BKZ Algorithms: An Introduction

Yuncong Zhang

April 1, 2020

1 Introduction

The *Shortest Vector Problem* (SVP) is finding the shortest non-zero vector in a lattice L given the basis $(\vec{b}_1, \cdots, \vec{b}_n)$. The norm of the shortest non-zero vector in L is denoted by $\lambda_1(L)$. The approximation version of SVP is denoted by γ-SVP, which is finding a vector in L whose norm is at most $\gamma(n)\lambda_1(L)$. Obviously, $\gamma(n) \geq 1$ is necessary for a meaningful γ-SVP problem, since by definition it is impossible to find a vector with norm less than $\lambda_1(L)$. For $\gamma = 1$, the γ-SVP is the original SVP. The quality of SVP solving algorithms are measured by γ, i.e. the smallest γ such that they can solve γ-SVP.

The current algorithms to solve SVP can be categorized into three classes. One class is the blockwise algorithms, which usually optimize the basis as a whole. Given appropriate parameters, these algorithms typically solves the SVP problem in polynomial time, with the approximation factor being exponential. The blockwise algorithms starts with the famous LLL reduction, and the ones that currently perform the best are the BKZ algorithms. The original BKZ algorithm was proposed by C. P. Schnorr [SE94] in 1994. It was then improved in various aspects, and such improvements were gathered into a new version of BKZ called BKZ 2.0 [CN11]. Specifically, the improvements adopted in BKZ 2.0 include extreme pruning in enumeration, early termination, local-block preprocessing, and limiting the point searching in enumeration by Gaussian Huristic.

Progressive BKZ is a type of BKZ algorithms that implement the strategy of adaptively increasing the blocksize. However, it is challenging to find an optimal strategy to increase the blocksize so that the computational cost is minimized. BKZ 2.0 uses a progressive BKZ in the local-block preprocessing in each step, though it does not use it globally.

Blockwise algorithms such as BKZ works together with another category of SVP solving algorithm called the enumeration algorithm. The enumeration algorithm, contrary to the blockwise algorithms, is able to find the exact solution to SVP, but requires expoenential time w.r.t. the lattice dimension. The blockwise algorithms and enumeration algorithms rely heavily on each other. In fact, BKZ repeatedly invokes enumeration algorithm to solve SVP in the local block which is a lattice of small dimension. On the other hand, the efficiency of enumeration algorithms is significantly impacted by the quality of the input

basis. Therefore, the enumeration algorithms usually require the input to be preprocessed by a blockwise algorithm.

The last category of algorithms solving SVP is called *sieving*. This kind of algorithms require large amount of memory, and was considered insignificant compared to blockwise and enumeration algorithms [GNR10]. However, the sieving algorithms have been developing rapidly recently, and is currently occupying the first positions in the TU Darmstadt Lattice Challenge. However, these algorithms are not much relevant to the topic in this article, so we will not cover them in the following.

This article aims to provide an illustration of the state-of-the-art BKZ algorithms. We start with necessary preliminaries to understand BKZ in Sec. 2. Sec. 3 describes the most basic BKZ algorithm. The improvements by BKZ 2.0 is explained in Sec. 4. Sec. 5 and Sec. 6 give the progressive BKZ algorithms. Finally, Sec. 7 concludes this article.

2 Preliminaries

2.1 Gram-Schmidt Basis

Given a lattice $L = L(\vec{b}_1, \cdots, \vec{b}_n) = \left\{ \sum x_i \vec{b}_i : x_i \in \mathbb{Z} \right\}$ of dimension n, and its basis $B = (\vec{b}_1, \cdots, \vec{b}_n)$, the Gram-Schmidt reduction of the basis B is defined as $B^* = (\vec{b}_1^*, \cdots, \vec{b}_n^*)$, where $\vec{b}_i^* = \vec{b}_i - \sum_{j=1}^{i-1} \langle \vec{b}_i, \vec{b}_j^* \rangle \vec{b}_j^* / \|\vec{b}_j^*\|^2$. For simplicity, let $\mu_{ij} = \langle \vec{b}_i, \vec{b}_j^* \rangle / \|\vec{b}_j^*\|^2$ and call it the *Gram-Schmidt coefficient* of B. Similarly, $\|\vec{b}_j^*\|$ is called the *Gram-Schmidt lengths*. Notice that B^* is not necessarily a basis for lattice L, since the Gram-Schmidt coefficients are not restricted to be integers, though it is still a basis of the linear space spanned by B. The Gram-Schmidt reduced basis B^* is an orthogonal basis for the linear space $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_n)$. Moreover, each $\vec{b}_i^*$ is perpendicular to the space $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1}) = \mathsf{span}(\vec{b}_1^*, \cdots, \vec{b}_{i-1}^*)$.

The Gram-Schmidt reduction does not change the determinant of the basis $\det(L) := \det(\vec{b}_1, \cdots, \vec{b}_n)$. To see that, notice that in each step of the reduction, the difference between $\vec{b}_i^*$ and $\vec{b}_i$ is a vector $\vec{b}_i - \vec{b}_i^* \in \mathsf{span}(vb_1, \cdots, \vec{b}_{i-1})$. Since $\det(\vec{b}_1, \cdots, \vec{b}_n)$ is linear w.r.t each vector, we have

$$\det(\vec{b}_1, \cdots, \vec{b}_i, \cdots, \vec{b}_n) = \det(\vec{b}_1, \cdots, \vec{b}_i - \vec{b}_i^*, \cdots, \vec{b}_n) + \det(\vec{b}_1, \cdots, \vec{b}_i^*, \cdots, \vec{b}_n) \tag{1}$$

The first summand of (1) is 0, since $\vec{b}_i - \vec{b}_i^*$ is not linearly independent from $(\vec{b}_1, \cdots, \vec{b}_{i-1})$. Therefore, we conclude that each step of the Gram-Schmidt reduction does not affect the determinant. This conclusion applies to the entire reduction, i.e. $\det(B^*) = \det(B)$.

Since B^* is an orthogonal basis, we also have $\det(B^*) = \prod_{i=1}^n \|\vec{b}_i^*\|$. This is the volume of the n-dimensional rectangle spanned by the vectors in B^*. In fact, this is also the volume of the n-dimensional *parallelepiped*, namely a cell of the lattice, spanned by the vectors in B. Therefore, we also use the notation $\mathrm{vol}(L) := \det(L) := \det(B)$ interchangably.

Fig.1 illustrates how Gram-Schmidt orthogonalization transforms a parallelepiped into rectangle, and the fact that the transformation does not change the volume of the basis.

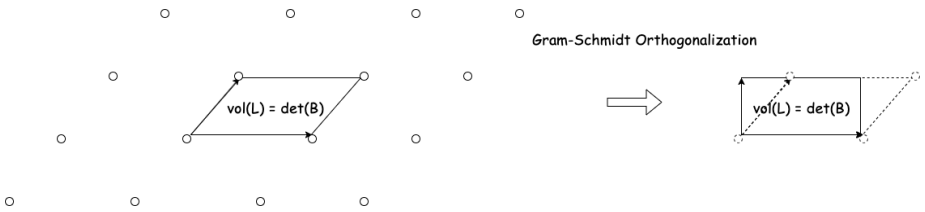

Figure 1: Gram-Schmidt orthogonalization and volume of lattice

Note that the Gram-Schmidt reduced basis B^* is no longer a basis of the original lattice. However, it is frequently used in the analysis of the efficiency of the SVP algorithms.

Geometric Series Assumption (GSA). The GSA introduced by Schnorr says that the Gram-Schmidt reduced basis vectors decreases by a constant r with the increment of i. Specifically, $\|\vec{b}_i^*\|^2/\|\vec{b}_1\|^2 = r^{i-1}$, where $r \in [3/4, 1)$ is called the GSA constant.

2.2 Projective Sublattice

We define the projection $\pi_i : \mathbb{R}^n \to \mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1})^\perp$ as in equation (2):

$$\pi_i(v) := \vec{v} - \sum_{j=1}^{i-1} \langle \vec{v}, \vec{b}_j^* \rangle \vec{b}_j^* / \|\vec{b}_j^*\| \tag{2}$$

Obviously, this is the same as the definition of one step in the Gram-Schmidt reduction. With this notion, the Gram-Schmidt reduction can be expressed as follows: for i from 1 up to n, $\vec{b}_i^* = \pi_i(\vec{b}_i)$. In fact, this projection removes the proportion of the vector inside the space $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1})$, and leaves the part that is in $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1})^\perp$. In particular, $\pi_1(\cdot)$ is the identity map, and for $i > 0$, $\pi_i(\cdot)$ projects the entire space $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1})$ to zero, including all the basis, i.e. $\vec{b}_j$ for $1 \le j < i$.

We further define the *local-block* of size $\beta = j - i + 1$ as the projective sublattice:

$$\begin{aligned} L_{[i:j]} &:= \pi_i(L(\vec{b}_i, \vec{b}_{i+1}, \cdots, \vec{b}_j)) \\ &:= L(\pi_i(\vec{b}_i), \pi_i(\vec{b}_{i+1}), \cdots, \pi_i(\vec{b}_j)) \end{aligned} \tag{3}$$

This is the lattice obtained by projecting all the vectors in $L(\vec{b}_i, \vec{b}_{i+1}, \cdots, \vec{b}_j)$ by π_i. The second equality can be easily deduced from linearity of π_i.

In fact, since π_i zerofy the entire subspace $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1})$, it is equivalent to define $L_{[i:j]}$ as $\pi_i(L(\vec{b}_1, \cdots, \vec{b}_j))$.

Fig.2 illustrates how a projection works on lattice in the simple 2-dimensional case.

For simplicity, we use the notation B_i for the local-block lattice $L_{[i:j]}$ when the blocksize β (thus the value of j) is clear in the context.

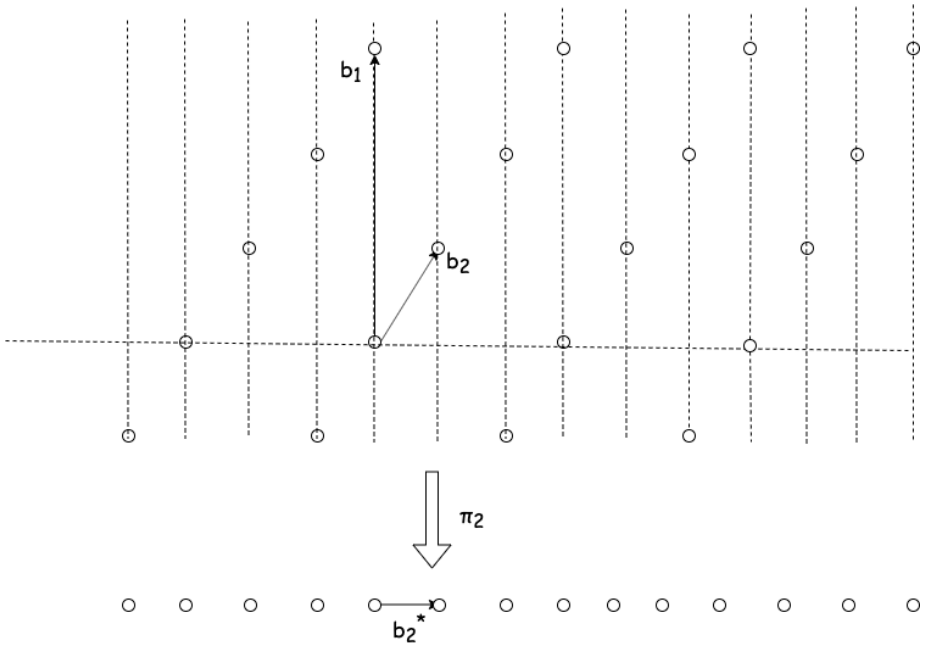

Figure 2: Projection π_2 on a lattice

2.3 Gaussian Huristic

The *Gaussian Huristic* is an estimation of the number of lattice points contained in an n-dimensional convex set S. It basically says that the number of lattice points contained in the set, denoted by $|S \cap L|$, is approximately proportional to the volume of the set. As we can image, the lattice points spread very evenly over the entire $\mathbb{R}^n$ space, it is not hard to understand this huristic. Specifically, the Gaussian Huristic says that $|S \cap L| \approx \text{vol}(S)/\text{vol}(L)$. Recall that $\text{vol}(L)$ is the volume of the parallelepiped spanned by a basis of L, i.e. a cell of the lattice. This is also understandable, since the number of cells contained in S is approximately the number of lattice points contained in S.

Fig.3 is an example where a 2-dimensional ball (i.e. a circle) contains 6 lattice points. The size of the ball is roughly 6 times of the size of a lattice cell.

The convex shape that we care most about is n-dimensional ball. Denote by $V_n(R)$ the volume of an n-dimensional ball with radius R. We have the following equation:

$$V_n(R) = R^n \cdot \frac{\pi^{n/2}}{\Gamma(n/2+1)} \tag{4}$$

where $\Gamma(x)$ is the gamma function, which is defined by $\Gamma(x) = \int_0^\infty t^{s-1} \cdot e^{-1} dt$.

The Gaussian heuristic of the shortest vector of L, i.e. $\lambda_1(L)$, is the radius of the ball with the same volume as a cell of the lattice. That is the solution of $V_n(x) = \text{vol}(L)$, which

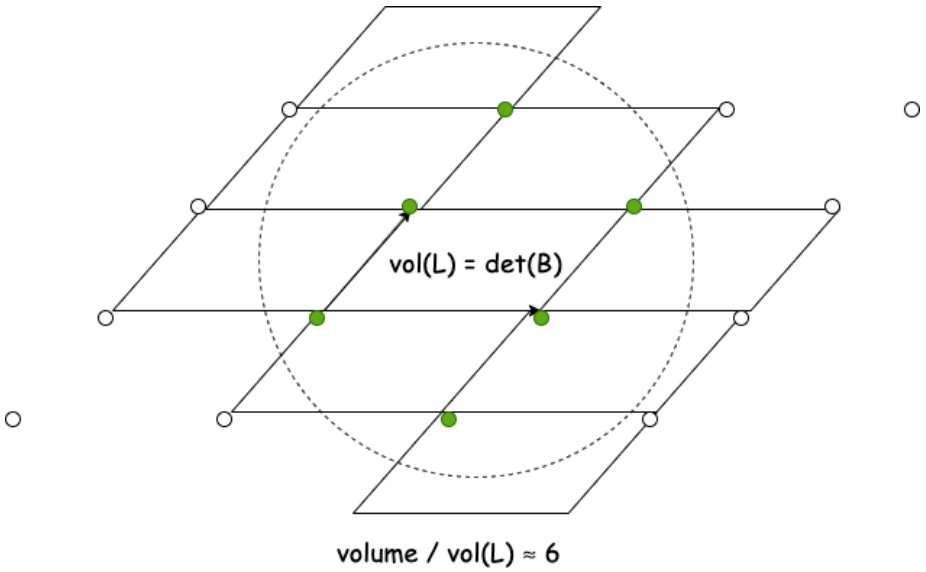

Figure 3: Gaussian Huristic for 2-dimensional Lattice

is:

$$(\mathrm{vol}(L)/V_n(1))^{1/n} = \left(\frac{\Gamma(n/2+1)\mathrm{vol}(L)}{\pi^{n/2}}\right)^{1/n} = \frac{(\Gamma(n/2+1)\mathrm{vol}(L))^{1/n}}{\pi^{1/2}} \tag{5}$$

This is denoted by $\mathrm{GH}(L)$, which is often used as a "prediction" of $\lambda_1(L)$. There are also counterexamples to this heuristic.

The Hermite constant γ_n is the supreme of $(\lambda_1(L)/\mathrm{vol}(L)^{1/n})^2$ over all lattices. By Minkowski's theorem, $\sqrt{\gamma_n} \leq 2 \cdot V_n(1)^{-1/n}$. From this we obtain that $\lambda_1(L) \leq 2\mathrm{GH}(L)$.

Gaussian Heuristic for Projective Lattices. The length $\lambda_1(B_i)$ of the shortest vector in the projective local lattice for small blocksizes β [CN11]. Recall that a projective local lattice B_i is the lattice $\pi_i(L(\vec{b}_i, \cdots, \vec{b}_{i+\beta-1}))$. Denote the ratio between $\lambda_1(B_i)$ and $\mathrm{GH}(B_i)$ for block of dimension β by τ_β, and call it the coefficient of modified Gaussian heurstic. According to the simulation provided in [CN11], τ_β decreases with β, and for $\beta > 50$, τ_β is very close to 1, i.e. $\lambda_1(B_i)$ is close to $\mathrm{GH}(B_i)$ for large values of β. Therefore, it is reasonable to estimate $\lambda_1(B_i) = \tau_i \mathrm{GH}(B_i)$ for $\beta \leq 50$, and $\lambda_1(B_i) = \mathrm{GH}(B_i)$ for $\beta > 50$.

2.4 Enumeration Algorithm

The enumeration algorithm [GNR10] searches for the shortest vector in a lattice. Given a lattice basis $B = (\vec{b}_1, \cdots, \vec{b}_n)$, the shortest vector $\vec{v}^*$, like any vectors in the lattice, can be

expressed as a combination of the basis with integer coefficients. Let

$$\vec{v}^* = a_1\vec{b}_1 + a_2\vec{b}_2 + \cdots + a_n\vec{b}_n \qquad \forall i \in [n] \tag{6}$$

Finding the shortest vector is equivalent to determining the coefficients a_i for all $i \in [n]$. However, the values of each of the a_i is boundless, which renders naive searching extremely impractical.

The enumeration algorithm exploits the following fact: for each $i \in [n]$, $\|\pi_i(\vec{v}^*)\| \leq \|\vec{v}^*\|$, since $\pi_i(\vec{v}^*)$ is a projection of $\vec{v}^*$ onto a subspace of $\mathbb{R}^n$. Therefore, by the fact that $\pi_i(\cdot)$ vanishes on $\mathsf{span}(\vec{b}_1, \cdots, \vec{b}_{i-1})$, we have

$$\|\pi_i(a_i\vec{b}_i + \cdots + a_n\vec{b}_n)\| = \|\pi_i(\vec{v}^*)\| \leq \|\vec{v}^*\| \qquad \forall i \in [n] \tag{7}$$

Equation (7) gives us a bound for the coefficients. Starting from $k = n$, we have $\|\pi_n(a_n\vec{b}_n)\| = \|a_n\pi_n(\vec{b}_n)\| \leq \|\vec{v}^*\|$, which limits the values of a_n to a set of finite size. Fixing a_n to each candidate, the value of a_{n-1} is also restricted to a finite set. Continuing this procedure, for each selection of $(a_k, a_{k+1}, \cdots, a_n)$, a_{k-1} is restricted to a finite set of values.

Therefore, the entire set of possible combinations of $(a_1, \cdots, a_n)$, that is, all the possible vectors $\vec{v}$, forms a tree. At the root of the tree is the zero vector. The first layer of the tree consists of all the vectors $a_n\vec{b}_n$ such that $\|a_n\pi_n(\vec{b}_n)\| \leq \|\vec{v}^*\|$. For each vector $\vec{v}$ in the k'th layer of the tree, its children consists of all the vectors $\vec{v} + a_{n-k}\vec{b}_{n-k}$ such that $\|\vec{v} + a_{n-k}\pi_{n-k}(\vec{b}_{n-k})\| \leq \|\vec{v}^*\|$. Fig.4 shows the structure of the searching tree, and gives an example of the vertices in a 2-dimensional searching tree.

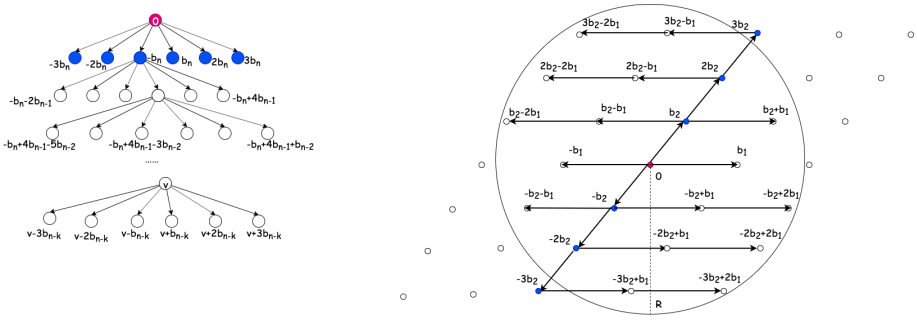

Figure 4: Searching Tree

However, we have no idea what $\|\vec{v}^*\|$ is before we actually find $\vec{v}^*$. Therefore, we must replace this bound with some value R that we already know. Of course, if we choose R to be larger than $\|\vec{v}^*\|$, such as $\|\vec{b}_i\|$ for any i, this algorithm will successfully find $\vec{v}^*$ with probability 1, but the amount of computation increases for larger R. On the other hand, smaller R improves the efficiency, but decreases the success probability. Further more, for

different layers, we can use a seperate bound R_k. Selecting these bounds is a major point of research for improvement of the enumeration algorithm.

The computational cost of the enumeration algorithm is measured by the number of nodes in the searching tree, which is denoted by N. Chen-Nguyen found a method to minimize N subject to a probability p to successfully find a vector of length less than $\alpha \cdot \mathrm{GH}(L)$, where $\alpha > 0$ is a parameter. We use $\mathrm{ENUMCost}(B; \alpha, p)$ to denote the computational cost N to find a short vector in B under these parameters.

2.5 LLL Algorithm

LLL is used as a subprocedure of the basic BKZ algorithm. The name is an acronym of its authors, A.K. Lenstra, Jr Lenstra, and L. Lovasz. It is possible to understand BKZ while regarding LLL as a blackbox. In fact, in the more advanced versions of BKZ, LLL is replaced by other procedures such as BKZ itself (with much smaller blocksize). However, LLL is so foundamental in the lattice reduction that we feel compulsory to provide a description here.

LLL is a generalization of the Lagrange/Gauss algorithm which deals with two-dimensional lattices.

2.5.1 The Lagrange/Gauss algorithm

The Lagrange/Gauss algorithm takes two vectors $\{\vec{b}_0, \vec{b}_1\}$, and update them such that $\|\vec{b}_0\| \leq \|\vec{b}_1\|$ and the Gram-Schmidt coefficient $\mu_{1,0} \leq 1/2$. Recall that the Gram-Schmidt coefficient is the ratio of the length of the projection of $\vec{b}_1$ on $\vec{b}_0$, and the length of $\vec{b}_0$.

Note that if the vectors can take coefficients over the entire $\mathbb{R}$ (instead of points in lattices), the Lagrange/Gauss algorithm is as simple as one step of the Gram-Schmidt orthogonization: take $\vec{b}_1$, subtract the projection of it on $\vec{b}_0$, which is $\mu_{1,0}\vec{b}_0$ by definition, and everything is done. However, to keep the vectors in lattice, the updated vectors must be linear combinations of the original vectors with integral coefficients. The Lagrange/Gauss algorithm thus replaces $\mu_{1,0}$ with $\lfloor \mu_{1,0} \rceil$ in the subtraction.

However, the updated pair of vectors $\vec{b}_0$ and $\vec{b}_1 - \lfloor \mu_{1,0} \rceil \vec{b}_0$ may not satisfy the requirement that $\mu_{1,0} \leq 1/2$. Therefore, the Lagrange/Gauss algorithm continues by swapping the vectors and repeat the above procedure, until the requirement is satisfied, at which time $\mu_{1,0} \leq 1/2$ which means the above procedure would not modify the vectors.

The procedure of Lagrange/Gauss algorithm is illustrated in Fig.5.

2.5.2 Extension to Higher Dimensions

The LLL algorithm is a generalization of the above procedure to multiple dimensions. However, due to the additional complexity and freedom of choices, this generalization can take many directions, and none of them is proved to be the optimal.

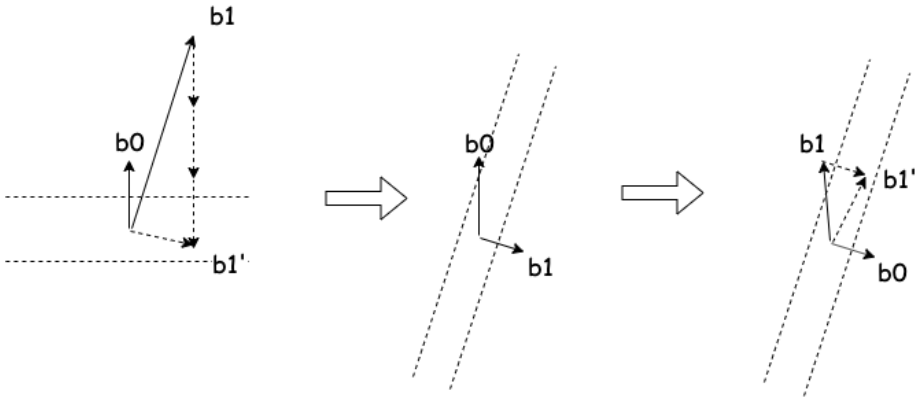

Figure 5: Lagrange / Gauss Reduction

Lovasz condition. A basis $\{\vec{b}_0, \vec{b}_1, \cdots, \}$ is said to satisfy the Lovasz condition for δ if for every i, $\|\vec{b}_i\|/\|\vec{b}_{i-1}\| \geq \delta - \mu_{i,i-1}^2$, where δ is a parameter s.t. $\delta \in (1/4, 1)$. A typical choice of δ is $3/4$. Note that for the two-dimensional case, δ can take the value of 1. However, for the general case, setting δ too close to 1 costs efficiency.

LLL-reduced basis. A basis $\{\vec{b}_0, \vec{b}_1, \cdots, \}$ is said to be LLL-reduced if it satisfies the Lovasz condition, and $\mu_{i,j} \leq 1/2$ for each pair of $i > j$.

LLL algorithm. The LLL algorithm works as follows. Start with $i = 1$, and for $j = i-1$ down to 0, subtract $\lfloor \mu_{i,j} \rceil \vec{b}_j$ from $\vec{b}_i$, just as in the Lagrange/Gauss algorithm. After that, check if Lovasz condition holds for $\vec{b}_i$ and $\vec{b}_{i-1}$, and if so, increment i, and repeat the above process. If the Lovasz condition does not hold, swap $\vec{b}_i$ and $\vec{b}_{i-1}$ and decrement i, and repeat the above process.

Table 1 summarizes the ways LLL algorithm generalizes the Lagrange/Gauss algorithm.

In Lagrange/Gauss algorithm	**In LLL algorithm**
Subtract $\lfloor \mu_{1,0} \rceil \vec{b}_0$ from $\vec{b}_1$	Subtracting $\lfloor \mu_{i,j} \rceil \vec{b}_j$ from $\vec{b}_i$ for each j from $i-1$ down to 0
Swap $\vec{b}_0$ and $\vec{b}_1$ if condition $\mu_{1,0} \leq 1/2$ does not hold	Swap $\vec{b}_{i-1}$ and $\vec{b}_i$ if Lovasz condition does not hold, and decrement i
Succeed when condition does hold	Increment i if Lovasz condition does hold for i

Table 1: Generalization of LLL w.r.t. Lagrange/Gauss Algorithm

LLL runs in polynomial time, if $\delta \in (1/4, 1)$, which is proved in the LLL paper [LLL+82] as it is proposed in 1982.

3 Basic BKZ Algorithm

In this section, we introduce the most basic BKZ algorithm. BKZ solves SVP by invoking LLL reduction and enumeration algorithms as subprocedures.

An execution of BKZ starts with input basis $(\vec{b}_1, \cdots, \vec{b}_n)$, and procedes in several rounds. In each round, BKZ updates the basis by a procedure that is introduced as follows. If in some round the basis stays unmodified after this procedure, the algorithm finishes.

In each round, BKZ iterates for i from 1 up to n. In each iteration, BKZ first invokes the enumeration algorithm to find a solution of the exact SVP in the block B_i of blocksize β, where β is fixed as a global parameter for this algorithm. Let $\vec{b}^*$ be the solution produced by the enumeration algorithm. After that, $\vec{b}^*$ is inserted to the basis in the i'th position, and we get an extended basis $B' = (\vec{b}_1, \vec{b}_2, \cdots, \vec{b}_{i-1}, \vec{b}^*, \vec{b}_i, \vec{b}_{i+1}, \cdots, \vec{b}_n)$ of size $n+1$. Finally, a reduction algorithm (LLL or small blocksize BKZ) is invoked to reduce B' back into a basis of size n, which completes this one iteration.

Note that the vector $\vec{b}^*$ produced in each iteration is actually in the projective lattice by π_i. Therefore, $\vec{b}^*$ is not necessarily a point in the original lattice. However, it is easy to get the coefficients of $\vec{b}^*$ as a linear combination of the projected basis $(\pi_i(\vec{b}_i), \cdots, \pi_i(\vec{b}_{i+\beta-1}))$. Apply the same linear combination to the original vectors $(\vec{b}_i, \cdots, \vec{b}_{i+\beta-1})$, and we get the original vector before projection.

Also note that in the start of each iteration, the basis is LLL-reduced, except in the first iteration in the first round. Therefore, at the start of the BKZ algorithm, we apply the LLL reduction to the basis.

4 BKZ 2.0

The BKZ 2.0 algorithm proposed by Chen and Nguyen [CN11] optimizes the original BKZ algorithm in the following ways:

- **Extreme pruning technique:** Instead of applying the enumeration algorithm to the local block B_i, BKZ 2.0 chooses a low probability p, randomizes B_i to $M = \lfloor 1/p \rceil$ blocks $G_1, \cdots, G_M$, and apply the enumeration algorithm to each G_i.
- **Limits point searching radius:** BKZ 2.0 sets the radius in the enumeration algorithm to $\min\{\alpha \mathrm{GH}(B_i), \|\vec{b}_i^*\|\}$ where $\alpha = \sqrt{1.1}$.
- **Local-block preprocessing:** Before executing the enumeration algorithm, BKZ 2.0 applies a sequence of BKZ algorithms to preprocess the local block, to reduce the cost of enumeration algorithm.
- **Early termination:** BKZ 2.0 uses the early terminating technique [HPS11], by aborting BKZ with small number of rounds, instead of waiting for a round in which

the basis stays unmodified.

BKZ 2.0 Simulator. The paper [CN11] that proposed BKZ 2.0 also presents a simulator to estimate the cost of BKZ 2.0 algorithm. Based on the simulation results, [CN11] also gives a table of cost of the enumeration algorithm Cost_β with respect to the blocksize β.

The simulator simulates the BKZ 2.0 algorithm by going through the procedure, without processing real vectors. Instead, the simulator starts with the array $(\ell_1, \cdots, \ell_n)$ representing the lengths of Gram-Schmidt vectors $\|\vec{b}_i^*\|$, and update the lengths in the rounds.

According to the table of simulation results, [AWHT16] extrapolates the cost as a function of β as:

$$\log_2(\text{Cost}_\beta) = 0.000784314\beta^2 + 0.366078\beta - 6.125 \tag{8}$$

Finally, the total cost of BKZ 2.0 (which is dominated by enumeration) is given by:

$$\#rounds \cdot \sum_{i=1}^{n-1} \text{Cost}_{\min\{\beta, n-i+1\}} \tag{9}$$

5 Progressive BKZ

Another strategy to improve BKZ algorithm is to progressively adapt the blocksize parameter during the execution of the algorithm. The BKZ algorithms that make use of this strategy are called progressive BKZ algorithms. The efficiency of BKZ is greatly impacted by the blocksize parameter. Specifically, increasing the blocksize produces shorter vectors as solutions to the enumeration algorithm, with the drawback of increasing the computational cost.

The progressive strategy is based on the following observation: with fixed blocksize, the computational cost with gradually drop after each rounds. Starting the BKZ with small blocksize, and gradually increasing it over the rounds of the execution, saves considerable amount of computation, without sacrificing the quality of the solution.

The progressive strategy is never implemented alone in the BKZ algorithm. Instead, it is often used as one of the strategies in other works trying to improve the performance of the BKZ algorithm. Therefore, there is no formal proposition of the so called "The Progressive BKZ algorithm".

The progressive strategy is usually utilized in two ways [HR$^+$19]:

1. **Globally**. The blocksize of the entire BKZ algorithm is updated gradually instead of being fixed.

2. **Locally**. The progressive strategy focuses on the local-block preprocessing, which could also be a BKZ algorithm.

BKZ 2.0 belongs to the second category of progressive BKZ algorithms [CN11]. In fact, BKZ 2.0 implements the local-block preprocessing by another BKZ algorithm with much smaller blocksize, and makes use of the progressive strategy in this procedure.

One of the greatest challenges in the research of Progressive BKZ algorithm is to find an affective strategy to increase the blocksize. This is also one of the key contribution of the algorithm proposed by [AWHT16], which will be explained in detail in the next section.

6 Improved Progressive BKZ

This section describes the improved progressive BKZ algorithm proposed by [AWHT16] in detail. The most crucial of this algorithm is a strategy to decide when to increase the blocksize. In summary, [AWHT16] implements a simulator which gives an estimation of the enumeration algorithm, provided with parameters n and blocksize β. At the end of each round, if the actual computational cost is less than the simulated cost, this is the time to increase the blocksize. If the blocksize already reaches the predetermined β_{end}, the algorithm stops.

The improved progressive BKZ algorithm does not share many similarities with BKZ 2.0. In fact, the only improvement they both have over the original BKZ is the preprocessing of local blocks. However, the improved progressive BKZ makes use of the BKZ 2.0 simulator to compute the creteria for increasing blocksize.

6.1 Optimizing parameters for local block

The paper [AWHT16] proposes a method to obtain the optimal parameters (α, p) for the local block B_i of size β such that the efficiency of enumeration algorithm is optimized.

First, fix α and β, the optimal success probability p and the resulting GSA constant r can be obtained as follows:

$$p = \frac{2}{\alpha^{\beta}} \tag{10}$$

$$r = \left(\frac{\beta+1}{\alpha\beta}\right)^{4/(\beta-1)} \cdot V_{\beta}(1)^{4/(\beta(\beta-1))} \tag{11}$$

Conversely, by fixing (β, r), one can first compute α by equation (11) and then get p by equation (10). Therefore, for a given GSA constant r, it is possible to search the optimal β which minimizes $\mathrm{ENUMCost}(B_i; \alpha, p)$. An estimation of $\mathrm{ENUMCost}(B_i; \alpha, p)$ is given by:

$$D = 2\alpha^{-\beta/2} \frac{V_{\beta/2}(1) V_{\beta}(1)^{-1/2}}{r^{\beta^2/16}} \tag{12}$$

Then from equation (11) remove the α in 12:

$$D \approx Const. \times r^{(\beta^2-2\beta)/16} \cdot \left(\frac{\beta}{e\pi}\right)^{\beta/4} \tag{13}$$

Letting the partial derivative of β be zero gives:

$$\log(r) = \frac{\log(c_1\beta + c_2)}{c_3\beta + c_4} \tag{14}$$

where c_1, c_2, c_3 and c_4 are constants needed to be fitted. Using the data collected from experiments, however, it is hard to fit the pairs $(\beta_i, \log r_i)$ with a single function $f(\beta)$. Therefore, [AWHT16] ends up with two functions. In summary

$$\log(r) = f(\beta) = \begin{cases} f_1(\beta) & = & -18.2139/(\beta + 318.978) & \beta \leq 100 \\ f_2(\beta) & = & (1.06889/(\beta - 31.0345)) \cdot \log(0.417419\beta - 25.4889) & \beta > 100 \end{cases} \tag{15}$$

With all the above equations, the $\text{ENUMCost}(B_i; a, \alpha)$ is represented as a function of β. To simplify this function, fitting experiments data pairs $(\beta, \text{ENUMCost}(B_i; a, \alpha))$ gives:

$$\log_2 \text{MINCost}(\beta) := \begin{cases} 0.1375\beta + 7.153 & \beta \in [60, 105] \\ 0.000898\beta^2 + 0.270\beta - 16.97 & \beta > 105 \end{cases} \tag{16}$$

Finally, for a fixed blocksize β, find the optimal parameters (α, p) by first computing r with equation (15), then getting α by (11), and finally p by (10).

6.2 Estimating Enumeration Cost

The full enumeration cost (FEC) is the number of nodes in the searching-tree, which is given by

$$\text{FEC}(B) = \sum_{k=1}^{n} \frac{V_k(\text{GH}(L))}{\prod_{i=n-k+1}^{n} \|\vec{b}_i^*\|} \tag{17}$$

where $\|\vec{b}_i^*\|$ is the Gram-Schmidt length.

To simulate the FEC, first simulate the Gram-Schmidt lengths $(\ell_1, \cdots, \ell_n) = \text{Sim-GS-Lengths}(n, \beta)$ depending on dimension n and blocksize β. Then, simulate the Gaussian Heuristic by $\text{Sim-GH}(\ell_1, \cdots, \ell_n) = V_n(1)^{-1/n} \prod_{j=1}^{n} \ell_j^{1/n}$. We thus get the simulated FEC

$$\text{Sim-FEC}(\ell_1, \cdots, \ell_n) = \sum_{k=1}^{n} \frac{V_k(\text{Sim-GH}(\ell_1, \cdots, \ell_n))}{\prod_{i=n-k+1}^{n} \ell_i} \tag{18}$$

Simulate Gram-Schmidt Lengths. First, set $\ell_n = 1$, then compute ℓ_i backwards, by solving the following equation:

$$\ell_i = \max\left\{\frac{\beta'}{\beta'+1}\alpha, \tau_{\beta'}\right\} \cdot \mathrm{GH}(\ell_i, \cdots, \ell_{i+\beta'-1}) \tag{19}$$

where $\beta' = \min(\beta, n-i+1)$.

Next, we optimize the above GS-lengths for the last indexes $i > n-\beta+1$ as follows. Select (α_i, p_i) for each i such that Sim-ENUMCost$(\ell_i, \cdots, \ell_n; \alpha_i, p_i) = \mathrm{MINCost}(\beta)$, where $\alpha_i = (2/p_i)^{n-i+1}$ according to (10). Then solve (19) again with α replaced by the previously selected α_i.

The GS-lengths of the first β indexes are also optimized. However, it is not very clearly described in [AWHT16].

Expected Number of BKZ Rounds for Specific Blocksize To estimate the number of rounds for a specific blocksize $\beta - 1$, i.e. the number of rounds that are executed with this blocksize before the condition for increasing blocksize is satisfied, we simulate the execution of rounds by (a modified version of) the BKZ 2.0 simulator. At the start, compute $(\ell_1, \cdots, \ell_n)$ by Sim-GS-Lengths$(n, \beta-1)$. Note that at this time Sim-FEC$(\ell_1, \cdots, \ell_n)$ is equal to Sim-FEC$(n, \beta-1)$ by definition. Then update those lengths by the BKZ 2.0 simulator, until Sim-FEC$(\ell_1, \cdots, \ell_n)$ drops below Sim-FEC(β, n).

Suppose at the end of round t, Sim-FEC$(\ell_1, \cdots, \ell_n)$ is smaller than Sim-FEC(β, n), while before this round, Sim-FEC$(\ell_1, \cdots, \ell_n)$ was larger. Then the expected number of rounds is estimated to be somewhere between $t-1$ and t. Specifically, assume the input of round t is $(\ell'_1, \cdots, \ell'_n)$ and the output is $(\ell_1, \cdots, \ell_n)$, the "somewhere" is estimated to be $(t-1) + \frac{\text{Sim-FEC}(\beta,n) - \text{Sim-FEC}(\ell'_1, \cdots, \ell'_n)}{\text{Sim-FEC}(\ell_1, \cdots, \ell_n) - \text{Sim-FEC}(\ell'_1, \cdots, \ell'_n)}$.

6.3 Optimizing Blocksize Strategy

If the basis B satisfies that FEC$(B) <$ Sim-FEC(n, β), then we say B is β-reduced. Currently, the blocksize strategy can be described as follows: at the start, B is already $(\beta-1)$-reduced, the current blocksize is β, the BKZ rounds update the basis until at the end of some round FEC$(B) <$ Sim-FEC(n, β), i.e. B is β-reduced, then update the blocksize to $\beta + 1$.

This could be generalized by a blocksize strategy characterized by a tuple $(\beta^{start}, \beta^{goal}, \beta^{alg})$. Concretely, at the start, B is β^{start}-reduced, then update B by blocksize β^{alg}, until B is β^{goal}-reduced. In this generalized definition, the original simple strategy can be described as the tuple $(\beta-1, \beta, \beta)$. A complete blocksize strategy consists of a list of such tuples $\{(\beta_i^{start}, \beta_i^{goal}, \beta_i^{alg})\}_{i=1}^{D}$. And a complete progressive BKZ algorithm loops through this list, apply the BKZ rounds with each strategy until B is β_i^{goal}-reduced.

Usually, β_i^{goal} is the same as β_{i+1}^{start}, so the list could be simplified to $\{(\beta_i^{goal}, \beta_i^{alg})\}_{i=1}^{D}$. Notice that β_1^{start} is not defined in this way, which we simply neglect, because at the start

of the BKZ algorithm B is only LLL-reduced, and it is not sure if B is β-reduced for any β.

Let $\text{TimeBKZ}(n, \beta_{i-1}^{goal} \xrightarrow{\beta_i^{alg}} \beta_i^{goal})$ denote the time required for the BKZ rounds to update a β_{i-1}^{goal} reduced basis to one that is β_i^{goal}-reduced with blocksize β_i^{alg}. The goal of optimizing blocksize strategy is to minimize the total time:

$$\sum_{i=1}^{D} \text{TimeBKZ}(n, \beta_{i-1}^{goal} \xrightarrow{\beta_i^{alg}} \beta_i^{goal}) \tag{20}$$

Denote by $\text{TimeBKZ}(n, \beta^{goal})$ the minimized total time of the entire BKZ algorithm, whose input is a basis that is LLL-reduced, and the output is one that is β^{goal}-reduced. The minimization is taken over all possible block strategies that are ended with β^{goal}.

By definition, we can also write $\text{TimeBKZ}(n, \beta^{goal})$ recursively as follows:

$$\text{TimeBKZ}(n, \beta^{goal}) = \min_{\beta', \beta^{goal}} \left\{ \text{TimeBKZ}(n, \beta') + \text{TimeBKZ}(n, \beta' \xrightarrow{\beta^{alg}} \beta^{goal}) \right\} \tag{21}$$

In this way, we can compute $\text{TimeBKZ}(n, \beta^{goal})$ in the dynamic programming way by first computing $\text{TimeBKZ}(n, \beta')$ for all $\beta' < \beta^{goal}$, then minimizing equation (21) over all pairs of (β', β^{alg}).

6.4 Implementation of BKZ Rounds and Computational Cost

Now is the time to actually compute the $\text{TimeBKZ}(n, \beta^{start} \xrightarrow{\beta^{alg}} \beta^{goal})$. By definition

$$\begin{aligned} &\text{TimeBKZ}(n, \beta^{start} \xrightarrow{\beta^{alg}} \beta^{goal}) = \\ &\sum_{t=1}^{\#rounds} \sum_{i=1}^{n-1} [\text{Time of processing local block } B_i \text{ with parameter } (\alpha, p)] \end{aligned} \tag{22}$$

where α and p are selected according to the strategy described in Section 6.1 with blocksize β_{alg}.

One iteration in the BKZ round in the improved progressive BKZ algorithm can be decomposed into the following steps. We will give the cost estimation for each step as we describe them.

Compute Gram-Schmidt lengths. In this step, we need to compute the Gram-Schmidt variables $\|\vec{b}_i^*\|$ and μ_{ij} for the local block B_i. They are needed in the computation in the following steps, for example, to estimate the optimal parameters α and p. In implementation, the Gram-Schmidt variables could be precomputed for the entire basis B, and for each block B_i we can simply copy the corresponding parts.

The cost of this step is negligible.

Compute bounding coefficients. We compute the bounding coefficients $(R_1, \cdots, R_\beta)$ using Aono's precomputing technique. Recall that these bounding coefficients are used in the enumeration algorithm for each layer in the searching tree.

The cost of this step is negligible.

Preprocess local block. We preprocess the local block B_i using this improved progressive BKZ algorithm, but with much smaller blocksizes. The blocksize strategy here takes the simple one-by-one strategy starting from 15.

The cost of this step is estimated to be that of the enuemration times a constant $A_{Preprocess}$. The constant will be determined by experiments.

Optimize bounding coefficients. We optimize the bounding coefficients $(R_1, \cdots, R_\beta)$ after preprocessing B_i, if the estimated number of nodes in the searching tree is larger than 10^8. This procedure is a simple algorithm that considers random perturbation of $(R_1, \cdots, R_\beta)$, which is also used in BKZ 2.0.

The cost of this optimization step is significantly smaller than the enumeration step. In small blocksizes, the optimization is often skipped, and the time consumption of this step is approximately zero. For large blocksizes, the enumeration step is also much larger than this step. Therefore, it is safe to ignore the computational cost here.

Enumeration. The enumeration is implemented in the standard way. In the end, it outputs $h = 16$ vectors whose projective lengths are small, and store them in $(\vec{v}_1, \cdots, \vec{v}_h)$.

The cost of this step is given by $A_{Enum} \cdot \beta \cdot \text{Sim-ENUMCost}(\ell_i, \cdots, \ell_{i+\beta-1}; \alpha, p)$, where A_{Enum} is a constant.

Construct degenerated basis. After we get the vectors $(\vec{v}_1, \cdots, \vec{v}_h)$ from enumeration, we select some of them and insert into the i'th position of basis $(\vec{b}_1, \cdots, \vec{b}_n)$ as follows. Starts with $\hat{B} = (\vec{b}_1, \cdots, \vec{b}_{i-1})$. In each step we select and remove one $\vec{v}_j$ from the vectors, and append it to the end of $\hat{B}$. The $\vec{v}_j$ selected in each step is the one that minimizes the norm of the projection of $\vec{v}_j$ onto the current $\hat{B}$, which is denoted by $\|\pi'(\vec{v}_j)\|$. Moreover, it is required that $0 < \|\pi'(\vec{v}_j)\| < \|\text{last vector of } \hat{B}\|$. If no such $\vec{v}_j$ satisfies this requirement, then stop the procedure and return the current $\hat{B}$ joined with the rest of basis, i.e. $(\vec{b}_i, \cdots, \vec{b}_n)$.

This cost of this step is negligible.

Apply LLL to degenerated basis. Assume that the degenerated basis created in the previous step is larger than the blocksize by g, i.e. the size is $i + \beta' - 1 + g$. We can assume that the first $i - 1$ vectors in the basis is already LLL-reduced, the computational cost can be approximated by applying LLL to the later $\beta' + g$ vectors in the basis, which is $O(n^2 \cdot \beta^2) = A_1 \cdot \beta^2 n^2$ where A_1 is constant.

Total Cost and Constants Fitting. Finally, the time cost for the entire BKZ algorithm can be expressed as follows.

$$\text{Time}(n, \beta, A_1, W_1) = \sum_{\beta^{start}}^{\beta^{goal}} \sum_{t=1}^{\#rounds} \left[A_1 \cdot \beta^2 n^3 + W_1 \cdot \beta \sum_{i=1}^{n-1} \text{ENUMCost}(B_i; \alpha, p) \right] \quad (23)$$

where A_1 and W_1 are constants that are determined by experiments. The coefficients are fitted by standard curve fitting method in semi-log scale. The results given in [AWHT16] are $A_1 = 1.5 \cdot 10^{-10}$ and $W_1 = 1.5 \cdot 10^{-8}$.

6.5 Pre/Post-Processing the Basis

By preprocessing or postprocessing the basis, it is possible to enhance the speed of the progressive BKZ algorithm. The preprocessing works similar to the extreme pruning algorithm proposed by [GNR10], by generating a number of randomized bases, reducing each of them by this progressive BKZ algorithm. In the postprocessing part, apply the enumeration algorithm with low success probability to each of the basis to find short vectors.

Concretely, the algorithm works as follows. Randomly generate M unimodular matrices U_i, and produces M randomized basis $U_i B$. Then apply the progressive BKZ to each of the random basis. In the postprocessing, set $\alpha = \gamma$ and probability $p = 2 \cdot \gamma^{-n}/M$ and apply the enumeration algorithm.

7 Conclusion

In this article we introduced the series of BKZ algorithms, including the preliminaries necessary for understanding BKZ. Specifically, we introduced Gram-Schmidt reduction, Gaussian Heuristic, LLL reduction and the enumeration algorithm. Based on the knowledge of above, we described the details of the basic BKZ algorithm, BKZ 2.0, Progressive BKZ, and finally the improved progressive BKZ proposed by [AWHT16].

The improved progressive BKZ modifies the original BKZ algorithm in the following aspects:

1. Implement the progressive strategy by increasing the blocksize when the actual cost of a round is smaller than the simulated cost.
2. Terminate the algorithm when the blocksize increases over β_{end}.
3. Preprocess the local block with progressive BKZ algorithm of smaller blocksize.
4. Apply extreme pruning technique globally.

References

[AWHT16] Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi. Improved progressive bkz algorithms and their precise cost estimation by sharp simulator. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 789–819. Springer, 2016.

[CN11] Yuanmi Chen and Phong Q Nguyen. Bkz 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011.

[GNR10] Nicolas Gama, Phong Q Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 257–278. Springer, 2010.

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In *Annual Cryptology Conference*, pages 447–464. Springer, 2011.

[HR+19] Md Haque, Mohammad Obaidur Rahman, et al. Analyzing progressive-bkz lattice reduction algorithm. *International Journal of Computer Network & Information Security*, 11(1), 2019.

[LLL+82] Hendrik Willem Lenstra, Arjen K Lenstra, L Lovfiasz, et al. Factoring polynomials with rational coeficients. 1982.

[SE94] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, 1994.