b_0 \leftarrow \text{GAUSS}(b_1, b_0)$ 
4:      $G \leftarrow \text{GRAM MATRIX}(B)$ 
5:      $\text{real\_lc} \leftarrow \text{REAL MINIMUM}(G, n - 1, 1)$        $\triangleright$  Fixes  $b_n$  to its 1-multiple
6:      $\text{candidate} \leftarrow \text{ROUND}(\text{real\_lc}) \cdot B$ 
7:      $\text{longest\_norm} \leftarrow g_{n,n}$ 
8:     if  $\|\text{candidate}\| > \text{longest\_norm}$  then
9:       break
10:     $b_n = \text{candidate}$ 
11:     $B \leftarrow \text{SORT BY NORM}(B)$ 
12:  return  $B$ 
```

Results

In Table 5.3 we can compare side-by-side the average number of iterations in the original Semaev's Algorithm 5.1 and the modified Algorithm 5.1.1 with rounding. The average number of iterations is practically the same as with the Algorithm 5.1. It is most likely only a coincidence we obtained the shortest vector in all calculations, as this algorithm is not proven to solve the SVP.

The algorithm's progress remains the same - the first step is the most significant one, reducing the size of the shortest vector approximately 10 to 10^3 times, as can be seen in Figure 5.2.

range for d_i	Exact SV	avg iterations	range for d_i	avg iterations
10^5	100	4.55	10^5	4.62
10^{10}	100	6.75	10^{10}	6.9
10^{15}	100	9.33	10^{15}	9.2
10^{20}	100	11.22	10^{20}	11.16
10^{25}	100	13.39	10^{25}	13.72
10^{30}	100	16.03	10^{30}	15.66
10^{35}	100	19.29	10^{35}	18.9
10^{40}	100	21.92	10^{40}	21.2

Table 5.3: Left: Number of iterations of the Semaev's algorithm with rounding depending on the range of d_i .

Right: Number of iterations of the original Semaev's algorithm on matrices with the same parameters. In both cases, the results in each line were computed on 100 random matrices of dimension 3.

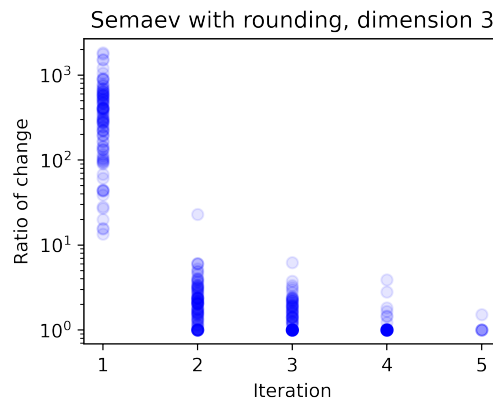


Figure 5.2: Semaev's algorithm with rounding, the ratio ϱ of the norms of the shortest vectors of two consecutive iterations. Computed on 100 matrices of dimension 3 with range for d_i 10^5 .

5.2 Generalizing for higher dimensions

There are multiple possible ways to generalize Semaev's algorithm for higher dimensions. Chapter 5.2.1 presents a naive variant, where the vectors b_1 and b_2 are reduced using the Lagrange-Gauss algorithm, but then it reduces b_n using all the vectors $b_{n-1}, \dots, b_2, b_1$. Chapter 5.2.2 then briefly introduces a modified version for dimension 4, which, at the cost of higher computation time, can reliably find the shortest vector with 100 % success rate in our experiments. Chapter 5.2.3 then returns to the idea of rounding and tests its results on dimensions greater than 3.

5.2.1 Naive approach

The original version of Semaev's algorithm is meant to be used on dimension 3, but this prerequisite can be omitted. Again, by doing that, we lose the certainty of finding the shortest vector. Line 5 of the Algorithm 5.1 computes the real-valued minimum by reducing the longest vector b_n with the vectors $b_{n-1}, \dots, b_1$ such, that the vectors b_2 and b_1 have been previously reduced using the Lagrange-Gauss algorithm.

The simplest and naive generalization of Semaev's algorithm for higher dimension is, to again reduce the longest vector by all the previous ones, i. e. to solve the problem

$$\min(|b_n + x_{n-1}b_{n-1} + \dots + x_2b_2 + x_1b_1|)$$

The question is: can this algorithm effectively find the shortest vector in any dimension?

Results

n	Tries	Exact SV	shorter than LLL	Not exact	avg iter	avg diff
4	100	100	2	0	13.61	56.9
6	100	98	2	3	27.23	3.49
10	100	96	7	4	51.2	0.58

Table 5.4: Semaev's algorithm naively applied to higher dimensions. The results of correspond to random matrices with range of d_i being 10^{10} .

The algorithm has been tested on 100 matrices in dimensions ranging from 4 to 15, and it indeed could find the shortest vector somewhat reliably, being successful in 99 % of cases in dimension 4, 96-97 % of cases in dimensions 6 and 10. The average absolute difference from the shortest vector s gradually decreased. This is probably due to the fact that we tested the algorithm on uniform range

Line	Operation	Multiplications	Additions	Other
3	Gauss reduction	$6n + 1$ per it	$5n - 4$ per it	1 per it
4	Gram matrix	n^3	$n^3 - n$	
5	Real minimum	$\frac{n^3}{6}$	$\frac{n^3}{6}$	$\frac{n(n-1)}{2}$ divisions and n square roots [4]
6	Shortest lc in cube	$2^{n-1} \cdot (n^2 + n)$	$2^{n-1} \cdot (n^2)$	
7	candidate	n^2	$n^2 - n$	
9	$ \text{candidate} $	n	$n - 1$	
1	sort by norm	n^2	$n^2 - n$	

Table 5.5: Number of individual arithmetic operations of the Semaev algorithm applied to higher dimensions. The *Real minimum* function assumes that the Cholesky method is used to solve the system of linear equations.

$d_i \in (-10^{10}, 10^{10})$ in the first column in the matrix, however, it is probably more difficult to reduce such high numbers with 4 vectors, than with 10.

However, the computation time of this algorithm was rather long. The computation time is enormous, as we are computing the norm of all 2^{n-1} possible vectors of the cube. A detailed breakdown of the number of operations for each iteration can be seen in Table 5.5. We shall try to bypass this problem in chapter 5.2.3.

As expected, this simple generalization does not generally return the shortest vector. This might be due to the fact that we are not reducing b_n by the shortest possible vectors $b_{n-1}, \dots, b_1$ as in the 3-dimensional case. We will try to prevent it in the following modification.

5.2.2 Modified algorithm for dimension 4

The modified algorithm aims to find the shortest basis vectors in dimension 4 using a modification of Semaev's algorithm 5.1. Previously, in Chapter 5.2.1, when running the algorithm in multiple dimensions, the stopping condition was reached prematurely.

Let's assume vectors b_1, b_2, b_3, b_4 that form a basis. Algorithm 5.1 minimized $\min(|b_4 + x_3b_3 + x_2b_2 + x_1b_1|)$. The algorithm terminated when the vector b_4 could no longer be reduced by the vectors b_3, b_2, b_1 . However, the issue is that it may still be possible to further reduce the vector b_3 , for example using the vectors b_4, b_2, b_1 , or even just using the pair of the shortest vectors b_2, b_1 .

Our modified version explores this case. If it occurs that the vector b_4 can no longer be reduced, the Algorithm 7 attempts to reduce the triplet of vectors b_3, b_2, b_1 using the original Semaev's algorithm for three dimensions. If it manages to reduce the three vectors, it tries to reduce b_4 again.

Algorithm 7 Semaev for higher dimensions, modified version

Require: B matrix of a basis, dimension 4, its vectors are sorted by norm

```

1: function SEMAEV FOR HIGHER DIMENSIONS( $B = (b_1, \dots, b_n)$ )
2:   loop
3:      $b_1, b_0 \leftarrow \text{GAUSS}(b_1, b_0)$ 
4:      $G \leftarrow \text{GRAM MATRIX}(B)$ 
5:      $\text{real\_lc} \leftarrow \text{REAL MINIMUM}(G, 3, 1)$ 
6:      $\text{cube\_shortest\_lc} \leftarrow \text{SHORTEST LC IN CUBE}(B, \text{real\_lc})$ 
7:      $\text{candidate} \leftarrow (\text{cube\_shortest\_lc}) \cdot B$ 
8:      $\text{longest\_norm} \leftarrow g_{4,4}$ 
9:     if  $\|\text{candidate}\| < \text{longest\_norm}$  then
10:       $b_4 \leftarrow \text{candidate}$ 
11:       $B \leftarrow \text{SORT BY NORM}(B)$ 
12:     continue
13:    $b'_1, b'_2, b'_3 \leftarrow \text{SEMAEV3}(b_1, b_2, b_3)$ 
14:   if  $b'_1, b'_2, b'_3 = b_1, b_2, b_3$  then
15:     break
16:    $b_1, b_2, b_3 \leftarrow b'_1, b'_2, b'_3$ 
17:    $B \leftarrow \text{SORT BY NORM}(B)$ 
18: return  $B$ 
```

▷ If b_n cannot be further shortened

▷ If 3-dimensional Semaev did not change anything

Another possible modification is to replace the Lagrange-Gauss step on line 3 directly with the Semaev's algorithm, but we haven't tested this method.

Results

n	Tries	Exact SV	shorter LLL	than	Not exact	avg iter
4	10 000	10 000	151		0	15.5
5	1000	995	19		5	23.7

Table 5.6: Accuracy of the Algorithm 7 on matrices of dimensions 4 and 5. Range of d_i is 10^{10} .

We have run the Algorithm 7 on 10 000 matrices of dimension 4 and the results are shown in Table 5.6. The success rate for finding the shortest vector is 100 % this time, while the average number of iterations grew from 13.9 in Table 7 to 15.5.

We have tried running the algorithm also on matrices of dimension 5, reaching a 99.5 % success rate. This result is not surprising, as we would have to modify the algorithm even further to accommodate higher dimensions and solve all edge cases. Hypothetically, we could recursively modify the Algorithm 7 for every dimension $n > 4$, but it would be computationally complex.

5.2.3 The rounding modification for higher dimensions

The rounding algorithm applied on dimension 3 seemed promising in terms of computation time and number of operations. We decided to run the Algorithm 5.1.1 on higher dimensions and measure the accuracy of finding the shortest vector, as its lower computation time might be useful for our final algorithm.

In Table 5.7 we can see the results in multiple dimensions. This time, however, the accuracy of finding the shortest vector is drastically decreasing with increasing dimension, with 65 % in dimension 6 and 0 % from dimension 10 onward. Not only the algorithm does not find the exact vector, but the relative difference between the result and the exact algorithm is much higher, the average ranging from 1.7.8 in dimension 4 and to 4386.9 in dimension 15.

n	Tries	Exact SV	shorter than LLL	Not exact	avg iter	$\frac{\ b_1\ }{\ s\ }$
3	100	100	0	0	6.66	-
4	100	98	3	2	15.34	1.7
6	100	62	1	38	39.34	268.2
10	100	0	0	100	16.85	2410.8
15	100	0	0	100	12.78	4386.9

Table 5.7: The rounding modification, Algorithm 5.1.1 applied to higher dimensions, range of d_i is 10^{10} . The last column shows the ratio of lengths of the result of Semaev's algorithm (b_1) and the exact shortest vector s in cases when Semaev's algorithm did not find the SV.

The accuracy of finding the shortest vector is not optimal, however, if we examine the ratio of change in the first step to the length of the shortest vector, as illustrated in Figure 5.3, it is evident that the first step still reduces the norm significantly more than the later steps. We shall use this result in the next chapter.

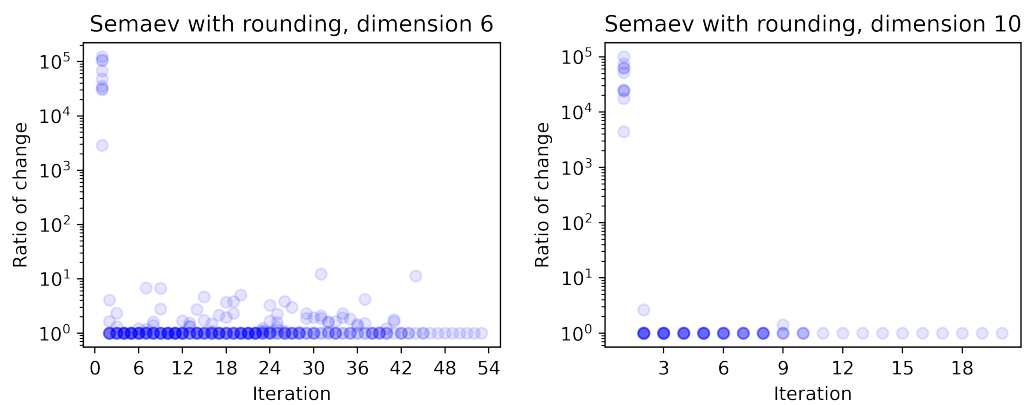


Figure 5.3: The size ratio ϱ of the modified Semaev's algorithm with rounding. Dimensions 6 and 10, range of d_i is 10^{10} .

Chapter 6

The proposed algorithm

The last chapter combines the knowledge gained in Chapter 5 and attempts to use it to improve the basic version of the LLL algorithm. In Chapter 6.1, we first perform an analysis of LLL similar to the one we did for Semaev's algorithm in Chapter 5. We examine the change in the norm in the basis during its run and the number of iterations of its main loop. We observed that the most significant reduction of the shortest vector and basis in the LLL algorithm occurs in the middle steps.

We attempt to combine the best of both approaches - the accuracy of the LLL algorithm, which is capable of finding sufficiently short vectors, and Semaev's algorithm, which can rapidly reduce the basis in the initial iteration. We expect the new algorithm to be quicker than the LLL algorithm and more accurate than Semaev's algorithm with rounding (Algorithm 5.1.1).

In Chapter 6.2, we then describe our proposed algorithm. We first apply our modified Semaev's algorithm to the basis $(b_1, \dots, b_n)$, which significantly shortens the vector b_n , and then we apply the LLL algorithm to this adjusted basis $(b'_1, \dots, b'_n)$. Chapter 6.3 then compares the results of LLL and our proposed algorithm.

6.1 Analysis of the LLL algorithm

To combine our algorithms from Chapter 5 with the LLL, it is worthwhile to conduct the same analysis on the LLL algorithm. We anticipate that the LLL algorithm may not be as efficient in some parts of the process. Given the nature of the algorithm, we would expect the size reduction to be slower and more gradual than in Semaev's case.

We have run the Algorithm 1 on a similar set of random matrices as the Semaev's. The iterations here are counted as the iterations of the outer while cycle on line 3 of Algorithm 1. Figure 6.1 confirms our predictions. In comparison to Semaev's algorithm, the most significant reduction occurs around iterations 4-6 on 3-dimensional matrices. Using the knowledge we gained in the previous chapter, we can now try to combine these two approaches.

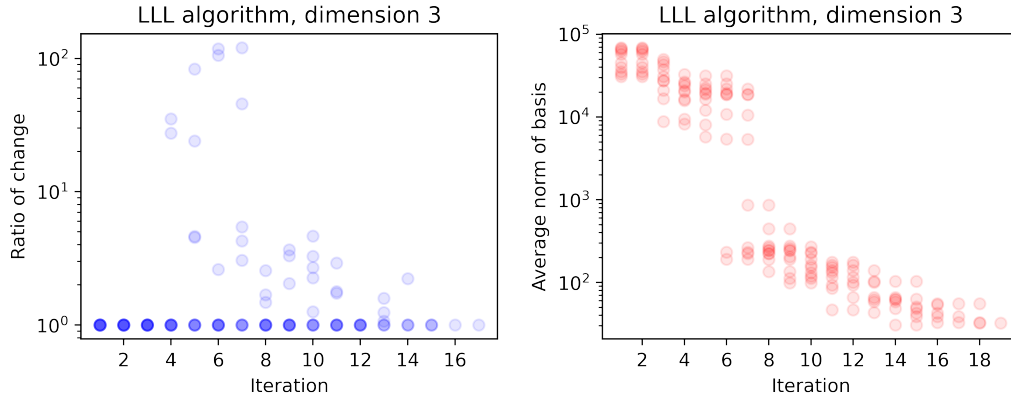


Figure 6.1: The ratios ϱ and the average length of the basis at the end of each iteration of the LLL algorithm. Random matrices of dimension 3, range 10^5 .

n	Tries	Exact SV	Not exact	avg iter of LLL	$\frac{\ b_1\ }{\ s\ }$
3	100	100	0	25.15	-
4	100	100	0	47.62	-
6	100	99	1	109.9	1.11
10	100	95	5	279.55	1.06
15	100	80	20	514.22	1.06
20	100	57	43	718.64	1.06

Table 6.1: Iterations of the LLL's while cycle in varying dimensions, range 10^{10} .

6.2 Semaev-enhanced LLL

When comparing the rates of shortening in Figures 5.3 and 6.1, we observed that the most significant reduction of the shortest vector and basis in the LLL algorithm occurs in the middle steps. Since the reduction during the first iteration of Semaev's algorithm with rounding (Algorithm 5.1.1) is very significant compared to the first few iterations of the LLL, our proposal is to essentially first reduce the basis with a single iteration of Algorithm 5.1.1 instead of determining the exact shortest vector of the cube, and then applying the LLL algorithm. This method could potentially accelerate the LLL algorithm and reduce the number of iterations. The proposed algorithm is as follows:

Algorithm 8 The proposed algorithm

Require: $b_1, \dots, b_n$ a lattice basis, dimension n , its vectors are sorted by norm

```

1: function THE PROPOSED ALGORITHM( $b_1, \dots, b_n$ )
2:    $b_2, b_1 \leftarrow \text{GAUSS}(b_2, b_1)$ 
3:    $G \leftarrow \text{GRAM MATRIX}(b_1, \dots, b_n)$ 
4:    $c_R \leftarrow \text{REAL MINIMUM}(G, n - 1, 1)$ 
5:    $\text{closest\_cube\_lc} \leftarrow \text{ROUND EVERY COMPONENT}(c_R)$ 
6:    $b_n \leftarrow (\text{closest\_cube\_lc}) \cdot B$ 
7:    $b_1, \dots, b_n \leftarrow \text{LLL}(b_1, \dots, b_n)$ 
8:   return  $b_1, \dots, b_n$ 

```

6.3 Results

We have run the Algorithm 8 on matrices with the same parameters as with the LLL algorithm in Chapter 6.1. The proposed algorithm performed slightly better than LLL in terms of accuracy, as it found the exact shortest vector 57 times in dimension 20, while LLL ran on the same matrices found the shortest vector only 29 times, as can be seen in Table 6.2.

The table 6.3 compares the average number of iterations of the LLL algorithm and the proposed algorithm. The number of iterations has decreased significantly, reducing the number of iterations of the LLL algorithm by 18 % on dimension 20.

n	Tries	Exact SV	shorter than LLL-only	Not exact	avg iter of LLL	$\frac{\ b_1\ }{\ s\ }$
3	100	100	1	0	14.17	0
4	100	99	1	1	34.71	1.03
6	100	100	1	0	88.41	0
10	100	97	8	3	235.83	1.09
15	100	80	11	20	435.11	1.08
20	100	57	28	43	585.16	1.09

Table 6.2: Accuracy and average iterations of the proposed algorithm, run on matrices of various dimensions, range of $d_i 10^{10}$.

	number of iterations of LLL		
dimension	LLL-only	the proposed algorithm	% change
3	25.15	14.17	- 45.9 %
4	47.62	34.71	- 24.8%
6	109.9	88.41	- 17.56 %
10	279.55	235.83	- 15.7 %
15	514.22	435.11	- 15.3 %
20	718.64	585.16	- 18.5 %

Table 6.3: Comparison of the average number of iterations of the standalone LLL algorithm and the proposed algorithm for various dimensions. Range of $d_i 10^{10}$.

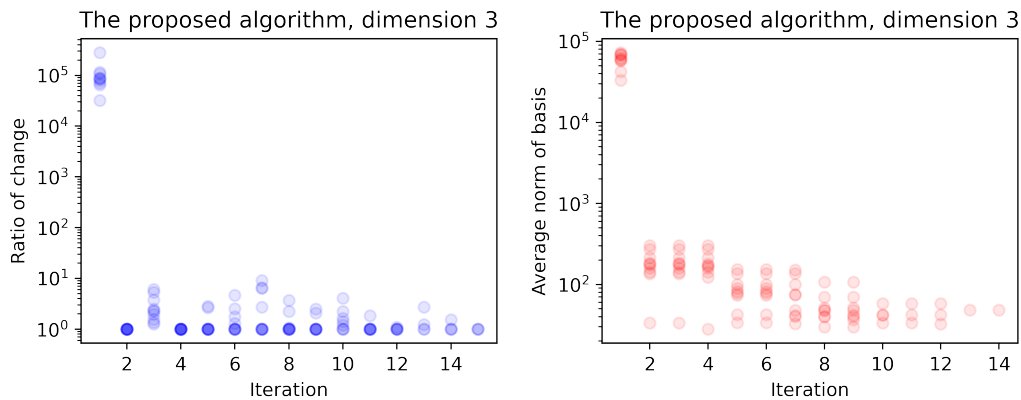


Figure 6.2: The size ratios ρ and the average length of the basis of the proposed algorithm. The first iteration is the single iteration of the Semaev's algorithm with rounding, the other iterations are of the LLL algorithm. Matrices of dimension 3.

Conclusion

We have generated random bases of various dimensions and sorted them into two sets according to whether the cube algorithm was able to find the shortest vector. We then analyzed these bases, but didn't manage to find a simple property that could determine whether the cube algorithm would return the shortest vector for a given base. However, we found that if there are two vectors with angle $\alpha < 5^\circ$ in the basis, then the cube algorithm is about ten times more likely to return the wrong vector. We discovered additional minor results related to the minimum's position within the cube, which were integrated into a simple algorithm; however, this approach failed to improve the success rate and efficiency of the shortest vector (SV) search.

After determining that the earlier approaches were ineffective, we shifted towards exploring other methods and focused on Semaev's exact algorithm for dimension 3. We analyzed the number of its iterations as a function of the range of input parameters and investigated the rate of size reduction of the bases in each iteration. We found that the greatest reduction occurs in its first iteration. Next, we attempted three different ways of generalizing Semaev's algorithm to higher dimensions. However, two of them were found to be computationally difficult due to an exponentially-difficult step. In the third method, we avoided the exponential step by rounding. This modification significantly accelerated the computation, but also lowered the accuracy, which decreased rapidly with increasing dimension.

In the last chapter, we aim to connect the rounding modification and the LLL algorithm. We have shown that in the LLL algorithm, significant base reduction only occurs in later iterations. Thus, we propose a new algorithm that first applies a single iteration of the modified Semaev's algorithm to the input basis and then applies the LLL. We showed that our proposed algorithm can reduce the number of iterations of the LLL algorithm itself by 15-18 % in dimensions 6-20.

In conclusion, we have shown that predicting whether the cube algorithm will correctly solve the shortest vector problem (SVP) is a non-trivial task. However, through the use of quadratic forms, and modifying the input basis, we have successfully reduced the number of iterations of the LLL algorithm by nearly one fifth.

Bibliography

- [1] Miklós Ajtai. “The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract)”. In: *Symposium on the Theory of Computing*. 1998. URL: <https://api.semanticscholar.org/CorpusID:4503998>.
- [2] Sheldon Axler. *Linear Algebra Done Right*. 4th ed. Undergraduate Texts in Mathematics. Springer Cham, 2023. ISBN: 978-3-031-41025-3. DOI: [10.1007/978-3-031-41026-0](https://doi.org/10.1007/978-3-031-41026-0). URL: <https://doi.org/10.1007/978-3-031-41026-0>.
- [3] Chuck Easttom. “Lattice-Based Cryptography”. In: *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. Cham: Springer International Publishing, 2022, pp. 409–425. ISBN: 978-3-031-12304-7. DOI: [10.1007/978-3-031-12304-7_20](https://doi.org/10.1007/978-3-031-12304-7_20). URL: https://doi.org/10.1007/978-3-031-12304-7_20.
- [4] I. Horová and J. Zelinka. *Numerické metody*. Czech. 2nd ed. Brno: Masaryk university, 2004.
- [5] A.K. Lenstra, H.W. Lenstra, and L. Lovász. “Factoring polynomials with rational coefficients”. In: *Math. Ann.* 261 (1982), pp. 515–534. DOI: [10.1007/BF01457454](https://doi.org/10.1007/BF01457454).
- [6] Jan Mačák. *LLL algoritmus a HNP problém [online]*. Bakalářská práce. SUPERVISOR : Marek Sýs. 2021. URL: <https://is.muni.cz/th/y9y14/>.
- [7] Phong Q. Nguyen and Brigitte Vallée, eds. *The LLL Algorithm. Survey and Applications*. 1st ed. Information Security and Cryptography. eBook ISBN: 978-3-642-02295-1. Softcover ISBN: 978-3-642-26164-0. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-02294-4. DOI: [10.1007/978-3-642-02295-1](https://doi.org/10.1007/978-3-642-02295-1). URL: <https://doi.org/10.1007/978-3-642-02295-1>.
- [8] Mojtaba Bisheh Niasar. (1) *Lattices and Kyber PQC Presentation - YouTube*. Accessed: 2024-02-05. Sept. 2022. URL: https://www.youtube.com/watch?v=zsej28sfycs&t=608s&ab_channel=mojtababishehniasar.
- [9] Chris Peikert and Eric Crockett. *Lattices in Cryptography*. Lecture notes from the University of Michigan. 2015. URL: <http://www.eecs.umich.edu/~cpeikert/lic15/>.
- [10] Igor A. Semaev. “A 3-Dimensional Lattice Reduction Algorithm”. In: *Cryptography and Lattices*. 2001. URL: <https://api.semanticscholar.org/CorpusID:13401663>.

- [11] František Šik. *Lineární algebra zaměřená na numerickou analýzu*. 1st ed. Brno: Masarykova univerzita, 1998.
- [12] Lucie Šikudová. *LLL algoritmus a Gramova matice [online]*. Bakalářská práce. SUPERVISOR : Marek Sýs. 2022. URL: <https://is.muni.cz/th/z5tld/>.
- [13] Mårten Trolin. "The Shortest Vector Problem in Lattices with Many Cycles". In: *Cryptography and Lattices*. Ed. by Joseph H. Silverman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 194–205. ISBN: 978-3-540-44670-5.
- [14] Dominik Velan. *Aplikace algoritmu LLL [online]*. Bakalářská práce. SUPERVISOR : Michal Bulant. 2014. URL: <https://is.muni.cz/th/m89fv/>.
- [15] Philip Wang. *Gram-Schmidt-LLL*. <https://github.com/pwang00/Gram-Schmidt-LLL>. 2021.
- [16] Eric W. Weisstein. *Quadratic Surface*. MathWorld—A Wolfram Web Resource. 2023. URL: <https://mathworld.wolfram.com/QuadraticSurface.html>.
- [17] D. Zwillinger, ed. *CRC Standard Mathematical Tables and Formulas*. 33rd ed. Chapman and Hall/CRC, 2018. DOI: [10.1201/9781315154978](https://doi.org/10.1201/9781315154978).

Attachments

We attach the following appendices to this thesis. The results are attached in Markdown in each of the corresponding .ipynb files.

- `experiments.zip` containing all files used in Chapter 4, namely:
 - `generate.py`, `generate_matrices_for_cube.ipynb`: files used for generating matrices in Chapter 4.1,
 - `cube_matrices.zip`: file with matrices generated in Chapter 4.1 and used for experiments,
 - `angles.ipynb` used in Chapter 4.2 to analyze extreme angles,
 - `close_to_wall.py`, `close_to_wall.ipynb` used for experiments in Chapter 4.3,
 - `LLL_implementation.py`, `gram_schmidt.py` used to analyze the LLL algorithm in Chapter 6.1, edited version of the code from [15],
 - `closest_vs_minimal.py`, `closest_vs_minimal.ipynb` used in Chapter 4.4.
 - `descent.py`, `descent.ipynb` used in Chapter 4.5,
- `semaev_and_modifications.ipynb` used in Chapter 5 and Chapter 6,
- `semaev_tests.zip` matrices used for experiments in Chapter 5 and Chapter 6,
- `ratios_avgnorms.ipynb` data and code used for generating figures in Chapter 5 and Chapter 6.

