Annelie Heuser

Department of Mathematics Technische Universität Darmstadt

Chess Reduction

Using artificial intelligence for lattice reduction

Diploma Thesis February 2010

Supervised by

Prof. Dr. Johannes Buchmann

Michael Schneider

Department of Computer Science

Cryptography and Computeralgebra

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Acknowledgment

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project. Therefore it is a pleasure to thank:

Prof. Dr. Johannes Buchmann and Michael Schneider
Prof. Dr. Johannes Fürnkranz

Agnes Stern, Jennifer Schenk, Bettina Stern, Sebastian Ehrhart, Katrin Stern

And last but not least my parents.

Contents

List of Figures

1 Introduction

1.1 Outline

Lattices are discrete subgroups of $\mathbb{R}^n$, which can be represented by linear independent vectors $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\} \in \mathbb{R}^n$ $(d \leq n)$ such as $\mathcal{L} = \{\sum_{i=1}^{d} x_i \mathbf{b_i} \mid x_i \in \mathbb{Z}\}$. $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\} \in \mathbb{R}^n$ is called basis and d the dimension of the lattice $\mathcal{L}$. In dimension ≥ 2 a lattice has infinitely many bases, but some are more useful than others.

Lattice reduction is the search for short and orthogonal vectors in a lattice. It helps to determine the actual hardness of cryptosystems which are based on the hardness of special lattice-based problems. Lattice-based cryptosystems exhibit strong security even in the presence of quantum computers.

The shortest vector problem (SVP) is the most famous lattice problem. It searches for the shortest non-zero vector in the lattice (usually in the Euclidean Norm l_2).

Algorithms to solve the SVP can be divided into two different categories: exact algorithms and algorithms which solve the γ-SVP which is an approximate version of the SVP. γ-SVP is the problem of finding a vector which is at most γ times the length of the shortest non-zero lattice vector.

Until today there exist two different types of algorithms to solve the SVP exactly: deterministic enumeration algorithms and probabilistic sieve algorithms. In [4] Ajtai, Kumar and Sivakumar introduced the first probabilistic algorithm called AKS Sieve. The latest and best version of AKS Sieve has been presented by Micciancio and Voulgaris in [17].

Deterministic algorithms were discovered by Kannan [12] and Fincke and Pohst [7]. In this thesis we will mostly focus on an improvement of these algorithms by Schnorr and Euchner called ENUM [25].

In higher dimensions only approximation algorithms are practical. LLL was the first algorithm to solve the γ-SVP. It was invented by Lenstra, Lenstra, and Lovász. It computes a $2^{n/2}$-SVP in polynomial time. BKZ introduced by Schnorr and Euchner is mostly used in practise today. It uses ENUM as a subroutine combined with LLL.

Since BKZ relies on a SVP solver, it is therefore very important to know what is the best exact SVP solver in low dimensions. ENUM can be interpreted as a depth first search through a search tree. In this thesis we try to use improvements done in the field of artificial intelligence to improve the search strategy of ENUM.

Search algorithms are a classic and well-developed part of artificial intelligence. They are the basis of many problem solving algorithms as many applications need to search for the best solution.

Game playing is one of the classic problems in artificial intelligence. Non-trivial games like nine-men's morris, connect-four and qubic have been solved. But in games like shogi, bridge and go computers can be easily outplayed by human experts due to the exponential growth in computational effort with increasing search depths. Since the 50's chess remains the most studied game in artificial intelligence.

Therefore we introduce and explain fundamental search algorithms used in the field of artificial intelligence and try to adapt each of them to ENUM. One aim of this paper is to determine if those adaptations are efficient.

We gain a deeper insight into search algorithms used for game playing by concentrating on one specific game. Consequently we picked the game chess. Our purpose is to give a survey of all used search enhancements used to play chess, following with an analyses about the comparability to ENUM.

Finally, we combine all acquired knowledge as well as a proposal given by Schnorr in [24] to improve and implement a new ENUM version called New ENUM.

1.2 Road-Map

The thesis is organized as follows. In Section 2.1, we provide neccessary background onto lattices and lattice reduction. In Section 2.2, we recall SVP solver

such as the enumeration of FINCKE and POHST with the improvement by SCHNORR (ENUM) as well as the probahilistic Sieve AKS algorithm.

Section 3.1 gives an overview of search strategies used in the field of artificial intelligence. Whereas in Section 3.2 we concentrate on one specific game, chess. We outline used algorithms as well as heuristics.

In Chapter 4 we try to adapt algorithms and heuristic presented in Chapter 3 to ENUM.

Chapter 5 introduces an algorithm called New ENUM which follows the conclusions made in 4 as well as a proposal given by SCHNORR in [24].

2 Lattice Reduction

We start with a short survey about the basics of lattices and its known $\mathcal{NP}$-hard problems. Further on we concentrate on the exact shortest vector problem (SVP) and algorithms to solve it. These algorithms can be classified into two different groups: deterministic enumeration and randomized Sieve algorithms. We then continue by giving a short overview of terms of reduction which can be achieved by special algorithms solving the γ-SVP (a relaxed version of the SVP).

2.1 Preliminaries

2.1.1 General

For any $x \in \mathbb{R}$, $\lceil x \rfloor = \lceil x - 0.5 \rceil \in \mathbb{Z}$ denotes the closest integer to x. We use bold lower case letters (e.g. $\mathbf{b}$) to denote vectors, whereas matrices are denoted by bold upper case letters (e.g. $\mathbf{B}$). $\mathbf{B} = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\} \in \mathbb{R}^{n \times d}$ stands for a column matrix with vectors $\mathbf{b_i} \in \mathbb{R}^n$. The ith coordinate of a vector is described as b_i.

Let $\mathbf{b} \in \mathbb{R}^n$ we denote $||\mathbf{b}|| = (\sum_{i=1}^{n} \mathbf{b_i}^2)^{1/2}$ as the Euclidean norm, unless otherwise stated and $\langle \cdot, \cdot \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ the standard scalar product, $\langle \mathbf{b_i}, \mathbf{b_j} \rangle = (\mathbf{b_i}^\mathsf{T} \mathbf{b_j})$. Let us denote by $\mathcal{B}_{t-1}(x, r)$ a $t-1$-dimensional sphere around x with radius r.

2.1.2 Lattices

A lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$, but for our purpose the following definition is more illustrating.

Definition 2.1 (Lattice) *Let* $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ *be a set of linear independent non-zero vectors in* $\mathbb{R}^n$*. A* ***lattice*** *of dimension* d *in* $\mathbb{R}^n$ *is defined as*

$$\mathcal{L}(\mathbf{b_1}, \ldots, \mathbf{b_d}) := \left\{ \sum_{i=1}^{d} x_i \mathbf{b_i} \mid x_i \in \mathbb{Z} \right\},$$

where $d \leq n$*.*

The simplest lattice is $\mathbb{Z}^n$. $\mathbf{B} = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ is called a **basis** of the lattice $\mathcal{L}$, which is not unique. If $d \geq 2$, a lattice $\mathcal{L}$ has infinitely many bases. We can transform one basis to another representing the same $\mathcal{L}$ by unimodular integer transformations $\mathbf{T}$, which have $\det(\mathbf{T}) = \pm 1$. Figure 2.1 shows two different bases $\mathbf{B} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and $\mathbf{B}' = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$, representing the same lattice $\mathcal{L}$ in $\mathbb{R}^2$.

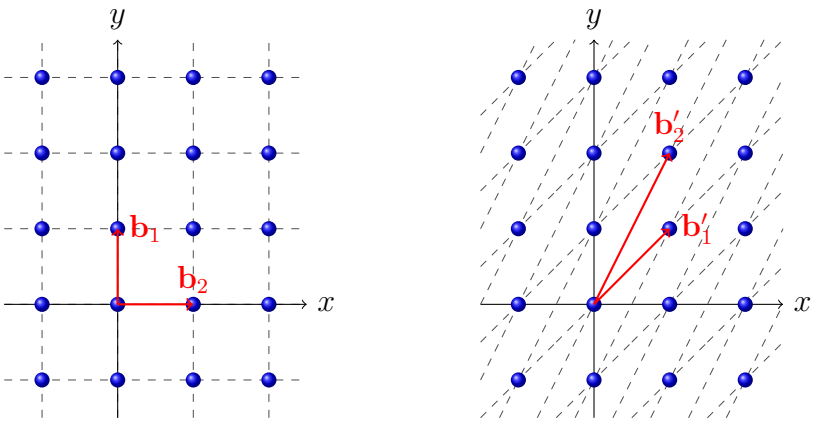


Figure 2.1: A lattice spanned by two different bases

One of the main goals of lattice reduction is finding a short basis to a given one, which represents the same lattice $\mathcal{L}$. A short basis consists of short vectors relative to a given norm, which is the same as nearly pairwise orthogonal lattice basis vectors. This is generally called reduced.

Definition 2.2 (Lattice Determinant) *The* ***determinant*** *of a lattice* $\mathcal{L} \subseteq \mathbb{R}^n$ *with basis* $\mathbf{B}$ *is defined as*

$$\det(\mathcal{L}) = \sqrt{\det(\mathbf{B}^t \mathbf{B})}$$

The determinant of a lattice $\det(\mathcal{L})$ is invariant of the choice of the lattice basis $\mathbf{B}$, because basis transformations have $\det = \pm 1$ and $\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B})$.

To explain geometrically, $\det(\mathcal{L})$ is the volume of the *fundamental region*, $|\det(\mathcal{L})| = vol(\mathcal{P}(\mathbf{B}))$. According to 2.1, Figure 2.2 shows the fundamental region of both bases. Since the determinant is a lattice invariant $vol(\mathcal{P}(\mathbf{B})) = vol(\mathcal{P}(\mathbf{B}'))$.

Definition 2.3 (Fundamental Region)

$$\mathcal{P}(\mathbf{B}) = \left\{ \sum_{i=1}^{d} x_i \mathbf{b_i} \mid x_i \in \mathbb{R},\ 0 \leq x_i < 1 \right\} \subset span(\mathcal{L}),$$

where $span(\mathcal{L}) := \{\mathbf{Bx} \mid \mathbf{x} \in \mathbb{R}^n\}$.

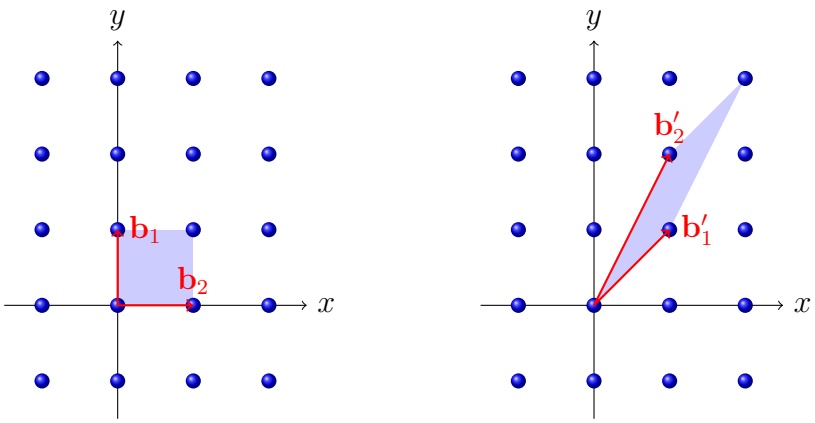


Figure 2.2: This Figure shows that the volume of both fundamental regions is the same

Definition 2.4 (Successive Minima) *Let* $\mathcal{L} \subseteq \mathbb{R}^n$ *be a lattice with* $dim(\mathcal{L}) = d$. *The* ***successive minima*** $\lambda_1, \ldots, \lambda_d$ *are defined as*

$$\lambda_i = \lambda_i(\mathcal{L}) := \min \left\{ r > 0 \mid dim(span(\mathcal{L} \cap \mathcal{B}_r(0))) \geq i \right\},$$

where $\mathcal{B}_r(0)$ *defines a sphere around the origin with radius* r. $\lambda_1(\mathcal{L})$ *is the length of the shortest non-zero lattice vector of* $\mathcal{L}$.

The first and second successive minimum of a lattice in $\mathbb{R}^2$ is shown in Figure 2.3. The successive minima give an evidence to the reduction of the lattice basis, if $\|b_i\| / \lambda_i$ is small the lattice basis is reduced. Note that $\lambda_1 \leq \cdots \leq \lambda_d$. Unfortunately, the existence of a lattice basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ with $\|\mathbf{b_i}\| = \lambda_i$ (for $i = 1, \ldots, d$) or orthogonal vector is quite rare.

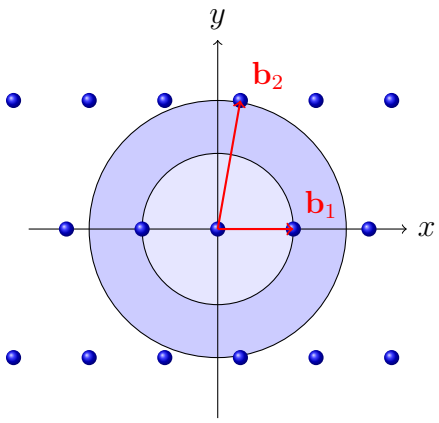


Figure 2.3: First and second successive minimum, represented by the circles

Definition 2.5 (Hermite constant) *Let $d \in \mathbb{N}$ be the dimension of a lattice $\mathcal{L}$. The **Hermite constant** is defined as*

$$\gamma_d = \sup_{\mathcal{L}} \left\{ \frac{\lambda_1^2(\mathcal{L})}{(\det(\mathcal{L}))^{\frac{2}{d}}} \right\}.$$

The Hermite constant γ_d of dimension d satisfies Minkowski's second theorem:

$$\prod_{i=1}^{d} \lambda_i(\mathcal{L}) \leq \sqrt{\gamma_d^d}\ \det(\mathcal{L})$$

The exact value for γ_d is only known for dimension $1 \leq d \leq 8$ and $d = 24$.

2.1.3 Orthogonalisation

As already mentioned, finding short nearly pairwise orthogonal lattice basis vectors is one of the main goals of lattice reduction. Finding an orthogonalization in a vector space can be done easily using the **QR**-decomposition, which decomposes a basis **B** into the product **QR**, where **Q** is an orthogonal and **R** an upper triangular matrix. Unfortunately finding an orthogonalization for a lattice basis **B** is not that easy since **Q** must not be integral. Hence, **Q** must not be a lattice basis, even if **R** can be computed as an unimodular transformation from **B** into **Q**.

However, we will see that the orthogonalization is used in some reduction algorithms, therefore we shortly present an orthogonalization method called Gram-Schmidt-algorithm:

Let $B = \{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ be a lattice basis. The orthogonalization $\widehat{\mathbf{B}} = \{\widehat{\mathbf{b}}_\mathbf{1}, \ldots, \widehat{\mathbf{b}}_\mathbf{d}\}$ can be computed recursively with the Gram-Schmidt-coefficients

$$\mu_{ij} = \frac{\langle \mathbf{b_i}, \widehat{\mathbf{b}}_\mathbf{j} \rangle}{\langle \widehat{\mathbf{b}}_\mathbf{j}, \widehat{\mathbf{b}}_\mathbf{j} \rangle}, \quad \widehat{\mathbf{b}}_\mathbf{1} = \mathbf{b_1}, \quad \widehat{\mathbf{b}}_\mathbf{i} = \mathbf{b_i} - \sum_{j=1}^{i-1} \mu_{ij} \widehat{\mathbf{b}}_\mathbf{j}.$$

The Gram-Schmidt-orthogonalization of a basis $\mathbf{B}$ in $\mathbb{R}^2$ is shown in Figure 2.4, illustrating that $\widehat{\mathbf{B}}$ must not span the same lattice.

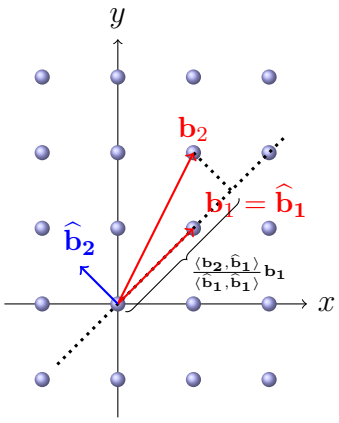


Figure 2.4: Gram-Schmidt orthogonalization of a basis in $\mathbb{R}^2$. $\widehat{\mathbf{b}}_\mathbf{2}$ is not a lattice vector

Let $\{\widehat{\mathbf{b}}_\mathbf{1}, \ldots, \widehat{\mathbf{b}}_\mathbf{d}\}$ be given to a basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$, then $\det(\mathcal{L}(\{\mathbf{b_1}, \ldots, \mathbf{b_d}\})) = \prod_{i=1}^{d} ||\widehat{\mathbf{b}}_\mathbf{i}||$.

Definition 2.6 (Orthogonal Projection) *Given a basis* $\mathbf{B}$ *of* $\mathcal{L}$ *the* ***orthogonal projection*** π_i *is defined as*

$$\pi_i : \mathbb{R}^n \longrightarrow span(\mathbf{b_1}, \ldots, \mathbf{b_{i-1}})^\perp, \ i = 1, \ldots, d.$$

Then $\mathcal{L}_i = \pi_i(\mathcal{L}) := (\mathcal{L}(\pi_i(\mathbf{b_i}), \ldots, \pi_i(\mathbf{b_d})))$ *is said to be the* ***projected lattice*** *of dimension* $d - i + 1$.

2.1.4 Existing Algorithmic ($\mathcal{NP}$-hard) Problems

- Shortest Vector Problem (SVP)
  Given a lattice $\mathcal{L}$ the task of SVP is finding a non-zero vector $\mathbf{v} \in \mathcal{L}$ such

that

$$\|\mathbf{v}\| = \lambda_1(\mathcal{L})$$

SVP is known to be $\mathcal{NP}$-hard under randomized reductions. [2]

- Approximate Shortest Vector Problem (γ-SVP)
  The approximate version is a relaxed version of the SVP. It looks for non-zero $\mathbf{v}$ with

$$\|\mathbf{v}\| = \gamma \cdot \lambda_1(\mathcal{L})$$

- Closest Vector Problem (CVP)
  Given a lattice $\mathcal{L}$ and a target vector $\mathbf{t} \in span(\mathcal{L})$. Finding a non-zero vector $v \in \mathcal{L}$ such that

$$\|\mathbf{v} - \mathbf{t}\| = \min_{u \in \mathcal{L}} \|\mathbf{u} - \mathbf{t}\|$$

  is called CVP. The problem is known to be $\mathcal{NP}$-hard to solve exactly.

- Shortest Basis Problem (SBP)
  Given a lattice $\mathcal{L}$ with $dim(\mathcal{L}) = d$. Finding a basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ with

$$\|\mathbf{b}_1\| = \lambda_1, \ldots, \|\mathbf{b}_d\| = \lambda_d$$

  is called SBP.

Beyond that exist approximate versions, γ-CVP, γ-SBP among others. All problems can be formulated in all norms, the Euclidean norm is the most common one. For more details see [3, 9, 15, 16]. Further on we will concentrate on the SVP and γ-SVP, which is the most famous and widely studied one in lattice reduction so far.

2.1.5 Terms of Reduction and Algorithms

There are two types of algorithms considered here: those who solve the SVP exactly and those who solve the γ-SVP. The best algorithm known for γ-SVP in high dimension (BKZ) depends on an exact SVP solver in low dimension (ENUM).

If the lattice dimension is sufficiently low, the SVP can be solved exactly using a complete enumeration. In [12] Kannan introduced an SVP-algorithm which suffers from its expensive preprocessing. Another SVP solver was invented by

FINCKE and POHST in [7]. It calculates the shortest vector by minimizing a quadratic form over an ellipsoid. SCHNORR and EUCHER invented in [25] some improvements, e.g. the zig-zag-path. Their method is the most efficient one known in practice, at least up to dimension ~ 40. A quite different approach is taken by AJTAI, KUMAR and SIVAKUMAR, who invented the AKS Sieve algorithm [4]. Each of them will be explained in Section 2.2.
In higher dimension (≥ 100) only approximation algorithms are achievable.

SIZE-REDUCED. A basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ is called size-reduced if $|\mu_{ij}| \leq 1/2$ for all $1 \leq j < i \leq d$. This notation was introduced by HERMITE. Size-reduced is too weak to be used exclusively, but it can be combined as the following terms show.

GAUSS-REDUCED. This term only refers to 2-dimensional lattices. A lattice basis $B = (\mathbf{b_1}, \mathbf{b_2})$ is called Gauss-reduced if

$$\|\mathbf{b_1}\| \leq \|\mathbf{b_2}\| \leq \|\mathbf{b_1} \pm \mathbf{b_2}\| .$$

For $\mu_{21} = \frac{\langle \mathbf{b_1}, \mathbf{b_2} \rangle}{\|\mathbf{b_1}\|^2}$, it follows that

$$\mu_{21} \leq \frac{1}{2} \Longleftrightarrow \|\mathbf{b_1}\| \leq \|\mathbf{b_1} - \mathbf{b_2}\| \quad \text{and}$$
$$\mu_{21} \geq 0 \Longleftrightarrow \|\mathbf{b_1} - \mathbf{b_2}\| \leq \|\mathbf{b_1} + \mathbf{b_2}\| .$$

According to that, the basis is reduced if and only if $\|\mathbf{b_1}\| \leq \|\mathbf{b_2}\|$ and $0 \leq \mu_{21} \leq \frac{1}{2}$. Let ϕ denote the angle between $\mathbf{b_1}, \mathbf{b_2}$ then $0 \leq \cos\phi \leq \frac{1}{2}$ ($60° \leq \phi \leq 90°$), as the following picture shows. Note that if a basis is Gauss-reduced then $\mathbf{b}_1$ is the shortest vector in $\mathcal{L}$. Algorithms which output Gauss-reduced bases are SVP solver in dimension 2.

The next term of reduction is a generalization of Gauss-reduced in higher dimension.

LLL-REDUCED. The LLL-algorithm of LENSTRA, LENSTRA, and LOVÁSZ was the first polynomial time algorithm to solve the γ-SVP in $dim \geq 2$. A basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ is LLL-reduced with factor δ for $1/4 \leq \delta \leq 1$, if it is size-reduced and

$$\delta \left\|\widehat{\mathbf{b}}_{\mathbf{i-1}}\right\|^2 \leq \left\|\widehat{\mathbf{b}}_{\mathbf{i}}\right\|^2 + \mu_{i,i-1}^2 \left\|\widehat{\mathbf{b}}_{\mathbf{i-1}}\right\|^2 , \; i = 2, \ldots, d.$$

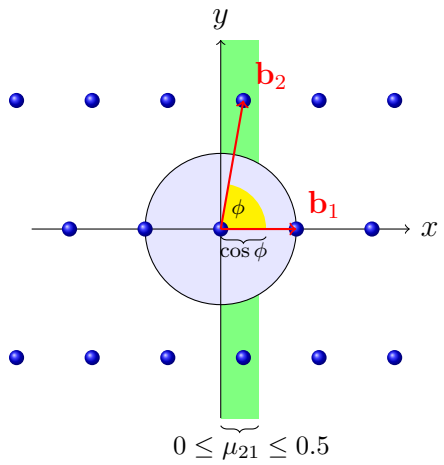


Figure 2.5: A Gauss-reduced basis

The closer δ gets to 1 the more reduced is the basis, but LLL-reduced is a weaker term of reduction than the following ones. It finds a non-zero lattice vector of length $\leq 2^{\frac{d-1}{2}}\lambda_1$ ($\delta = 3/4$), according to that, the quality of the approximation decreases exponentially in $dim(\mathcal{L}) = d$. In [25] Schnorr and Euchner present an improvement called deep insertion and a floating point variant L^3FP which is implemented in the NTL. Nguyen and Stehlè present a floating point variant called L^2 which is implemented in the fpLLL library [18], which is faster than the NTL version [27].

HKZ-Reduced. Hermite, Korkine, and Zolotarev defined a strong term of reduction called HKZ. A basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ is called HKZ-reduced if it is size-reduced and $\widehat{\mathbf{b}_\mathbf{i}}$ is the shortest vector of the projected lattice $\pi_i(\mathcal{L})$, $\forall\ 1 \leq i \leq d$.

$$\left\|\widehat{\mathbf{b}_\mathbf{i}}\right\| = \lambda_i(\mathcal{L}_i),\ i = 1, \ldots, d.$$

HKZ-reduction is very strong, but no polynominal time computation is known. On the contrary, LLL-reduction is fairly cheap, but an LLL-reduced basis is of much lower quality. The next term lies in between these two.

BKZ-reduced. Schnorr and Euchner invented the BKZ-algorithm. Up to now it is the best practical known algorithm to solve the γ-SVP at least for small β. BKZ (Block Korkine-Zolotarev) combines LLL and HKZ by applying an HKZ-reduction to a block β of vectors. A basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$ is called β-reduced

with $\beta \in \{2, \ldots, d\}$ if the basis is size-reduced and

$$\{\pi_i(\mathbf{b_i}), \pi_i(\mathbf{b_{i+1}}), \ldots, \pi_i(\mathbf{b_{i+\beta-1}})\}$$

is an HKZ-reduced basis for $i = 1, \ldots, n - \beta + 1$.

The BKZ algorithm uses an enumeration called ENUM which is used as a subroutine to calculate the shortest vector from a block β of vectors. If BKZ terminates it finds a lattice vector of length $\leq \gamma_\beta^{(n-1)/(\beta-1)}$ [23], where $\gamma_\beta \sim \frac{\beta}{\pi e}$. For $\beta = 20$ BKZ is about 10 times slower than LLL, for large β the factor increases about $\beta^{\mathcal{O}(\beta)}$. This high delay factor results from the complete enumeration of all short lattice vectors over a block of size β. We see that BKZ relies on an exact SVP solver over small dimensions β. Hence, we take a deeper look on the SVP solver in the next section.

2.2 SVP Solver

Since the SVP is $\mathcal{NP}$-hard under randomized reductions [2], algorithms to solve the SVP exactly are not assumed to work in polynomial time. As already mentioned, we describe three different versions of SVP solvers.

The first two algorithms can be classified as deterministic enumeration algorithms. They are searching through an area around the origin, in which the shortest vector is guaranteed. The third is a randomized Sieve algorithm.

2.2.1 Enumeration of Fincke and Pohst

The enumeration of Fincke and Pohst described in [7] minimizes a quadratic form over an ellipsoid. Their version was a huge progress to the enumeration algorithms at that time, which used a cuboid.

Let $\mathcal{L} \subseteq \mathbb{R}^n$ be a lattice with a respective basis $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\}$. Then $\mathcal{Q} : \mathbb{R}^d \to \mathbb{R}$ is called a quadratic form, with

$$\mathcal{Q}(\mathbf{x}) := \mathbf{x} \underbrace{\mathbf{B}\mathbf{B}^\mathsf{T}}_{:=\mathbf{A}} \mathbf{x}^\mathsf{T} = \left\| \sum_{i=1}^{d} x_i \mathbf{b_i} \right\|^2 .$$

Note that $\mathbf{A}$ is positive definite, if $\mathbf{B}$ consists of independent vectors.

$\{x \in \mathbb{R}^d \mid \mathcal{Q}(\mathbf{x}) \leq C\}$ describes an ellipsoid in $\mathbb{R}^d$ for $C \in \mathbb{R}^+$. The shortest vector $\mathbf{b}$ can be calculated by $\min\{\ \mathbf{x} \mid \mathcal{Q}(\mathbf{x}) < C\ \}$, but it gets easier if we transform the quadratic form by completing the square.

We can simplify $\mathcal{Q}(\mathbf{x})$ knowing that $\mathbf{A}$ is positive definite (1) and completing the square (2) to

$$\mathcal{Q}(\mathbf{x}) \overset{(1)}{=} \sum_{i=1}^{d}\sum_{j=1}^{d} a_{ij}\mathbf{x_i}\mathbf{x_j} \overset{(2)}{=} \sum_{i=1}^{d} q_{ii}(\mathbf{x_i} + \sum_{j=i+1}^{d} q_{ij}\mathbf{x_j})^2, \tag{2.1}$$

with $q_{ij} \in \mathbb{R}$ for $1 \leq i \leq j \leq d$ and $q_{ii} > 0$ for $1 \leq i \leq d$.

To calculate the constraints for x_d we can transform (2.1) into

$$\sum_{i=1}^{d-1} q_{ii}(\mathbf{x_i} + \sum_{j=i+1}^{d} q_{ij}\mathbf{x_j})^2 + q_{dd}z_d^2 \leq C \Rightarrow$$

$$\left\lceil -\sqrt{\frac{C}{q_{dd}}} \right\rceil \leq \mathbf{x_d} \leq \left\lfloor \sqrt{\frac{C}{q_{dd}}} \right\rfloor$$

According to that, one can recursively calculate for given $\mathbf{x_{t+1}}, \ldots, \mathbf{x_d}$ constraints for $\mathbf{x_t}$ $(1 \leq t \leq d)$.

Backes [6] showed that the version from Fincke and Pohst is equal to the one from Schnorr and Euchner (without improvements). Hence, we will not go into detail and concentrate on the version by Schnorr and Euchner. For more details and the algorithm see [6, 7].

2.2.2 ENUM of Schnorr and Euchner

Schnorr and Euchner introduced their enumeration algorithm called ENUM in [25] as a subroutine of the BKZ. Instead of minimizing a quadratic form, they use the following equation 2.2.

Let $\{\mathbf{b_1}, \ldots, \mathbf{b_d}\} \in \mathbb{R}^n$ be a lattice basis with the related orthogonalization $\{\widehat{\mathbf{b}}_\mathbf{1}, \ldots, \widehat{\mathbf{b}}_\mathbf{d}\}$, $c_i = ||\widehat{\mathbf{b}}_\mathbf{i}||^2$ and Gram-Schmidt-coefficients μ_{ij}, such that $\mathbf{b_i} = \sum_{j=1}^{i} \mu_{ij}\widehat{\mathbf{b}}_\mathbf{j}\ \forall\ i = 1, \ldots, d$. ENUM minimizes for $(\tilde{u}_1, \ldots, \tilde{u}_d) \in \mathbb{Z}^d \setminus \{0\}$

$$\tilde{c}_t := c_t(\tilde{u}_t, \ldots, \tilde{u}_d) := \left\| \pi_t(\sum_{i=t}^{d} \tilde{u}_i\mathbf{b_i}) \right\|^2 := \sum_{i=t}^{d} (\sum_{j=t}^{i} \tilde{u}_i \cdot \mu_{ij})^2 \cdot c_j. \tag{2.2}$$

The algorithm searches for a non-zero integer vector $\tilde{\mathbf{u}}$ of temporary minimal reduction coefficients, which satisfies $c_j(\tilde{u}_1, \ldots, \tilde{u}_d) < A$, with the current minimum stored in A.

Therefore ENUM traverses a search tree in depth first search, as Figure 2.6 shows.

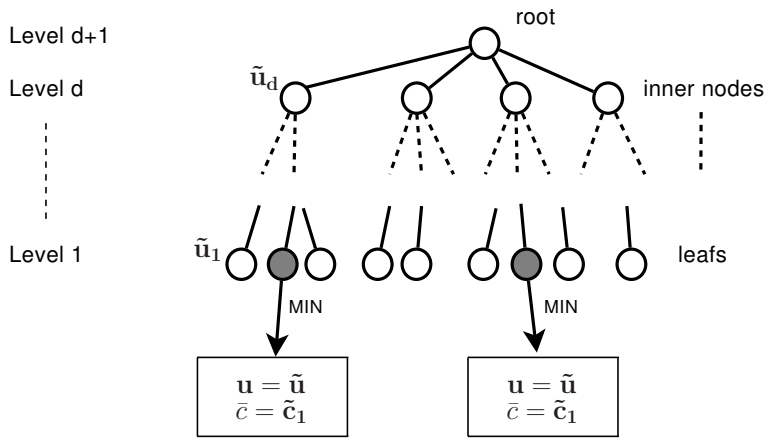


Figure 2.6: ENUM traversation of a search tree in depth first order. Showing the values to updated if a local minimum is found

ENUM starts with $\tilde{\mathbf{u}} = (1, 0, \ldots, 0)$, meaning that the first minimum only depends on $\tilde{c}_1$, $A = \tilde{c}_1 = \mathbf{c_1}$, which is the squared norm of the first basis vector.

In each iteration t $(1 \leq t \leq d)$ ENUM tests whether the current $\tilde{c}_t$ is still smaller than A, whereas $\tilde{\mathbf{u}}$ looks like

$$(\times, \ldots, \times, \underbrace{\tilde{u}_t, \ldots, \tilde{u}_d}_{fixed}).$$

If $\tilde{c}_t < A$ the algorithm steps down $t \to t-1$ and calculates $\tilde{u}_{t-1}$. If the test fails the algorithm steps up $t \to t+1$, meaning that the examined branch can be cut off, since $\tilde{c}_t \leq \tilde{c}_{t-1},\ \forall\ 1 \leq t \leq d$.

As level 1 is reached the whole $\tilde{\mathbf{u}}$ is available, if the test is still successful a new minimum A is found. To avoid redundancy $\tilde{u}_i$ is set > 0 for the largest i with $\tilde{u}_i \neq 0$.

ZIG-ZAG-PATH. An improvement compared to the version from FINCKE and POHST is called zig-zag-path. For fixed $(\tilde{u}_t, \ldots, \tilde{u}_d)$ the next value to be calculated

is $\tilde{u}_{t\pm1}$. Assume without loss of generality that the next level is $\tilde{u}_{t-1}$.

FINCKE and POHST looked at it in a straight increasing fashion, the zig-zag-path chooses $\tilde{u}_{t-1}$ from the center of the interval and takes all upper and lower values into consideration afterwards. This has the same effect as sorting all possible $\tilde{u}_{t-1}$ in a non deceasing sequence. We define $y_t := \sum_{i=t+1}^{s} \tilde{u}_i \mu_{it}$ and $\mathbf{v_{t-1}} = \lceil -y_{t-1} \rfloor$. The first choice of $\tilde{u}_{t-1} = \mathbf{v_{t-1}}$, which is the center of the interval to be chosen from.

Afterwards ENUM generates a sequence $\mathbf{v_{t-1}} \pm 1$, $\mathbf{v_{t-1}} \pm 2$, $\mathbf{v_{t-1}} \pm 3, \ldots$ until the end of the interval is reached. The two different possibilities are shown in Figure 2.7.

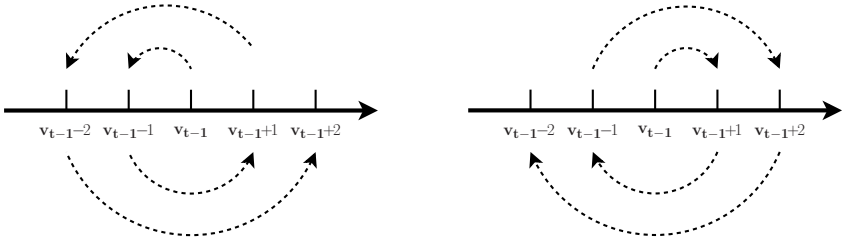


Figure 2.7: The two different possibilities of the zig-zag path

Since we know that $\tilde{c}_{t-1} < A$ we can formulate the constraints of the interval.

$$\tilde{c}_t + (y_{t-1} + \tilde{u}_{t-1})^2 \cdot \mathbf{c_{j-1}} < A$$
$$\iff \tilde{u}_{t-1} < \pm\sqrt{\frac{A - \tilde{c}_t}{\mathbf{c_{j-1}}}} - y_{t-1}.$$

This new control sequence chooses the path which guarantees most success first, so the value of A may be decreased. Thus, more cut offs will occur, which speeds up the enumeration. However, to implement the zig-zag-path ENUM needs the following additional variables: $\mathbf{v}, \Delta, \delta$.

Further on we will give a detailed description of the ENUM algorithm and show a short example. Afterwards we will introduce another improvement called *The Gaussian Volume Heuristic*.

The Algorithm uses a start stage j and a end stage k, since ENUM can be used inside BKZ. Normally BKZ loops ENUM with $k - j + 1 = 20$.

Algorithm 1: ENUM algorithm

Input: j, k with $1 \leq j \leq k \leq m$ and $\mathbf{c_j}, \ldots, \mathbf{c_m}$

Result: the minimal place $(\mathbf{u_j}, \ldots, \mathbf{u_k}) \in \mathbb{Z}^{k-j+1} \setminus \{0\}$ and the minimum A

$A := \mathbf{c_j},\ \tilde{u}_j := \mathbf{u_j} := 1,\ y_j := \Delta_j := v_j := 0,\ s := t := j,\ \delta_j := 1$
▷ *initialization*
for $i = j+1, \ldots, k+1$ **do** $\tilde{c}_i := \mathbf{u_i} := \tilde{u}_i := y_i := \Delta_i := v_i := 0,\ \delta_i := 1$
while $t \leq k$ **do**
  $\tilde{c}_t := \tilde{c}_{t+1} + (y_t + \tilde{u}_t)^2 \cdot \mathbf{c_t}$ ▷ *compute 2.2*
  if $\tilde{c}_t < A$ **then** ▷ *still smaller than the minimum?*
    if $t > j$ **then** ▷ *leaf?*
      $t := t - 1$ ▷ *inner node → step down*
      $y_t := \sum_{i=t+1}^{s} \tilde{u}_i \mu_{it},\ \tilde{u}_t := v_t := \lceil -y_t \rfloor,\ \Delta_t := 0$
      if $\tilde{u}_t > -y_t$ **then** $\delta_t := -1$ **else** $\delta_t := 1$
    else ▷ *leaf and a new minimum*
      $A := \tilde{c}_j,\ \mathbf{u} := \tilde{\mathbf{u}}$
    end
  else
    $t := t + 1$ ▷ *no minimum → make a cut and step up*
    $s := \max(s, t)$
    if $t < s$ **then** $\Delta_t := -\Delta_t$; ▷ *calculate Δ_t for the "zig-zag"-path*
    if $\Delta_t \cdot \delta_t \geq 0$ **then** $\Delta_t := \Delta_t + \delta_t$
    $\tilde{u}_t := v_t + \Delta_t$ ▷ *compute the next $\tilde{\mathbf{u}}$*
  end
end

Example. In Figure 2.8 we list all examined vectors $\tilde{\mathbf{u}}$ in a search tree. The table 2.2.2 gives a step-by-step account of the ENUM progress. ENUM outputs $\mathbf{u} = (-1\ 0\ 0\ 1)$ as the global minimum. Hence, the shortest lattice vector is $\mathbf{b} = (1\ 1\ 1\ 0)$ with $||\mathbf{b}|| = 3$.

Input: Let (j, k) be $(1, 4)$ to enumerate over the whole basis, with $\mathbf{B} = \begin{pmatrix} 2 & 4 & 1 & 3 \\ 6 & 3 & 8 & 7 \\ 8 & 0 & 2 & 9 \\ 1 & 2 & 0 & 1 \end{pmatrix}$.

The GSO outputs

$$\mu \approx \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0,2\bar{6} & 1 & 0 & 0 \\ 0,628 & 0,482 & 1 & 0 \\ 1,152 & 0,126 & -0,016 & 1 \end{pmatrix} \text{ and } \mathbf{c} \approx \begin{pmatrix} 105 & 21,5\bar{3} & 22,491 & 0,208 \end{pmatrix}$$

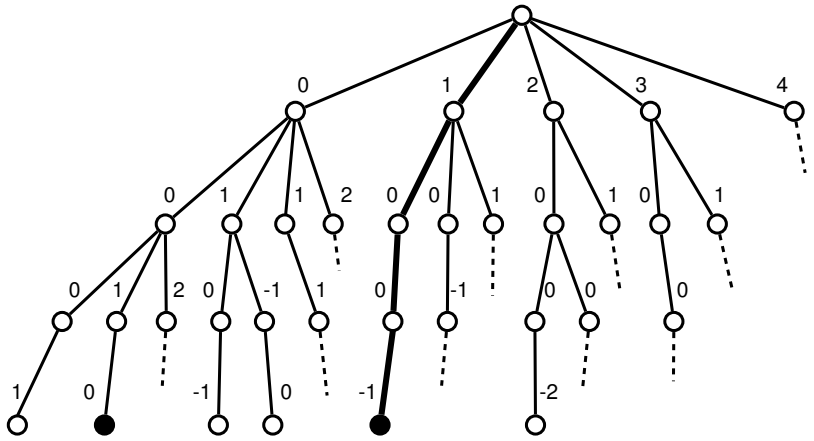


Figure 2.8: Example of an ENUM search tree. The branch of the global minimum is drawn bold

Branches which cause a cut off are drawn with a dotted line in Figure 2.8 and specified with a $\uparrow$ in Table 2.2.2. Local minimum nodes are filled black, whereas they are marked in the table with $leaf$. The last local minimum is the global minimum $\mathbf{u}$, which is found in iteration 16. It is drawn bold in Table 2.2.2 and Figure 2.8.

ENUM terminates because in iteration 30 $\tilde{c}_4 = 3.3 > 3 = A$, meaning that for all $\tilde{u}_4 \geq 4 \quad (\tilde{u}_4 \cdot \mu_{44})^2 \cdot \widehat{\mathbf{b}_4} > A$.

ENUM iterations:

Iteration	t	A	$\tilde{c}_t$	$\tilde{c}_t < A$?	updates	$\tilde{u}_d, \ldots, \tilde{u}_1$
–	–	105	–	–	0 0 0 1	–
1	1	105	105	no $\uparrow$	$t = 2, s = 2, \Delta_t = 1, \tilde{u}_t = 1$	0 0 1 $\perp$
2	2	105	22	yes $\downarrow$	$t = 1, y_t = 0.27, \tilde{u}_t = 0, \Delta_t = 0$	0 0 1 0
3	1	105	29	yes $\downarrow$	$leaf, A = 29, u = 0100$	–
4	1	29	29	no $\uparrow$	$t = 2, s = 2, \Delta_t = 2, \tilde{u}_t = 2$	0 0 2 $\perp$
5	2	29	86	no $\uparrow$	$t = 3, s = 3, \Delta_t = 1, \tilde{u}_t = 1$	0 1 $\perp\perp$
6	3	29	22	yes $\downarrow$	$t = 2, y_t = 0.48, \tilde{u}_t = 0, \Delta_t = 0$	0 1 0 $\perp$
7	2	29	28	yes $\downarrow$	$t = 1, y_t = 0.63, \tilde{u}_t = -1, \Delta_t = 0$	0 1 0 -1
8	1	29	42	no $\uparrow$	$t = 2, s = 3, \Delta_t = -1, \tilde{u}_t = -1$	0 1 -1 $\perp$
9	2	29	28	yes $\downarrow$	$t = 1, y_t = 0.36, \tilde{u}_t = 0, \Delta_t = 0$	0 1 -1 0
10	1	29	42	no $\uparrow$	$t = 2, s = 3, \Delta_t = 1, \tilde{u}_t = 1,$	0 1 1 $\perp$
11	2	29	70	no $\uparrow$	$t = 3, s = 3, \Delta_t = 2, \tilde{u}_t = 2,$	0 2 $\perp\perp$
12	3	29	90	no $\uparrow$	$t = 4, s = 4, \Delta_t = 1, \tilde{u}_t = 1,$	1 $\perp\perp\perp$
13	4	29	0.21	yes $\downarrow$	$t = 3, y_t = -0.017, \tilde{u}_t = 0, \Delta_t = 0$	1 0 $\perp\perp$
14	3	29	0.21	yes $\downarrow$	$t = 2, y_t = 0.13, \tilde{u}_t = 0, \Delta_t = 0$	1 0 0 $\perp$
15	2	29	0.56	yes $\downarrow$	$t =, y_t = 1.2, \tilde{u}_t = -1, \Delta_t = 0$	1 0 0 -1
16	**1**	**29**	**3**	**yes** $\downarrow$	$\mathbf{leaf}, \mathbf{A = 3}, \mathbf{u = -1001}$	–
17	1	3	3	no $\uparrow$	$t = 2, s = 2, \Delta_t = -1, \tilde{u}_t = -1$	1 0 -1 $\perp$
18	2	3	17	no $\uparrow$	$t = 3, s = 4, \Delta_t = 1, \tilde{u}_t = 1$	1 1 $\perp\perp$
19	3	3	22	no $\uparrow$	$t = 4, s = 4, \Delta_t = 2, \tilde{u}_t = 2$	2 $\perp\perp\perp$
20	4	3	0.83	yes $\downarrow$	$t = 3, y_t = -0.034, \tilde{u}_t = 0, \Delta_t = 0$	2 0 $\perp\perp$
21	3	3	0.86	yes $\downarrow$	$t = 2, y_t = 0.25, \tilde{u}_t = 0, \Delta_t = 0$	2 0 0 $\perp$
22	2	3	2.2	yes $\downarrow$	$t = 1, y_t = 2.3, \tilde{u}_t = -2, \Delta_t = 0$	2 0 0 -2
23	1	3	12	no $\uparrow$	$t = 2, s = 4, \Delta_t = -1, \tilde{u}_t = -1$	2 0 -1 $\perp$
24	2	3	13	no $\uparrow$	$t = 3, s = 4, \Delta_t = 1, \tilde{u}_t = 1$	2 1 $\perp\perp$
25	3	3	22	no $\uparrow$	$t = 4, s = 4, \Delta_t = 3, \tilde{u}_t = 3$	3 $\perp\perp\perp$
26	4	3	1.9	yes $\downarrow$	$t = 3, y_t = -0.05, \tilde{u}_t = 0, \Delta_t = 0$	3 0 $\perp\perp$
27	3	3	1.9	yes $\downarrow$	$t = 2, y_t = 0.38, \tilde{u}_t = 0, \Delta_t = 0$	3 0 0 $\perp$
28	2	3	5.1	no $\uparrow$	$t = 3, s = 4, \Delta_t = 1, \tilde{u}_t = 1$	3 1 $\perp\perp$
29	3	3	22	no $\uparrow$	$t = 4, s = 4, \Delta_t = 4, \tilde{u}_t = 4$	4 $\perp\perp\perp$
30	4	3	3.3	no $\uparrow$	$t = 5, s = 4, \Delta_t = 1, \tilde{u}_t = 1$	$\perp\perp\perp\perp$

The Gaussian Volume Heuristic. In [11, 26] Schnorr and Hörner introduced another improvement called the Gaussian Volume Heuristic, which is a pruning technique most likely finding the shortest vector.

It is based on a general principle introduced by Gauss. Let $S \subset span(\mathcal{L})$ than

$$\frac{vol(S)}{\det(\mathcal{L})} \tag{2.3}$$

estimates the number of points in $S \cap \mathcal{L}$.

We are going to apply 2.3 to ENUM. Let $\bar{\mathcal{L}}$ denote the lattice $\mathcal{L}(\mathbf{b_1}, \ldots, \mathbf{b_{t-1}})$. Assuming ENUM stands on stage t with fixed

$$(\tilde{u}_t, \ldots, \tilde{u}_d) \in \mathbb{Z}^{d-t+1} \setminus \{0\} \quad \text{and} \quad \mathbf{b} = \sum_{i=t}^{n} \tilde{u}_i \mathbf{b_i}$$

we search for

$$(\tilde{u}_1, \ldots, \tilde{u}_{t-1}) \in \mathbb{Z}^{t-1} \setminus \{0\} \quad \text{and} \quad \bar{\mathbf{b}} = \sum_{i=1}^{t-1} \tilde{u}_i \mathbf{b_i} \in \bar{\mathcal{L}} \tag{2.4}$$

satisfying $\tilde{c}(\tilde{u}_1, \ldots, \tilde{u}_d) < A$ or $\left\|\mathbf{b} + \bar{\mathbf{b}}\right\|^2 < A$.

We split $\mathbf{b}$ into orthogonal parts regarding $span(\bar{\mathcal{L}})$:

$$\mathbf{b} = \underbrace{\sum_{j=1}^{t-1} \sum_{i=t}^{d} \tilde{u}_i \mu_{ij} \widehat{\mathbf{b}_\mathbf{j}}}_{:=-\zeta_t \in span(\bar{\mathcal{L}})} + \underbrace{\sum_{j=t}^{d} \sum_{i=t}^{d} \tilde{u}_i \mu_{ij} \widehat{\mathbf{b}_\mathbf{j}}}_{:=y \in span(\bar{\mathcal{L}})^\perp}$$

According to that, we can specify 2.4: We are searching for $\bar{\mathbf{b}} \in \bar{\mathcal{L}}$ in

$$(\mathbf{b} + \bar{\mathcal{L}}) \cap \mathcal{B}_{t-1}(y, \sqrt{A - \tilde{c}_t}) = \bar{\mathcal{L}} \cap \mathcal{B}_{t-1}(\zeta_t, \sqrt{A - \tilde{c}_t}).$$

Since $\mathcal{B}_{t-1}(\zeta_t, \sqrt{A - \tilde{c}_t}) \subset span(\bar{\mathcal{L}})$ we can apply 2.3:

$$\begin{aligned} \beta_t :=& E[\ |\{(\tilde{u}_1, \ldots, \tilde{u}_{t-1})\} \in \mathbb{Z}^{t-1} : c_j(\tilde{u}_1, \ldots, \tilde{u}_n) < A\}|\ ] \\ =& \frac{\mathrm{vol}(\mathcal{B}_{t-1}(\zeta_t, \rho_t))}{\det \bar{\mathcal{L}}}, \text{ with } \quad \rho_t^2 = (A - \tilde{c}_t). \end{aligned}$$

Regarding to this SCHNORR and HÖRNER propose to cut off the enummeration if $\beta_t < 2^p$, where the parameter $0 \leq p < 1$ can be freely chosen. [11] suggests $p = 13$.

Note that the smaller p gets, the more the probability increases that the shortest vector is the one that is already found.

To adapt the heuristic to ENUM we have to determine the radius of $\mathcal{B}_t - 1$ for a fixed parameter p and t.

With the help of the Stirling approximation (1) we can approximate:

$$\begin{aligned} 2^{-p} =& \frac{\mathrm{vol}(\mathcal{B}_{t-1}(\varepsilon_{t,p}))}{\det \bar{\mathcal{L}}} \\ \overset{(1)}{=}& \frac{(\pi \cdot \varepsilon_{t,p})^{(t-1)/2}}{\Gamma(1 + (t-1)/2)} \cdot \frac{1}{\prod_{i=1}^{t-1} ||\widehat{\mathbf{b}_\mathbf{i}}||} \\ \approx& \frac{(\frac{2e\pi}{t-1} \cdot \varepsilon_{t,p})^{(t-1)/2}}{\sqrt{\pi(t-1)} \cdot \prod_{i=1}^{t-1} ||\widehat{\mathbf{b}_\mathbf{i}}||}. \end{aligned}$$

Thus $\varepsilon_{t,p} \approx \frac{t-1}{2e\pi}(\sqrt{\pi(t-1) \cdot 2^{-p}} \cdot \prod_{i=1}^{t-1} ||\widehat{\mathbf{b}_\mathbf{i}}||)^{2/(t-1)}$.

Hence, the following line has to be changed:

Algorithm 2: Gauss-ENUM algorithm

if $\tilde{c}_t < A - \varepsilon_{t,p}$ **then** ...; ▷ *change in line 5, adding the new heuristic parameter* $\varepsilon_{t,p}$

RUNNING TIME. The actual running time of ENUM depends on the quality of the input basis, since the running time is roughly proportional to the number of lattice points of the region to be explored. [9] compared ENUM with different reduced basis, like LLL and BKZ. If the input basis is only LLL-reduced ENUM outputs the shortest vector in $2^{O(d^2)}$ polynomial time operations. They also showed that a 60-dimensional lattice can be solved within an hour, but dimension 100 would take at least 35000 years. A stronger preprocessing, like BKZ, reduces the running time a bit, but dimension 100 seems out of reach.

2.2.3 AKS Sieve Algorithm

The AKS Sieve algorithm was introduced by Ajtai, Kumar and Sivakumar in [4]. It uses a randomization instead of a complete enumeration which effects the running time to a simple exponentional function $2^{O(d)}$. However, there is a huge drawback since AKS requires exponential space. In [19] Nguyen and Vidick suggest a practical variant, but it is not competitive with ENUM at least up to dimension 50, thus making it already impractical.

The latest version is presented by Micciancio and Voulgaris in [17]. Their new probabilistic algorithm List Sieve finds the shortest vector in any n dimensional lattice in $O(2^{3.199d})$ time needing $O(2^{1.325d})$ space. The practical variant Gauss Sieve also introduced in [17] grants much better space bounds and outperforms the best previous practical implementation [19] of the AKS Sieve. We will give a short overview of the AKS variants, *List Sieve* and *Gauss Sieve*, presented by Micciancio and Voulgaris.

The List Sieve Algorithm. The algorithm starts with an empty list L of lattice points. In each iteration the subfunction *Sample* generates a new lattice point $\mathbf{v}$. Before adding $\mathbf{v}$ to L another subfunction *ListReduce* reduces $\mathbf{v}$ if appropriate by repeatedly subtracting all lattice vectors already in L, ensuring that none of the points in L are close to each other. Finally, $\mathbf{v}$ is included in L if it is not already in L, so no collision exists. Points already in L never change.

The Samples are limited to K rounds (K will be specified below: running time and space requirement). The algorithm terminates successfully if two lattice vectors $\mathbf{v_i}$, $\mathbf{v_j}$ within distance μ are found, $||\mathbf{v_i} - \mathbf{v_i}|| < \mu$, otherwise it outputs $\perp$. μ is one of the input variables.

Further, we show how *Sample* exactly works. It uses the same method as the original Sieve algorithm [4]. Instead of working with the lattice point $\mathbf{v}$ directly, it uses a perturbed version $\mathbf{p} = \mathbf{v} + \mathbf{e}$.

Sample algorithm. It samples a pair $(\mathbf{p}, \mathbf{e})$. $\mathbf{e}$ is chosen uniformly at random within a sphere of radius $\xi\mu$. Setting $\mathbf{p} = \mathbf{e} \mod \mathbf{B}$ guarantees that the sphere $\mathcal{B}_{\xi\mu}(\mathbf{p})$ contains at least one lattice point $\mathbf{v} = \mathbf{p} - \mathbf{e}$. Note that $\mathbf{v}$ (given $\mathbf{p}$) is uniformly distributed over all lattice points in $\mathcal{B}_{\xi\mu}(\mathbf{p})$. Moreover, the probability for any $\xi > 0.5$, $\mathcal{B}_{\xi\mu}(\mathbf{p})$ containing more than one lattice point, is strictly positive.

[17] showed that for larger values of ξ the size of L increases, whereas the probability rises that the algorithm produces collisions if ξ gets smaller. Below we will discuss how to choose ξ.

List Reduction. For each $\mathbf{v} \in L$ the function ListReduce reduces a vector $\mathbf{p}$ by subtracting $\mathbf{v}$ from $\mathbf{p}$ if $||\mathbf{p} - \mathbf{v}|| < \delta||\mathbf{p}||$, where $\delta < 1$ is used as a slackness parameter to subtract $\mathbf{v}$ from $\mathbf{p}$ only if this reduces $||\mathbf{p}||$ at least with a factor δ.

Note that, the vectors in L can be in any order and reducing $\mathbf{p}$ with $\mathbf{v_i} \in L$ and afterwards with $\mathbf{v_j} \in L$ may affect that $\mathbf{p}$ is no longer reduced according to $\mathbf{v_i}$. Therefore all list vectors are considered repeatedly until $||\mathbf{p}||$ cannot be reduced any more. Since ListReduce terminates after a finite number of iterations, [17] suggests to choose $\delta(n) = 1 - 1/n$, so that the number of iterations is bounded by a polynomial $n^{O(1)}$.

The Gauss Sieve. Micciancio and Voulgaris also introduced a practical variant called the Gauss Sieve, which is said to have much better space complexity.

One main difference affects the reduction function: not only the list vector $\mathbf{v}$ to be added is reduced, but also the vectors in the list using $\mathbf{v}$. More precisely, if $\min(||\mathbf{v} \pm \mathbf{u}||) < \max(||\mathbf{v}||, ||\mathbf{u}||)$ then the longer of $\mathbf{v}, \mathbf{u}$ will be replaced by the shorter of $\mathbf{v} \pm \mathbf{u}$. Regarding to that, L will always consist of pairwise reduced vectors, satisfying $\min(||\mathbf{v} \pm \mathbf{u}||) \geq \max(||\mathbf{v}||, ||\mathbf{u}||)$ which is the definition of a Gauss reduced basis for two dimensional lattices.

Moreover, Gauss Sieve does not work with error vectors: $\mathbf{p} = \mathbf{v}$ is chosen randomly. This allows an integer only implementation, since only lattice points are considered. Unfortunately, no upper bound for the number of samples exists, so that a heuristic termination condition based on experiments is used, which terminates after a certain number of collisions.

Running time and space requirement. List Sieve: The running time depends on the list size and the collision probability. Therefore ξ has to be chosen as an appropriate trade-off between keeping both the list size and the collision probability small.
[17] proved the following bounds for space and time complexity: The number of points in L is limited by $N = poly(d) \cdot 2^{c_1 d}$, with $c_1 = \log(\xi + \sqrt{\xi^2 + 1}) + 0.401$.

List Sieve outputs a lattice point $\mathbf{s}$ with $||\mathbf{s}|| < \mu$ (if it exists) using $K = O(2^{(c_1+c_2)d})$ samples with high probability, where $c_2 = \log(\xi/\sqrt{\xi^2 - 0.25})$.

Gauss Sieve: Since we know that the angle between $\mathbf{v}, \mathbf{u}$ is at least $\pi/3$ (or 60 degree), we can bound the size of L to the kissing number τ_n. Unfortunately, [17] cannot prove any interesting bounds on the number of samples, nor on the running time of Gauss Sieve.

But their experiments show that Gauss Sieve outperforms ENUM for dimension $\gtrapprox 40$. This is a huge improvement compared to the version from [19], which was incommensurable to ENUM.

3 Artificial Intelligence

There is an immense range of problems solvable through search in artificial intelligence. We can distinguish between so-called toy problems and so-called real-world problems. Toy problems are used to illustrate or exercise various problem-solving methods. Whereas real-world problems tend to be more complex and not unique, for example a robot navigation or route finding for air plains, which does not only depend on the shortest route. In this paper we will concentrate on toy problems, since we are only interested in search strategies itself.

Afterwards we will take a look on two-player games, especially chess. ENUM is not comparable with two-player games in the point of two players, but we will see that many used techniques (search strategies and heuristics) are transferable to strategies which ENUM already uses or which can be adapted.

3.1 Search Strategies

In artificial intelligence a game tree is a directed graph whose nodes are positions in a game and whose edges represent moves. We need to distinguish between a node and a state. In this section we assume that a node is a data structure consisting of five components: the *state* in the state space to which the node corresponds, the *parent node*, an *operator* that was applied to generate the node, *depth* of this node, the *path cost* so far. On the contrary a state represents a configuration of the "world". For this reason it is possible that two different nodes contain the same state, if this state is generated via two different sequences.

For example we look on the following toy problem on Figure 3.1: 8-puzzle, which belongs to the family of sliding-block-puzzles.

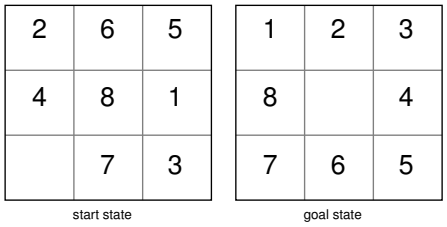


Figure 3.1: Showing an example of the 8-Puzzle with start- and end-stage

States: a state is described by the value of each inner box.

Operators: movement of the blank box in one of the following directions: left, right, up, down.

Path cost: each movement costs 1, so the path cost is just the length of the path.

Search strategies can be classified into uninformed and informed. Uninformed means that they have no information about the number of steps or the path cost from the current state to the goal. Informed search strategies are often also called heuristic search, since they are using problem-specific heuristic information. Each algorithm will be analyzed regarding four categories: Time/ space complexity, completeness, and optimality. If an algorithm is guaranteed to find a solution (if one exits) it is said to be complete. Whereas the algorithm is optimal if the highest-quality solution is found, if several different solutions exist. All strategies start with an initial state (root node) and end if a solution is found.

3.1.1 Uninformed Strategies

Since most uninformed strategies are well known we will only give a brief description.

BFS. Breath-first search is one of the simpliest strategies. At first all nodes at depth d will be expanded, afterwards their successors at depth $d+1$. If a solution exists BFS is guaranteed to find it. If more solutions exist it finds the shallowest one. Given a branching factor b and a graph depth d the time complexity bound is $O(b^d)$. The space complexity is the same since all leaf nodes of the tree have to be maintained in memory at the same time.

UCS. The uniform cost search differs from BFS in always expanding the lowest-cost (measured by the path cost $g(n)$) node first rather than the lowest-depth

node. BFS is UCS with $g(n) = depth(n)$. If $g(successor(n)) \geq g(n)$ ($\forall$ nodes n) UCS always finds the cheapest solution, thus being complete and optimal. The complexity bounds are equal to one from BFS.

DFS. Starting from the root depth first search explores all successors as far as possible. If a dead end is reached (not the goal state) DFS goes back and explores nodes at a shallower level. DFS needs to store only one single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. The time complexity is $O(b^m)$ and only requires storage of $b \cdot m$ nodes, where m is the maximum depth of the search tree. Since some problems may have very deep or infinite search trees, DFS gets stuck exploring the wrong path. Besides it may find a solution path that is longer than the optimum. Causing DFS to be neither complete nor optimal.

DLS. Depth-limited search avoids the drawback to get stuck going down the wrong path if searching with DFS. DLS is the same as DFS with a limited depth l, according to that, it takes $O(b^l)$ time and $O(bl)$ space.

IDS. Iterative deepening search is similar to DLS, but it increases the maximal depth about 1 each round. In effect, IDS combines the benefits of DFS and BFS. It may seem wasteful because so many states are expanded multiple times, it expands about 11% more nodes than a single BFS or DFS search. However, IDS is used when there is a large search space and the depth of the solution is not known.

Bidirectional search. As the name implies bidirectional search searches simultaneously forward from the initial state and backward from the goal until both searches meet. If the branching factor b is the same in both directions bidirectional search can make a huge improvement. Assuming the solution lies in depth d, the forward and backward search only has to go half way: only $O(2b^{d/2}) = O(b^{d/2})$ steps. Nevertheless, this sounds good in theory, but there are several games specific issues which need to be addressed, for example: operators have to be reversible, predecessor and successor sets must be identical and both easily computable. The complexity $O(b^{d/2})$ also assumes that testing if a node has already been found by the opposite side can be done in constant time. Bidirectional search needs $O(b^{d/2})$ space since at least all nodes from one of the searches have to stay in memory for comparison.

3.1.2 Informed Srategies

In contrast to uninformed strategies which systematically generate new states and therefore mostly inefficient we will look on strategies which use problem-specific knowledge and thus are more efficient.

For this purpose we need an evaluation function which estimates the goodness of all nodes to be expanded next. Hence, we can order those nodes and choose the one with the best evaluation value first. This strategy is called best-first search.

Greedy search. In most situations the cost of reaching the goal from a particular state cannot be determinated exactly only estimated. We call $h(n)$ an heuristic function, which estimates the cost of the current node n to the goal. A best-first search using $h(n)$ is called greedy search. $h(n)$ can be any function with $h(goal) = 0$ which depends on the particular problem. The time complexity is like DFS: $O(b^m)$, but a good heuristic can give a dramatic improvement. The space complexity is $O(b^m)$, because all nodes have to be kept. Unfortunately greedy search is incomplete and not optimal, as it can get stuck in an infinite path like DFS.

A^*. Another famous best-search algorithm is called A^*. A^* combines greedy search and USC, hence it minimizes the total path cost. Therefore both heuristic functions are combined: $f(n) = g(n) + h(n)$, whereas $h(n)$ has to be admissible meaning never overestimating the costs to reach the goal formally:

Definition 3.1 (admissible heuristic function) *Let $h(n) \geq 0$ ($\forall$ nodes n) be a heuristic function and $h^*(n)$ the true cost from n to the goal $h(n) \leq h^*(n)$. We call $h(n)$ admissible if*

$$h(n) \leq h^*(n).$$

The key of an A^* algorithm is to find a good $h(n)$. A^* search is optimal if $h(n)$ is consistent and complete for graphs with a finite branching factor provided there is some positive constant δ such that every operator costs at least δ. The time complexity depends on the heuristic. Let $h^*(n)$ denote the actual path cost, the number of nodes grows exponentially unless $|h(n) - h^*(n) \leq O(log(h^*(n))|$. However, the space complexity is the main drawback of A^* since all nodes have to be kept in memory.

By using the prior 8-puzzles example we demonstrate two possible admissible heuristic functions.

h_1 = the number of inner boxes which are in the wrong position.
h_2 = the sum of the distance of the inner boxes from their goal state (Manhattan distance).

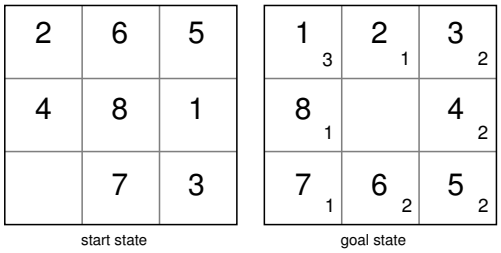


Figure 3.2: Showing an example of the Manhattan distance in the 8-Puzzle

h_1 is admissible since all wrong boxes have to be moved at least once to get into the goal position. h_2 is admissible, because any move can only move one box one step closer to the goal position. In our case: $h_1 = 8$ and $h_2 = 3+1+2+2+2+2+1+2 = 15$. Figure 3.1.2 illustrates the Manhattan distance for each inner box.

A^* has a few variants to decrease the space needed, we are going to shortly discuss one of them.

IDA^*. Iterative deepening A^* search is a mixture of IDS and A^*. Each iteration is an IDS search but rather using a depth limit it uses an f-cost limit. Once all nodes are expanded with $f(n) \leq f$, a new iteration starts with a new f-cost limit. IDA^* is complete and optimal. Assuming δ is the smallest operator cost and f^* the optimal solution cost then IDA^* needs to store $\frac{bf^*}{\delta}$ nodes in the worst case. (On average: bd). The time complexity strongly depends on the number of iteration. On average IDA^* needs two or three iterations and since the last iteration expands roughly the same number of nodes as A^* its efficiency is similar to A^*.

3.2 Techniques Used in Chess

This section deals with the most important techniques (algorithms and heuristics) used in chess. First of all we briefly introduce how to adapt two players in a

search tree. Further, we concentrate on the algorithms and heuristics used to play chess.

3.2.1 Preliminaries

Two-player games are divided into different categories. In this thesis we concentrate on zero-sum games with perfect information, because they mostly match on the mechanics of ENUM. Perfect information is reached if both players are aware of all previous moves. Chess and tic-tac-toe are examples for games with perfect information, whereas in most card games, like poker, some informations are hidden. Zero-sum describes a situation in which one player's gain is the other player's loss.

An *AND/OR-trees* models the behavior of a two-player game. Each node n represents a position in a game. Starting from the root, all nodes on one depth are alternately either an AND- or an OR- node, representing Player 1 (MAX-Player) or Player 2 (MIN-Player). An example is shown in Figure 3.3. MAX is moving from the white nodes, MIN is moving from the black nodes.

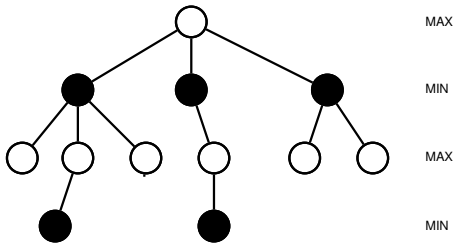


Figure 3.3: AND-OR Tree modeling the behavior of two different players

Games are about winning and loosing, therefore we need an evaluation function $eval(l)$ to decide who wins. Each leaf l represents an end-stage of the game, so that $eval(l)$ can be computed in all leafs. Talking about MAX's point of view, he tries to maximize $eval(l)$, whereas MIN tries to minimize it.

Knowing the value of $eval(l)$ for all leafs, we can recursively rate all inner nodes as well as the root. This rating is called *minimax value*. It determines the evaluation which can be archived if the opponent plays perfectly.

Definition 3.2 (minimax value)

$$v(n) = \begin{cases} eval(n) & \text{if } n \text{ is leaf} \\ max\{v(r) \mid r \in successors(n)\} & \text{if } n \text{ is a MAX-node} \\ min\{v(r) \mid r \in successors(n)\} & \text{if } n \text{ is a MIN-node.} \end{cases}$$

For an example, see Figure 3.4. It shows the minimax value in the game tic-tac-toe. We used:

$$eval(l) = \{1, 0, -1\} \mathrel{\widehat{=}} \{\text{MAX wins, draw, MAX looses}\}$$

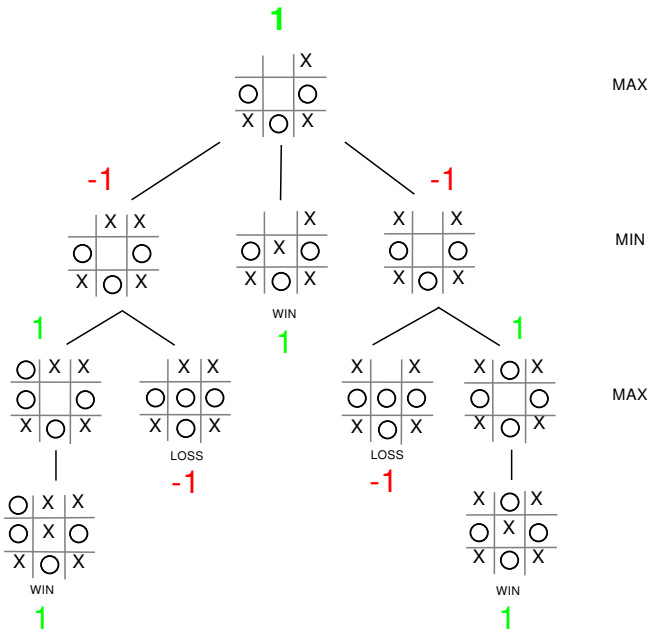


Figure 3.4: An example for minimax values in the game tic-tac-toe

3.2.2 Algorithm

The *minimax algorithm* is the basis for many search algorithms in zero-sum board games, especially for chess. The algorithm works recursively, and computes the minimax value for all inner nodes. It is a brute force, depth-first-search algorithm starting from the left branch to examine all possible movements.

The sequence of moves which are considered best and therefore expected to be played is called the *principal variation*. Therefore minimax has to store the moves which return the best minimax value.

Since we are searching in zero-sum games, we can simplify the minimax value. Knowing that $min\{a, b\} = -max\{-a, -b\}$ we can formulate a value called *negamax value*:

Definition 3.3 (negamax value)

$$n(n) = \begin{cases} f(n) & \textit{if } n \textit{ is leaf} \\ max\{v(r) \mid r \in successors(n)\} & \textit{if } n \textit{ is an inner node} \end{cases},$$

with

$$f(n) = \begin{cases} eval(n) & \textit{if } n \textit{ is a MAX-leaf} \\ -eval(n) & \textit{if } n \textit{ is a MIN-leaf} \end{cases}$$

Negamax works just like minimax, but without inner nodes differentiation into MIN or MAX.

Minimax and negamax play theoretically perfect. If the search tree is small both algorithms are applicable, for example in tic-tac-toe.

But in games like chess it would take an enormously long time. Assuming that at each position 35 moves can be done and 40-60 moves are made in one game, minimax has to estimate about 10^{123} nodes [8]. For comparison, tic-tac-toe has about 10^6 choices until the game ends, checkers has about 10^{78} and go about 10^{170}.

Because of that, a different approach is taken. Search is done only to a fixed depth. The evaluation function is exchanged to a heuristic one. Instead of knowing which player wins, it estimates the position, returning not just win or loose, but a numerical value.

The basic idea is to search as much as possible in the quickest time. But how can you improve your search?

The *Alpha Beta Algorithm* (short $\alpha\beta$) is an extension of the minimax algorithm. It also calculates the minimax value of an AND/OR-tree, but it prunes the search tree, if the current value cannot improve the minimax value. To prune the search tree $\alpha\beta$ has to keep two values (α, β). MAX can obtain a value at least bigger

than α, whereas MIN makes sure he gets a value smaller than β. Each node is examined with a search window (α, β), initialized at the root with $(-\infty, \infty)$ and updated if a better value is found.

Cut offs:

- alpha cut off
  standing on a MIN-node, this node has more than one child, one child returns a value with $\alpha \geq value \Rightarrow$ cut off all other children
- beta cut off
  standing in a MAX-node, this node has more than one child, one child returns a value with $\beta \leq value \Rightarrow$ cut off all other children

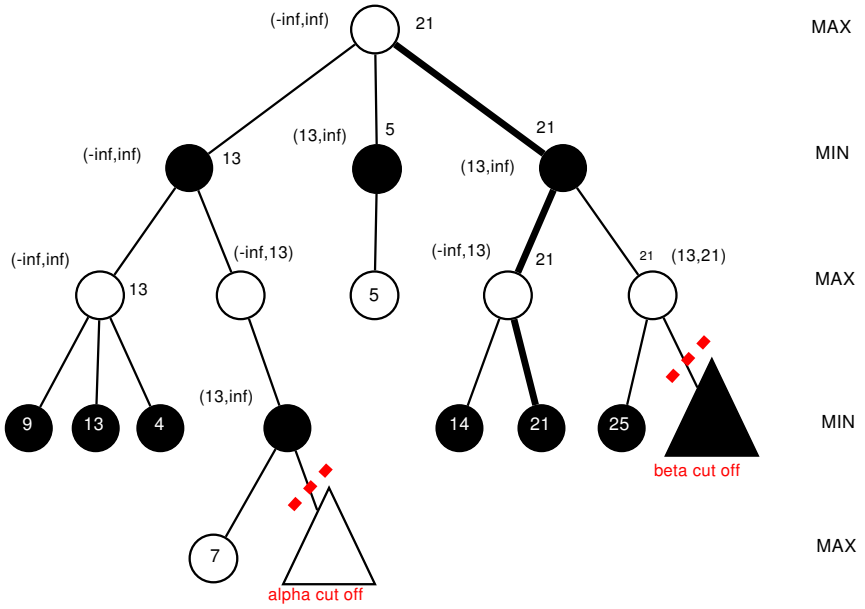


Figure 3.5: An example for an $\alpha\beta$ search tree, representing both possible cut offs: alpha and beta cut off

Both cut offs are shown in Figure 3.5. First MAX finds a leaf with $v(n) = 7$ since $\alpha = 13$ all other children can be cut off. The second cut off occurs as MIN finds $v(n) = 25$ whereas β is only 21. The principal variation is drawn bold, MAX can obtain 21.

The actual number of cut offs depends on the order of the search tree. But how can we specify a good move ordering? If $\alpha\beta$ finds the principal variation first, we know that the minimax value does not change, so we can eliminate

many branches. The subset of a search tree which is required to determine the minimax value at the root is called the *minimal* tree. Hence, only nodes from the minimal tree affect the minimax value at the root, regardless of way the other nodes are assessed. The minimal tree contains of three types of nodes: PV-, CUT- and ALL-nodes [14]:

- The root node is defined to be a PV node
- At a PV node all children have to be investigated. At least one child has the minimax value of the root. Define such child to be a PV node, and the remaining child nodes to be CUT nodes.
- At a CUT node the child causing a β-cut is an ALL node. In a perfectly ordered tree only one child of a CUT node has to be explored.
- At an ALL node all the children have to be explored. The successors of an ALL node are CUT nodes.

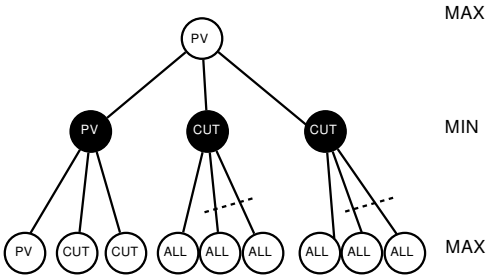


Figure 3.6: Minimal tree with 3 types of nodes: PV/ ALL / CUT

Ideas to get such a move ordering are described in 3.2.3.

Assume we have a fixed depth d and a fixed branching factor b. In the best-case $\alpha\beta$ examines $b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$ nodes. This is the minimum number of nodes that must be examined by any search algorithm to determine the minimax value. A proof and more information is given in [13].

Suppose $b = 10$ and $d = 9$:

	minimax	$\alpha\beta$
nodes	$10^9 = 1,000,000,000$	$10^5 + 10^4 = 110,000$

However, the worst-case of $\alpha\beta$ is the same as the minimax algorithm, b^d.

In [20] PEARL introduce the *SCOUT* algorithm. His idea was to decide whether a branch can improve the minimax value without knowing its exact outcome, based on the assumption that most of the subtrees will prove inferior to the best subtree searched so far.

The SCOUT-algorithm requires two boolean testfunctions $t_1(n)$ and $t_2(n)$ for MAX and MIN nodes and one evaluation $eval(n)$ function.

Let n be the root of the branch to be discovered and m the current best minimax value:

- n = MAX-node: t_1 returns *true* if $v(n) > m$ and $false$ otherwise,
- n = MIN-node: t_2 returns *true* if $v(n) < m$ and $false$ otherwise.

The evaluation function conduces to calculate the correct minimax value of a branch.

The SCOUT-algorithms uses the minimax value of the left branch as a reference value for further search, as shown in 3.7. Afterwards all following successors are tested, with t_1 or t_2. If one of the testfunctions returns true, the branch has to be examined again. In this re-search $eval(n)$ calculates the exact value. But most of the time the testfunction fails, so no re-search is needed. Because of that, the extra costs from the re-search are smaller than the savings of the efficient testfunction.

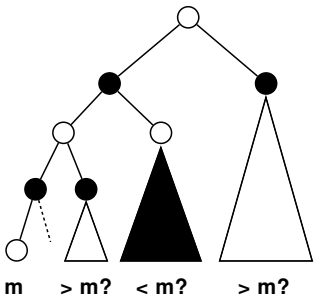


Figure 3.7: Repesenting the basic idea of the SCOUT algorithm

REINEFELD took on to that idea and invented the *NegaScout*-algorithm [21], which takes the ideas from PEARL to the $\alpha\beta$-algorithm, but as the name implies it uses a negamax-version of $\alpha\beta$. The evaluation function is done with an open

window: $(\alpha, \beta) = (-\infty, \infty)$, whereas a minimal window $(\alpha, \beta) = (v, v+1)$ represents the testfunctions, with the best available minimax value v so far. Returning a $value \leq v$ means the subtree is indeed inferior (fail-low), whereas the test fails if $value > v$ (fail-high). In this case the branch has to be re-searched with a wider search window to calculate the correct value.

As described above NegaScout, $\alpha\beta$, minimax are Depth-First-Search algorithms, a simple backtracking strategy that traverses the search tree in the same order in which successor nodes are generated. Another strategy is to search promising parts first by using specific heuristic information, they are called best-first-algorithms.

*SSS** is a non-directional algorithm to search AND-OR-trees similar to the famous A*, which has been analyzed more precisely in Section 3.1. SSS* never examines more (or other) nodes than the $\alpha\beta$-algorithm, but usually less [22]. As 'best-first-search' implies, the algorithm searches the best branch first, therefore it has to keep node information. SSS* uses a large data structure, an OPEN-list with descriptors of the active nodes. The algorithm is divided into two phases. Phase one spans a subtree containing all direct successors of MAX nodes but only one successor to every MIN node (top down MIN strategy). Phase two is called solution phase, it's a bottom up search for the best MAX strategy.
A descriptor (n, s, h) consists of:

- a node identifier n
- status $s \in \{LIVE, SOLVED\}$
- a merit h

LIVE: n is still unexpanded and h is an upper bound on the true minimax value.
SOLVED: h is the true minimax value.
Descriptors are sorted in decreasing order of their merit in OPEN.

The search starts with only the root node descriptor $(root, LIVE, \infty)$ and ends if the descriptor $(root, SOLVED, h)$ appears on the top of OPEN.

SSS* is about five to ten times slower than $\alpha\beta$, this makes it impractical for real game playing programs, even if it examines only a subset of nodes. At the 8th Advances in Computer Chess conference 1996, SSS* was finally declared 'dead'.

3.2.3 Heuristics

Since game tree search can't be done completely, it has become an error minimization problem. First some search extensions are described, which strive to explore potential good moves deeper. Afterwards we take a look on some pruning techniques, which try to prune off bad-looking moves to avoid searching redundant branches.

Iterative-Deepening Search (IDS). Heuristic search to a fixed depth, presents a problem with real time game-playing programs. They require a decision during an adequate time. If it exceeds the time-limit without returning, the program is unable to make a move. As mentioned before IDS repeatedly calls a depth-limited search with increasing depth d. Thus by applying IDS to the root node, a game-playing program is able to respond once the search returns a result. But it can search as deeply as possible until a time limit is reached.

Quiescence Search. Searching to only fixed depth without looking forward may cause the so called horizon effect. Assuming you stop on depth s with a node n and $eval(n) = 15$ but at depth $s + 5$ with node t the move will turn out to $eval(t) = -10$. Human players usually have enough intuition to decide whether to abandon a bad-looking move, or search a promising move to a greater depth. Quiescence search tries to imitate this intuition. It searches 'interesting' positions to a greater depth than 'quiet' ones to make sure there are no hidden traps.

Move Ordering. As already seen, a good move ordering is essential to produce more cut offs using $\alpha\beta$ and NegaScout. A slightly better move ordering can improve search performance by 50% up to 100% in a gameplaying program. Move ordering heuristics allow modern chess programs to search game-trees which have only 20% - 30% more nodes than the minimal $\alpha\beta$ game tree [10]. According to that, move ordering heuristics are usually used in practical implementations.

Dynamic Move-Ordering.

Dynamic move ordering heuristics collect information about moves during search. This information is used to order subsequent moves, e.g. killer and history heuristic. The killer heuristic is described below, for more information see [5, 28].

Static Move-Ordering.

Static move ordering heuristics do not depend on information of previous

searches, but there are several domain-independent techniques that work well in games such as MVV/LVA (Most Valuable Victim - Least Valuable Aggressor/Attacker), SEE (Static Exchange Evaluation) and SOMA.

Killer Heuristic. A move is called a killer, if it produces a cut off. Killers are assumed to cause a cut off again, in a different situation. Those moves are considered first, if they are excepted by the rules.

Aspiration search. $\alpha\beta$ or NegaScout starts with a $(-\infty, \infty)$ search window. The idea is to reduce the initial window, if there is a clue in which range the minimax window will fall. This may speed up the search, because more cuts will occur at the beginning. If the range is too tight the tree must be re-searched. Aspiration search is about 15% faster than the original $\alpha\beta$, if there exists a good range.

Transposition Table. In some games it is possible to reach the same position several times, those positions are called transpositions. To avoid analyzing the same position several times, hashtables store information from preview searches. One common technique for creating hash codes in game-playing programs is zobrist hashing. The advantages of the zobrist key are: simple to implement, incremental and fairly collision resistant.

Forward Pruning. Pruning is a name for every heuristic that completely removes certain branches of the search tree, assuming they have no influence to the search result. $\alpha\beta$ may be considered as *backward pruning*, because we found a refutation after searching.

Forward pruning always involves some risks to overlook something, because a node is discarded without searching beyond that node if it is believed that the node will not affect the final minimax value of the node. Thus forward pruning requires a compromise between accepting some risk of error and pruning more. Pruning more includes making more errors potentially, but also allows deeper searches in other branches.

As these techniques are developed for chess only, we only give a short listing:

- at expected CUT nodes: Multi-Cut, ProbCut, Null Move Pruning, which is described below
- at expected ALL nodes: Razoring, Sibling Prediction Pruning, ProbCut

Null Move Pruning. Null Move pruning assumes that if a player can make two movements in a row, even if passing is illegal, it is always an outstanding improvement. So the heuristic makes a null move, as if the opponent passes, by simply changing the turn to move to the opponent and performs a reduced-depth search. If the reduced-depth search returns a score greater than β, then the node is likely to be a strong position. Otherwise it is pruned.

4 Chess Reduction

In this chapter we want to give an overview which search strategies and heuristics used in Artificial Intelligence are comparable with ENUM.

One main difference between game search and searching for the shortest vector (SVP) is the formulation of the goal state. In game search we can give an exact goal state formulation, hence we know when to abort. If the algorithm is complete and optimal we find the highest-quality solution. For example we look at the 8-puzzle again. We have an exact goal state formulation: for each inner box one fixed position - so if we found it (with IDA^*) we can stop and the problem is solved. In contrast, we cannot formulate a specific goal state for the SVP. The only goal formulation we can make is: find the shortest one, without knowing the value of it. Transfered to the 8-puzzle the goal formulation would be called: Find the "best" status which is d steps ahead, where "best" means nearest to the actual goal state which is $> d$ steps away. Hence, ENUM cannot terminate if one good solution is found, it has to check all other solutions until one is left over in the end, which is the optimal solution.

According to this aspect, we will discuss which strategies can be adapted efficiently.

4.1 Adapting Search Strategies

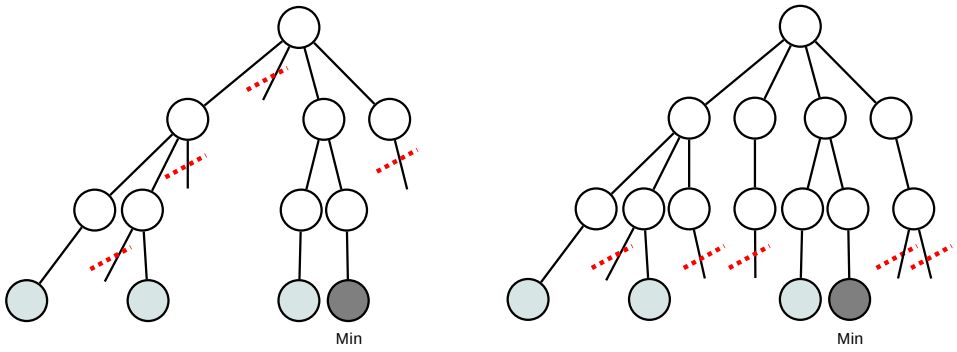
ENUM uses a DFS search, first of all we want to discuss which search strategies introduced in Chapter 3.1 could be adapted to advance ENUM.

4.1.1 Uninformed Search

The drawbacks of DFS such as getting stuck in a wrong path do not effect ENUM. Since the search tree is limited to the depth d with an input basis $\mathbf{B} \in \mathbb{Z}^{n \times d}$

ENUM performs already a DLS search. Since the depth d is known IDS would be wasteful.

BFS would not be efficient because the space complexity is in $O(b^d)$. Another disadvantage adapting BFS to ENUM would be minor cut offs. A cut off only occurs if a complete path with length d has already been discovered, which happens not until the last round is reached with depth d, as the Figure 4.1 shows.



(a) DFS (b) BSF

Figure 4.1: Possible cut offs using DSF and BSF

The same reasons as for BFS count against UCS, even if it would be better to perform the nodes with the lowest cost first. UCS uses a queue Q to store all successor nodes. Each round the node with the smallest path cost so far is examined and all successors are stored in Q. Transfered to ENUM the node with the smallest $\tilde{c}_t$ would be discovered first, but Q would increase enormously for each depth.

Unfortunately, it is not easy to use a bidirectional search for the SVP like in some problems in artificial intelligence. If we adapt the bidirectional search the algorithm will search from both ends $t_1 = d$ and $t_2 = 1$ simultaneously until they reach $d/2$. But even if both searches use DFS the branching factor will be too large, because the original starting bound $A = \left\|\widehat{\mathbf{b}}_\mathbf{1}\right\|^2$ is too large. Therefore we need two different bounds for t_1 and t_2, since the input basis is reduced with LLL or BKZ and thus sorted ascending by length. However, even if we get a result from both searches and complete $\tilde{\mathbf{u}}$ to get A, we have to search with ENUM again. It is questionable if the advantage of searching to depth $d/2$ still holds if we have to do a re-search even though we have a good bound A from the searches before.

4.1.2 Informed Search

The only heuristic known so far to decide whether a path is promising or not is the Gaussian volume heuristic explained in Section 2.2.

Let $\mathbf{b} = \sum_{i=t}^{n} \tilde{u}_i \mathbf{b_i}$. The heuristic estimates

$$\beta_t := E[\ |\{(\tilde{u}_1, \ldots, \tilde{u}_{t-1})\} \in \mathbb{Z}^{t-1} : c_j(\tilde{u}_1, \ldots, \tilde{u}_n) < A\}|\ ] = \frac{\mathrm{vol}(\mathcal{B}_{t-1}(\zeta_t, \rho_t))}{\det \bar{\mathcal{L}}},$$
$$\text{with} \quad \zeta_t = \mathbf{b} - \pi_t(\mathbf{b}) \quad \text{and} \quad \rho_t^2 = (A - \|\pi_t(\mathbf{b})\|^2).$$

Given $(\tilde{u}_t, \ldots, \tilde{u}_d) \in \mathbb{Z}^{d-t}$, β_t denotes the expected number of vectors $(\tilde{u}_1, \ldots, \tilde{u}_{t-1}) \in \mathbb{Z}^{t-1}$ satisfying $c_j(\tilde{u}_1, \ldots, \tilde{u}_n) < A$.

Even though β_t depends on the steps already made as well as on the steps to the goal, it only estimates the path from a node at stage t to the goal:

$$\underbrace{(\tilde{u}_t, \ldots, \tilde{u}_d)}_{\text{nodes already discovered}} \quad + \quad \underbrace{(\tilde{u}_1, \ldots, \tilde{u}_{t-1})}_{\text{steps till the “goal“}}.$$

If β_t is “big” the path seems promising. In Chapter 5 we will concretise what “big” stands for.

Transfered to informed search strategies the heuristic function $h(n) \mathrel{\widehat{=}} \beta_t$, so we may adapt the greedy search. As explained in Section 3.1 the greedy search algorithm uses $h(n)$ to select the node to be expanded next. Therefore a queue Q is used. Starting with $Q \in \{startNode\}$ each round the node n with the smallest $h(n)$ is examined and all successor nodes are stored in Q. The algorithm ends if the *goalState* is taken out from Q. Greedy search also uses a closed list L to check for cycles.

Transfered to the SVP we could say that in each round the node is selected with the highest β_t. In contrast of the problems in artificial intelligence the algorithm for the SVP has to search until Q is empty. Unfortunately this may have the same drawbacks as UCS, since the branching factor can be extremely high. ENUM would suggest all successors $\tilde{u}_{t-1}$ with $\tilde{u}_{t-1} < \pm\sqrt{\frac{A-\tilde{c}_t}{\mathbf{c_{j-1}}}} - \mathbf{y_{t-1}}$, as shown in Figure 4.2 for the first two rounds. This may be enormous for depth $t < d/2$, since $A - \tilde{c}_t$ will be too large at this stages because we do not have a good bound A at the beginning.

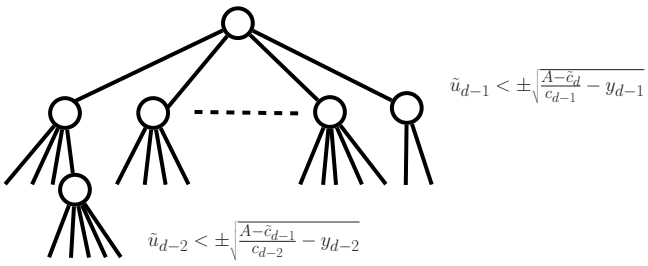


Figure 4.2: Possible number of successors of ENUM using best-first search algorithms

So if we already have a good bound A the idea of the greedy search - to sort nodes according to $h(n)$ and perform the promising ones first - can be adapted.

SCHNORR suggests in [24] a new version of ENUM called New ENUM which follows the ideas of the greedy search. We will go into detail in Chapter 5 and also show some experiments comparing ENUM and New ENUM.

Finally, we want to discuss if A^* could be adapted to solve the SVP. As mentioned in Section 3.1, A^* is optimally efficient, meaning it is guaranteed that there is no other optimal algorithm expanding fewer nodes than A^*. Transfered to the SVP it would mean that the shortest vector would be found earlier as the greedy search does, causing more cut offs for the rest of the search. But how can we get $f(n) = g(n) + h(n)$ for the SVP? The problem of combining $g(n) \mathrel{\widehat{=}} \tilde{c}_t$ and $h(n) \mathrel{\widehat{=}} \beta_t$ is that there are two different measures. Again looking at the 8-puzzle: $g(n)$ = applied moves so far and $h(n)$ = sum of the distances of the inner boxes from the goal. Both functions measure the number of moves. How to combine $\tilde{c}_t$ (path cost) and β_t (expected number of vectors) is one of the open questions.

4.2 Comparing with Chess

As already mentioned, the search tree in chess is too huge to be discovered completely in the short period of time in which a move has to be made. Therefore the search is limited to a fixed depth and thus more comparable to ENUM, insofar as both have to search to a fixed depth and return the best solution over all solutions found.

4.2.1 Algorithms

As we saw in Chapter 3.2 $\alpha\beta$ is the most popular algorithm to play chess. Besides the fact of two-players $\alpha\beta$ and ENUM are mostly alike. Both traverse the search tree in depth first order and cut off branches which return an inferior value. Unfortunately the improvement of Negascout cannot be adapted, since it relies on the fact of different bounds for each player.

The same holds for the classification of nodes. In a search tree of ENUM we cannot distinguish between nodes which can be cut off and nodes which have to be searched through anyway. Except the root, all nodes can be cut off if they return an inferior value.

The last algorithm described in 3.2 named SSS* which follows the same principle as A* transfered to two-player games. SSS* cannot be used in chess efficiently. However, we have given a brief introduction how to adapt greedy search above, which will be concretised in the following chapter.

Since $\alpha\beta$ follows the same principle as ENUM, we will examine the heuristics $\alpha\beta$ uses for chess and how far the transferability is given to ENUM, if it is not already used there.

4.2.2 Heuristics

One of the most important heuristics is the order of moves. In game playing there exist two different types: dynamic and static move-ordering. ENUM already uses a dynamic move-ordering, namely the zig-zag-path. The next possible value for $\tilde{u}_t$ depends on the value of the previous round. The preprocessing of LLL or BKZ can be seen as a static move-ordering, since the input basis is reduced and ordered.

Another important heuristic uses transposition tables to detect cycles. Fortunately, ENUM does not need to store all positions to avoid analyzing the same position twice. If we claim $\tilde{u}_i > 0$ for the largest i with $\tilde{u}_i \neq 0$ all possible redundancies are excluded.

Quienscene search cannot be adapted directly, since ENUM has to search to depth d for all nodes, not only a fixed depth $\ll d$. The idea of quienscene search is to search promising branches deeper, which we try to adapt by combining greedy

search, DFS and quienscene search, as shown in Figure 4.3. We perform a DFS to a depth $1 < m < d$, if DFS reaches a level $\geq m$ nodes are distinguished between good and bad as described above. Bad nodes are stored in queue. Good nodes are performed directly using DFS, if successors turn out to be bad they are stored in queue as well. If DFS finishes, the queue is performed using greedy search until all nodes are examined.

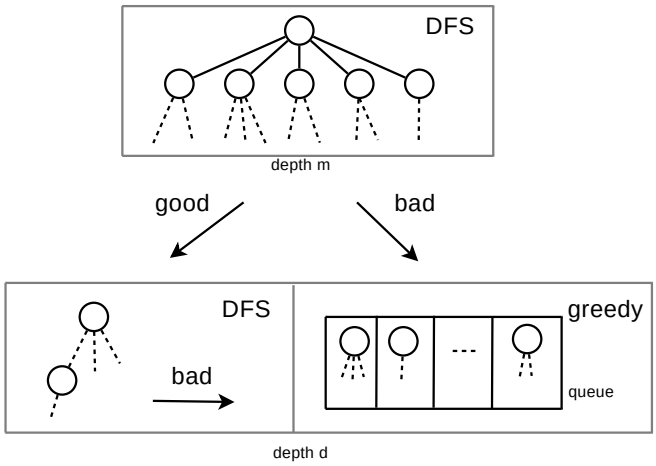


Figure 4.3: Combining DFS and A* with the idea of quienscene search

Aspiration search, which reduces the initial search window, is one of the heuristics which ENUM already uses. ENUM starts with $A = \left\|\widehat{\mathbf{b}_\mathbf{i}}\right\|^2$. SCHNORR suggests in [24] an even better bound: $A := \frac{d}{4}(\det(\mathbf{B}^\mathsf{T}\mathbf{B}))^{2/d}$. The new $A > \lambda_1^2$ holds for $d \geq 10$ as $\gamma_d < \frac{d}{4}$ for $d \geq 10$.

In game playing especially in chess many techniques to prune the search tree exist. As mentioned before, $\alpha\beta$ and ENUM can be seen as backward pruning. Forward pruning works quite differently, it prunes a branch without calculating the exact outcome, if it is believed that the node is unlikely to affect the result. These methods depend on the defined problem. In Section 3.1 we briefly described some common techniques which are used in chess, but there are not portable one-to-one for other problem classes like the SVP.

5 New ENUM

5.1 Overview

In [24] Schnorr introduced a new version of ENUM called New ENUM. New ENUM uses the Gaussian volume heuristic to categorise "good" and "bad" nodes. Good ones are performed first to set a good bound, whereas bad ones are stored in a list L and performed afterwards, because they are expected to be mostly cut off.

5.1.1 Node-Evaluation

To show how New ENUM rates a node we shortly recall ENUM and the Gaussian volume heuristic. Standing on stage t with chosen $(\tilde{u}_t, \ldots, \tilde{u}_d) \in \mathbb{Z}^{d-t+1} \setminus \{0\}$ ENUM searches for $(\tilde{u}_1, \ldots, \tilde{u}_{t-1})$ as to satisfy $\tilde{c}(\tilde{u}_1, \ldots, \tilde{u}_d) < A$. More precisely, given a vector $\mathbf{b} = \sum_{i=t}^{n} \tilde{u}_i \mathbf{b_i}$ we want to add $\bar{\mathbf{b}} = \sum_{i=1}^{t-1} \tilde{u}_i \mathbf{b_i} \in \bar{\mathcal{L}}$ as to satisfy $\left\|\mathbf{b} + \bar{\mathbf{b}}\right\|^2 < A$. As already mentioned in Section 2.2 the Gaussian volume heuristic estimates $|\mathcal{B}_{t-1}(\zeta_t, \rho_t) \cap \bar{\mathcal{L}}|$ for $t > 1$ to $\beta_t := \frac{\mathrm{vol}(\mathcal{B}_{t-1}(\zeta_t, \rho_t))}{\det \bar{\mathcal{L}}}$, with $\zeta_t = \mathbf{b} - \pi_t(\mathbf{b})$ and $\rho_t^2 = (A - \|\pi_t(\mathbf{b})\|^2)$. Gauss-ENUM uses the success rate β_t as a pruning technique, since all stages with a small success rate ($\beta_t < 2^{-p}$) are cut off.

New ENUM uses β_t differently, namely to rate nodes. All stages with $\beta_t \geq 2^{-r}$ are performed first, whereas stages $\beta_t < 2^{-r}$ are collected in a list L_r and performed afterwards in increasing order of t and for fixed t in order of decreasing β_t. Parameter r describes the number of rounds, which starts with $r = 1$ and increases ($r = r + 1$) each round.

This classification holds, because very short lattice vectors $\mathbf{b} := \sum_{i=1}^{d} \tilde{u}_i \mathbf{b_i}$ have large values β_t at the stages $(\tilde{u}_t, \ldots, \tilde{u}_d)$. On average $\|\pi_t(\mathbf{b})\|^2 \approx \frac{d-t+1}{d} \|\mathbf{b}\|^2$, so if $\mathbf{b}$ is short so are the $\pi_t(\mathbf{b})$. Hence $\rho_t^2 = A - \|\pi_t(\mathbf{b})\|^2$ is large and so is β_t.

5.1.2 Algorithm

New ENUM can be divided into two different parts. First a modified version of ENUM is performed: Start with $r = 1$, perform ENUM with only one modification: store the node in L_r if $\tilde{c}_t < A$ and $\beta_t < 2^{-r}$.

The second part performs all nodes from the current list L_r with level $r + 1$ and collects delayed stages with $\beta_t < 2^{-r-1}$ in L_{r+1}. New ENUM loops part two until $L_{r+1} = \emptyset$, meaning until all nodes are analysed.

Since the first part of New ENUM is nearly the same as ENUM we will concentrate on the second part, which is not seriously discussed in [24] so we will give a detailed description.

Suppose New ENUM finishes the first phase and starts the second one with the list L_1 at least consisting of one node. The list is sorted in decreasing order of t and for fixed t in increasing order of β_t. According to that, the first node n_1 is taken out.

One of the following three cases may occur:

- If $\tilde{c}_t \geq A$ the node will not lead to a new minimum as its value is already greater than the minimum, a new node will be taken.
- If $\tilde{c}_t < A$ and $\beta_t \geq 2^{-r-1}$ the stage increases ($t = t - 1$),
- otherwise the node is stored and a new node is taken from the list.

We take a look at the second possibility. Thus, we can specify which variables need to be stored if $\tilde{c}_t \leq A$ and $\beta_t < 2^{-r-1}$. Schnorr makes no suggestions in which order the successors are taken into consideration. We outline two different approaches:

If the algorithm steps down, all possible successors are appended to L_{r+1}. This has the advantage that all successors are considered at once and lesser variables have to be stored for a node, but it blows up the list size needlessly.

The second possibility only considers the best successor according to the zig-zag-path. The next successor is calculated if the prior is finished. It has the drawback of storing more informations for each node but keeps the size of the list as small as possible.

We will apply the second one. Even if we append all successors at once, New ENUM considers the best successors according to the zig-zag-path first. Most beneficial is the size of L, which is one of the critical factors as we will see later.

Node is finished. To understand how all successors are taken into consideration, we make a short example: Assume a node n is taken out of L_r which stands on stage t with $(\tilde{u}_t, \ldots, \tilde{u}_d)$. Afterwards the algorithm has made a ($a < d-t$) steps down - standing on stage $t-a$ with $(\tilde{u}_a, \ldots, \tilde{u}_d)$. If $c_a > A$, $\tilde{c}_t < A$ and $\beta_t < 2^{-r}$ or $t = 1$ the algorithm has to step up. Each time the algorithm steps up the next possible successor $\tilde{u}_i$ is calculated. The algorithm may go down again if the $\tilde{c}$ is still smaller than A. To take all possible successors from n into consideration the algorithm has to step up until stage t with $\tilde{c}_t > A$ is reached.

Which variables need to be stored. The actual cost $\tilde{c}_t$ is calculated from $\tilde{c}_{t+1}, y_t, \tilde{u}_t$ and c_t, where y_t depends on all prior $\tilde{u}$ and $\tilde{u}_t$ depends on v_t, Δ_t and δ_t. According to that, a node has to contain the following variables:

$$t, \tilde{c}_{t+1}, (\tilde{u}_t, \ldots, \tilde{u}_d), \Delta_t, \delta_t, v_t, \text{ ($\tilde{c}_t$ and s to protect duplicate computations)}$$

The algorithm of the second part of New ENUM is shown in Algorithm 3.

Algorithm 3: New ENUM List-algorithm

Input: List L_r, $\mu \in \mathbb{R}^{d\times d}$, $c_1, \ldots, c_d$, current minimum A
Result: the minimal place $(u_1, \ldots, u_d) \in \mathbb{Z}^d \setminus \{0\}$ and the minimum A

1 **if** L_r *not empty* **then** sort L_r, $n \leftarrow L_r[0]$, $t_{max} = n(t)$, l=size L **else** return
2 **while** *true* **do**
3 $\quad \tilde{c}_t := \tilde{c}_{t+1} + (y_t + \tilde{u}_t)^2 \cdot c_t$
4 $\quad$ **if** $\tilde{c}_t < A$ **then**
5 $\quad\quad$ **if** $t == 0$ **then** ▷ *minimum found*
6 $\quad\quad\quad A := \tilde{c}_j$, $\mathbf{u} := \tilde{\mathbf{u}}$
7 $\quad\quad$ **else**
8 $\quad\quad\quad$ calculate β_t
9 $\quad\quad\quad$ **if** $\beta_t \geq 2^{-r-1}$ *or* $l == 0$ *or* $t == 1$ **then** ▷ *good node*
10 $\quad\quad\quad\quad t := t - 1$, $y_t := \sum_{i=t+1}^{s} \tilde{u}_i \mu_{it}$, $\tilde{u}_t := \mathbf{v_t} := \lceil -\mathbf{y_t} \rfloor$, $\Delta_t := 0$
11 $\quad\quad\quad\quad$ **if** $\tilde{u}_t > -y_t$ **then** $\delta_t := -1$ **else** $\delta_t := 1$
12 $\quad\quad\quad\quad$ continue ▷ *step up*
13 $\quad\quad\quad$ **else** ▷ *bad node*
14 $\quad\quad\quad\quad L_{r+1}.push\_back(n)$, $l++$
15 $\quad\quad\quad$ **end**
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad$ **if** $l == 0$ *and* $t_{max} == t$ **then** break; ▷ *all possible nodes discoverd*
19 $\quad$ **if** $t_{max} == t$ **then**
20 $\quad\quad$ **if** L_r *is empty* **then** $r + 1$, sort L_r, $L_{r+1} = \emptyset$ ▷ *r finished*
21 $\quad\quad l - 1$, $n \leftarrow L_r[0]$ ▷ *take a new node*
22 $\quad$ **else** ▷ *update*
23 $\quad\quad t := t + 1$, $s := \max(s, t)$
24 $\quad\quad$ **if** $t < s$ **then** $\Delta_t := -\Delta_t$
25 $\quad\quad$ **if** $\Delta_t \cdot \delta_t \geq 0$ **then** $\Delta_t := \Delta_t + \delta_t$
26 $\quad\quad \tilde{u}_t := v_t + \Delta_t$
27 $\quad$ **end**
28 **end**

5.2 Implementation and Results

5.2.1 Overview

We implemented the New ENUM algorithm using C++. All tests were run on Linux (openSuse 11.1) using gcc version 4.3.2. with a 32-bit Intel(R) Core 2 CPU (2.00 GHz) and 2.0 GB RAM. We used an already existing matrices class representation for floating point and integer matrices. All floating point variables such as $\tilde{c}$, μ are stored as `double` (precision: 53 bits). Our implementation also provides multiple-precision floating-point computations using the MPFR Library [1], but `double` seems sufficient if the input basis is at least LLL-reduced. We used LLL-reduced random lattices with $bitsize = 10 * dimension$ for all tests.

5.2.2 Using One List

We first tried to reduce the two lists L_r and L_{r+1} into one: the next node is taken from the front whereas new nodes are stored at the end. We used a counter `listSize` to recognize when the current round r is finished, as shown in Figure 5.1. Therefore we used `std::deque<Node*>`,which is a double-ended queue. Tests showed that `std::deque<Node*>` is about 99% faster than `std::vector<Node*>` and about 20% faster than `std::list<Node*>`. This results from the only removal/ adding of elements from the beginning or the end.

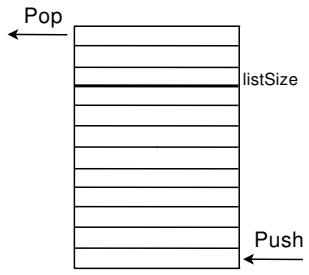


Figure 5.1: New ENUM using only one list

They all provide Random Access Iterators, as we used

```
void sort ( RandomAccessIterator first, RandomAccessIterator last );
```

from `STL Algorithms` to sort our list.

Our first tests over dimension 42 failed, as the program terminated after throwing an instance of `std::bad_alloc`.
Since the elements in a deque are stored in several smaller chunks to provide massive reallocations they are a little more complex internally. We try to use

`std::vector<Node*>` instead. Vectors are very similar to plain arrays. They grow by reallocating all of their elements in an unique block when their capacity is exhausted. Because of that we can only provide access at the ending of a vector.

5.2.3 Using Two Lists

Instead of using one list we use two different lists L_1 and L_2. In a round r we use L_1 to pop nodes from the end and L_2 to store nodes at the end. If a round r ends meaning L_1 is empty we sort L_2 and switch parts: L_1 is filled and L_2 emptied. Since we only perform operations at the end of both lists we use `std::vector<Node*>`.
As illustrated in Figure 5.2 New ENUM uses two pointers `prtPush` and `ptrPop` to adress the respective list $L_{1/2}$.

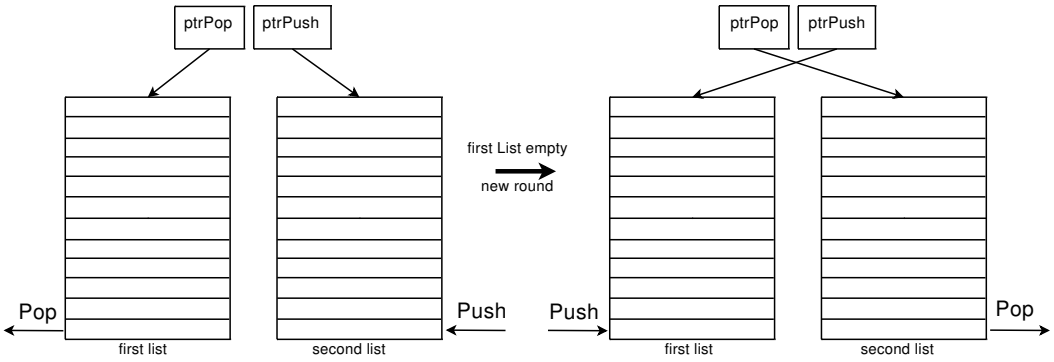


Figure 5.2: New ENUM using two different lists

5.2.4 NewEnum Class

We are going to give a brief description of the New ENUM class. As already mentioned we use two different pointers `ptrPop` and `ptrPush` which point to the respective list `list1` or `list2`. `FacBeta` is the edge between good and bad nodes, each round we decrease `FacBeta=FacBeta/2`. The current minimum is stored in `A` and `listSize` contains the sum of the size from both lists.

Listing 5.1: private class members

```
const Matrix<double>& mu;
const double* c;
```

```
    const int cols;
    double FacBeta;
    unsigned int listSize;
    double A;
    std::vector <Node*> list1;
    std::vector <Node*> list2;
    std::vector <Node*>* ptrPop;
    std::vector <Node*>* ptrPush;
```

We used a function called `getNext` listed in 5.2 which prevents deleting and creating the same node in one iteration. The function finds the next node in the list which is $> 2^{-r-1}$, if all nodes $< 2^{-r-1}$ r increases.

Listing 5.2: function to get the next possible node

```
getNext(Node* &ptrNode){

  if (ptrPop->empty()){
    FacBeta/=2;
    //switch pointer
    swap(ptrPush,ptrPop);
    std::sort(ptrPop->begin(),ptrPop->end(),&nodeSortEnd);
  }
  ptrNode = ptrPop->back();
  if(listSize>1 && ptrNode->t>1){
    //while node is too bad
    while(ptrNode->coeff_betat < FacBeta && ptrNode->t>1){
      //push_back node
      ptrPush->push_back(ptrNode);
      ptrPop->pop_back();
      ptrNode=ptrPop->back();

      if (ptrPop->empty()){
        FacBeta/=2;  // rounds r++
        //switch pointers
        swap(ptrPush,ptrPop);
        sort(ptrPop->begin(),ptrPop->end(),&nodeSortEnd);
        ptrNode=ptrPop->back();
      }
    }
  }
}
```

5.2.5 Results

Testing New ENUM against ENUM showed that New ENUM is uncompetitive to ENUM for dimensions greater than 35.

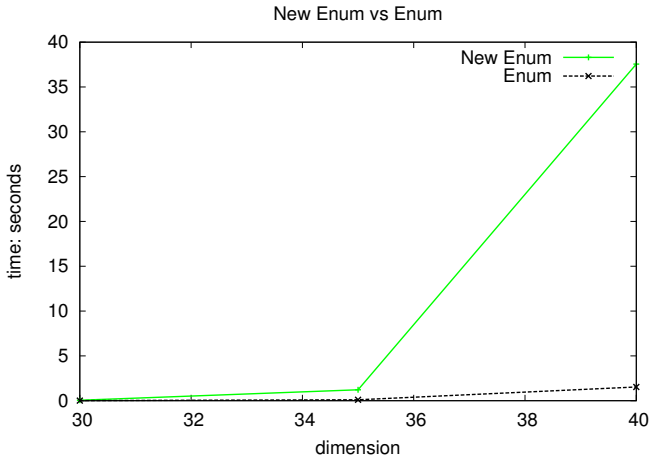


Figure 5.3: Running time of New ENUM and ENUM

In dimensions ≥ 40 New ENUM stores more than 5.000.000 nodes which reduces the speed immensely. Figure 5.4 shows the maximal storage of nodes up to dimension 40. On that account we take a look on the process of node storage in dimension 40. Figure 5.5 shows that by the time the minimum is found the size of the list is acceptable, but afterwards it grows massively.

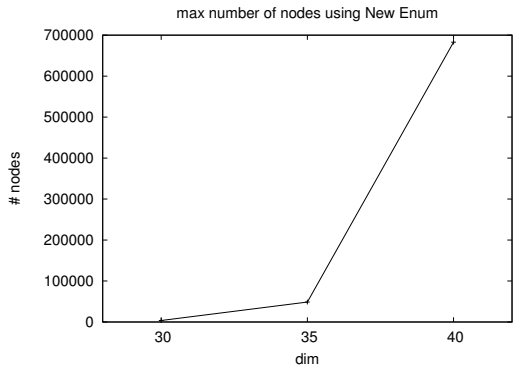


Figure 5.4: The average maximal number of nodes New ENUM stores in a list

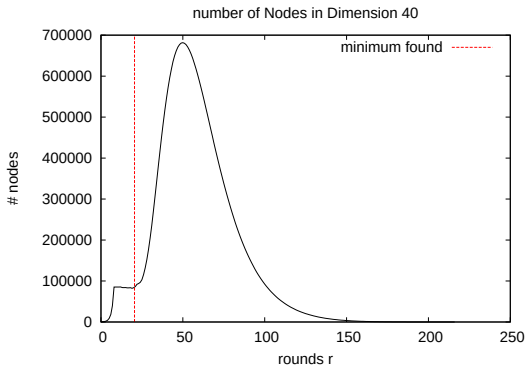


Figure 5.5: The average node storage of New ENUM in dim 40

It seems that after the minimum is found all remaining bad nodes are hold back and stored instead of performing cut offs.

5.2.6 Variations

On account of the massive boost of nodes, New ENUM can only be performed up to dimension 40, but is still incomparable with ENUM. Next we will discuss some variations to get rid of the huge storage problem.

Reduce List. As seen in Figure 5.5 the minimum in dimension 40 is found on average in round $20,5$. Our new idea is to skip storing after the minimum has been found and perform all nodes in the list. Therefore we use a function called `JustList` which performs all nodes (and successors) in the list without calculating β_t nor storing any node. According to the average round in which the minimum is found, we skip storing and perform `JustList`, the result is shown in Figure 5.6.

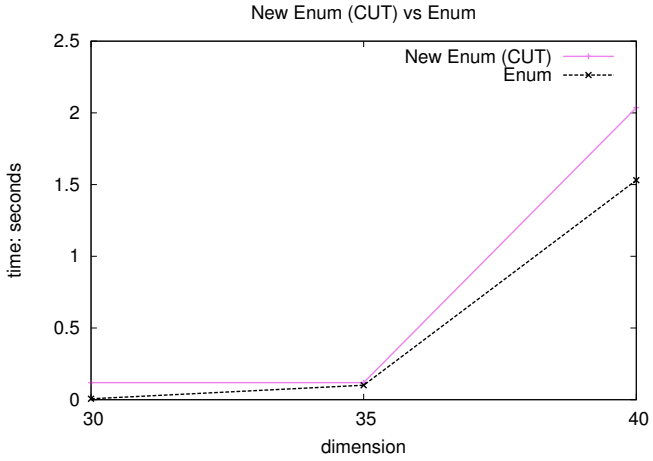


Figure 5.6: Running time of New ENUM (CUT) and ENUM

As mentioned before it is very ineffective if New ENUM needs to store more than 5.000.000 nodes, but for dimension ≥ 42 the minimum is not found within this barrier. Thus we reduce New ENUM to the minimum of storage. The first part is performed as usual: if $\beta_t < 2^{-1}$ the node is stored otherwise all possible successor nodes are discovered. In the second part we only perform `JustList`. Figure 5.2.6 shows that for dimension ≥ 44 New ENUM `JustList`is faster than ENUM.

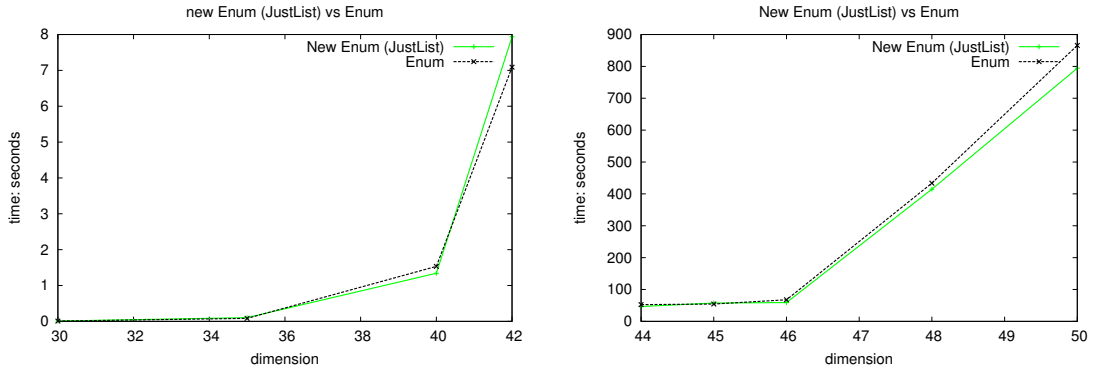


Figure 5.7: Running time of ENUM and New ENUM (JustList)

Pruning branches. Another approach to get rid of storing too many nodes is pruning stages with the original Gaussian volume heuristi c. We set a variable `MIN_BETA`$= 2^{-p}$, where p denotes the pruning factor. Hörner suggested $p = 13$, which holds for all our tests. Figure 5.8 shows that New ENUM (`JustList`) using the Gauss-pruning makes a small speed-up, except for dimensions 45 and 52.

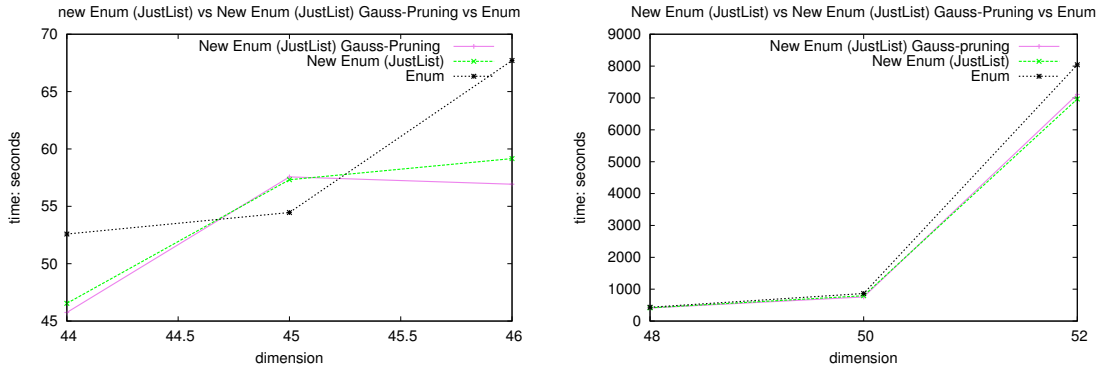


Figure 5.8: Running time of New ENUM (JustList) with pruning (Gauss Heuristic: p=13)

Quienscene New ENUM. As suggested in Chapter 4 we try to set a barrier b for storage, meaning only nodes with $t < b$ can be stored. We tested bases in dimensions $44 - 50$ using $b = 10$ and $b = 15$. In average the time changes for the worse.

We also tried different barriers for node storages or stopping at various stages r, but these techniques could not hold for all bases at one dimension and none of them could reach the time of `JustList`(with pruning). The heuristic β does not seem to compensate the drawback of the storage of nodes.

6 Conclusion

6.1 Chess Reduction

In this thesis we tried to adapt search strategies used in the field of artificial intelligence to the ENUM algorithm as well as common techniques used in chess.

We showed that the goal formulation is one of the major differences, which causes a low disadvantage. We pointed out that adapting search strategies like BFS and UCS is not effective. An adaption of bidirectional search sounded promising insofar as depth d would be reduced to $d/2$. However, all external circumstances seemed to ruin the advantage, so we abandon this option.

Best first search seems to be more promising. Since the only known heuristic for ENUM estimates the path cost from a node to the goal we adapted the greedy search. In order to reduce the problem of high branching factors we made some improvements concerning the successor nodes generation. We pointed out that A^* is the most auspicious search strategy, since it takes the total path cost into consideration. Combining the path cost already made with an adequate heuristic would be interesting for further research.

We worked out that most transferable techniques used in chess are already used by ENUM. Nevertheless, the number of pruning techniques is of decisive importance. In game playing especially in chess various forward and backward pruning techniques exist which cause an enormous speed-up. Those techniques compensate the drawbacks of the used search strategies. Up to date, the only known heuristic which prunes the search tree of ENUM is the Gaussian volume heuristic.

6.2 New ENUM

In the last part of our thesis we implemented the New ENUM algorithm. We combined ideas of greedy search as well as ideas of Schnorr in [24]. Therefore

we rated notes into "good" and "bad" using the Gaussian heuristic. "Good" ones were performed directly, whereas "bad" ones were kept in a list. Each round we decreased the limit between "good" and "bad", which turned out to be disadvantageous. New ENUM could only be performed up to dimension ≥ 40. After finding the optimal solution the storage of nodes rose massively. Therefore we tried another approach (`JustList`): we performed only one round of node classification, afterwards we reduced the remaining list in decreasing order of the heuristic. In average New ENUM `(JustList)` is faster than ENUM. We also implemented New ENUM `(JustList)` using the Gaussian volume heuristic as a pruning technique, which also causes a small speed-up for most our tests.

We observed that the discrepancy of the running time (between ENUM and New ENUM `(JustList)`) for specific input basis can be quite high. This phenomenon might be of interest for further work. Discovering the reason for this phenomenon would probably simplify the work on new heuristics as well as an improvement of both algorithms. Given an input basis it would be even possible to decide which algorithm is more sufficient.

Another approach for further research might be the detection of an optimal storage container, because this has turned out to be one of the crucial factors affecting the running time.

7 Bibliography

[1] Multiple-precision floating-point computations with correct rounding (MPFR) library for C. `http://www.mpfr.org/`.

[2] Miklós Ajtai. The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the Annual Symposium on the Theory of Computing (STOC) 1998*, pages 10–19. ACM Press, 1998.

[3] Miklós Ajtai. Generating hard instances of the short basis problem. In *International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, pages 1–9. Springer-Verlag, 1999.

[4] Miklos Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *ACM Symposium on Theory of Computing*, pages 601–610, 2001.

[5] Selim G. Akl and Monroe M. Newborn. The principal continuation and the killer heuristic. In *Proceedings of the 1977 annual conference*, pages 466–473, New York, NY, USA, 1977. ACM Press.

[6] Werner Backes. Berechnung kürzester Gittervektoren. Diploma thesis, Max-Planck-Institut für Informatik in Saarbrücken, 1998.

[7] U. Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):463–471, 1985.

[8] Aviezri. S Fraenkel and David Lichtenstein. Computing a perfect strategy for n*n chess requires time exponential. *Springer LNCS 115*, pages 278–293, 1981.

[9] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *Advances in Cryptology — Eurocrypt 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer-Verlag, 2008.

[10] Ernst A. Heinz. *Scalable Search in Computer Chess*. Vieweg Verlag, 2000.

[11] Horst Helmut Hörner. Verbesserte Gitterbasenreduktion; getestet am Chor-Rivest Kryptosystem und an allgemeinen Rucksack-Problemen. Master's thesis, Frankfurt am Main, 1994.

[12] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Annual Symposium on the Theory of Computing (STOC) 1983*, pages 193–206, New York, NY, USA, 1983. ACM Press.

[13] Donald Knuth and Ronald Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[14] Tony A. Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, 1985.

[15] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *Society for Industrial and Applied Mathematics*, 37:267–302, April 2007.

[16] Daniele Micciancio and Salil Vadhan. Statistical zero-knowledge proofs with efficient provers: Lattice problems and more. In *CRYPTO*, pages 282–298. Springer-Verlag, 2003.

[17] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the Annual Symposium on Discrete Algorithms (SODA) 2010*, 2010.

[18] Phong Q. Nguyen and Damien Stehlé. Floating-point lll revisited. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233. Springer-Verlag, 2005.

[19] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *J. of Mathematical Cryptology*, 2(2), 2008.

[20] Judea Pearl. Scout: A simple game-searching algorithm with proven optimal properties. In *AAAI*, pages 143–145, 1980.

[21] Alexander Reinefeld. An improvement of the scout tree search algorithm. *ICCA Journal*, 6:4–14, 1983.

[22] Alexander Reinefeld. A minimax algorithm faster than alpha-beta. In *University of Limburg*, pages 237–250, 1994.

[23] Claus-Peter Schnorr. Vorlesungen von Prof. Dr. C.P. Schnorr: Gitter und Kryptographie an der Johann Wolfgang Goethe-Universität Frankfurt/Main im Sommersemester 2009.

[24] Claus-Peter Schnorr. Average time fast SVP and CVP algorithms for low density lattices., 2010. `http://www.mi.informatik.uni-frankfurt.de/research/papers.html`.

[25] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994.

[26] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest cryptosystem by improved lattice reduction. In *Advances in Cryptology — Eurocrypt 1995*, volume 921 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1995.

[27] Victor Shoup. Number theory library (NTL) for C++. `http://www.shoup.net/ntl/`.

[28] Mark H. M. Winands, Erik C. D. van der Werf, H. Jaap van den Herik, and Jos W. H. M. Uiterwijk. The relative history heuristic. In *Computers and Games*, pages 262–272, 2004.