A Greedy Global Framework for LLL

Sanjay Bhattacherjee, Julio Hernandez-Castro, and Jack Moyler[⋆]

Institute of Cyber Security for Society and School of Computing,
Keynes College, University of Kent, CT2 7NP, UK
{s.bhattacherjee,j.c.hernandez-castro,j.moyler}@kent.ac.uk

Abstract. LLL-style lattice reduction algorithms employ two operations on ordered basis vectors - size reduction and reordering - to improve the basis quality by iteratively finding shorter and more orthogonal vectors. These algorithms typically have two design features. First, they work with a local or global measure of basis quality. Second, they reorder a subset of the basis vectors based on the basis quality before and after reordering. In this work, we introduce a new generic framework for designing lattice reduction algorithms. An algorithm in the framework makes greedy basis reordering choices globally on the whole basis in every iteration, based on a measure of basis quality. The greedy choice allows to attain the desired quality very quickly making the algorithms extremely efficient in practice. The framework is instantiated using two quality measures (1) the potential of the basis, and (2) the squared sum of its Gram-Schmidt orthogonalised vectors, to get two new basis reduction algorithms. We prove that both algorithms run in polynomial time and provide quality guarantees on their outputs. Our squared sum based algorithm has runtime close to LLL while outperforming BKZ-12 in output quality at higher dimensions. We have made our implementations and the experimental results public.

Keywords: Lattice reduction · LLL · DeepLLL · BKZ · greedy global framework · potential · squared sum.

1 Introduction

A Euclidean lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^m$. It can always be represented by a basis matrix $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in \mathbb{R}^{m \times n}$ made of linearly independent column vectors $\mathbf{b}_i \in \mathbb{R}^m$ such that $\mathcal{L} = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}$. There are infinitely many bases for any lattice with $n \geq 2$ and there are ways to transform a basis into another for the same lattice. The quality of a given lattice basis is determined by the length of the vectors and how close to orthogonal they are from each other. Bases with shorter and more orthogonal vectors are considered to be of better quality. Given a lattice specified by a basis, finding a good quality basis and short vectors therein is of importance. The process of transforming a given basis into one of better quality is generally called lattice reduction. It has

⋆ Corresponding author

many applications including in cryptology [24], algorithmic number theory [5], etc. In particular, it is used as a subroutine in the Block Korkine-Zolotarev (BKZ) [28] algorithm for efficient lattice reduction and establishing records of shortest vectors [6] in lattices.

In 1982, Lenstra, Lenstra and Lovász [18] presented the first lattice reduction algorithm that came to be called LLL after its inventors. LLL uses the Gram-Schmidt orthogonalisation (GSO) $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$ of the basis $\mathbf{B}$. The GSO process assumes an inherent ordering of the vectors and LLL works with the same order. Starting from index $k = 2$ of the ordered basis, LLL traverses up and down the order in a loop by incrementing or decrementing the index k by 1 in each iteration. There are two kinds of operations – size reductions and swaps – that are executed within the loop, until the entire basis is of sufficiently good quality. The quality of the basis is determined by the optimisation criterion called the Lovász condition (LC) on all pairs of consecutive vectors $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$. This condition is given by $\left\|\mathbf{b}_k^* + \mu_{k,k-1}\mathbf{b}_{k-1}^*\right\|^2 \geq \delta \left\|\mathbf{b}_{k-1}^*\right\|^2$, where the $\mu_{i,j}$'s are the GSO coefficients. After LLL terminates, vector $\mathbf{b}_i$ in the output basis is an exponential approximation of the i^{th} shortest linearly independent vector in the lattice. In [18], LLL was shown to run in polynomial time. They used an argument surrounding a quantity known as the *potential* of the basis which is essentially a measure of basis quality. The potential was further analysed in [14], where the author discussed a notion of lattice reduction based on maximally reducing the basis potential for a given lattice.

Schnorr and Euchner introduced a variant of the LLL algorithm called LLL with deep insertions, or DeepLLL [27]. The key algorithmic novelty was in the reordering of the vectors. They introduced the notion of deep insertions whereby instead of just swapping vector $\mathbf{b}_k$ with the immediate previous vector $\mathbf{b}_{k-1}$, it could be inserted before any one of the previous vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$. This essentially meant that the index k could be decremented to any value between $\{2, \ldots, k-1\}$. They also extended the LC-constraint from consecutive pairs $(\mathbf{b}_{k-1}, \mathbf{b}_k)$ to all pairs $(\mathbf{b}_i, \mathbf{b}_k)$ for $i < k$ in the ordering[1]. This introduced more constraints on the output basis and as a result, the quality of the output basis is provably better [31] than in LLL. In other words, the i^{th} vector of the output basis is a better approximation of the i^{th} shortest linearly independent vector of the lattice, as compared to the LLL output [31, Theorem 1]. However, DeepLLL requires additional size reduction steps and bookkeeping that make it significantly more time-consuming than LLL.

Since Schnorr and Euchner introduced DeepLLL, there have been two new deep-insertion based algorithms - Pot-LLL [8] and SS-LLL [31]. These algorithms replace the extended Lovász condition of DeepLLL with a check on the improvement of a basis quality. They use the quality measures *potential* $\mathrm{Pot}(\mathbf{B}) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|^{2(n-i+1)}$ and *squared sum* $\mathrm{SS}(\mathbf{B}) = \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2$ respectively, computed directly from the Gram-Schmidt orthogonalised basis. To stress that they are essentially variants of DeepLLL, we call them Pot-DeepLLL and SS-DeepLLL respectively. They are both polynomial-time algorithms that pro-

[1] A pair $(\mathbf{b}_i, \mathbf{b}_k)$ in a basis can simply be identified by the pair of indices (i, k).

vide efficiency versus basis quality trade-offs in between LLL and DeepLLL. They typically find shorter vectors than LLL but not as short as DeepLLL. They are slower than LLL, but faster than DeepLLL.

In every iteration of DeepLLL and its variants Pot-DeepLLL and SS-DeepLLL, the algorithms only work with the sublattice $\mathcal{L}_k$ generated by a subset $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. Each of these algorithms attempt to iteratively improve the respective measure. The use of $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ as measures of quality has been quite clear in the proofs of basis quality and runtime complexity of these algorithms. However, DeepLLL has not been interpreted as or represented in a form where it is improving an explicit quality measure in every iteration, to the best of our knowledge. We do this exercise of interpreting the (generalised) Lovász condition as a reordering constraint used to improve the length $\|\mathbf{b}_i^*\|$ of the i^{th} GSO vector of the basis, which is a localised measure of quality of the basis. In contrast, $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ are global measures on the entire basis. We thus have a generalised understanding of all three algorithms based on deep insertions looking to improve quality measures of a basis.

We propose a new framework for LLL-style algorithms whose novelty lies in the reordering of the basis at a deep insertion step. All previous LLL-style algorithms [18,27,8,31] maintain an index k of the vector to be inserted at a previous position $i \in \{1, \ldots, k-1\}$ in the basis ordering. In the algorithms using deep insertions [27,8,31], this restricts the deep insertion choices within the sublattice $\mathcal{L}_k$ in an iteration. Our framework defines a generalised algorithm X-GGLLL for lattice reduction that works with a general quality measure $X(\mathbf{B})$ of the basis. We move away from the technique of maintaining an index k. Instead, *we make a dynamic greedy choice of a pair of indices (i, k), $1 \leq i < k \leq n$ globally over the entire basis such that the deep insertion of $\mathbf{b}_k$ at position i minimises the basis quality measure $X(\cdot)$.* Such deep insertions are carried out as long as the measure of the reordered basis decreases by *at least* a fraction $(1-\delta)$ of its previous value. When the algorithm terminates, the output basis is guaranteed to have a measure that can not be reduced appreciably (by a fraction $(1-\delta)$) any further through deep insertions. For a measure X, we call such a basis δ-X-DeepLLL reduced. *By choosing the maximum change possible at each iteration, our greedy algorithm reaches such a state in a small (if not the smallest) number of iterations.* When the measure has a positive lower bound, the algorithm is guaranteed to terminate.

The choice of the measure $X(\cdot)$ is a key determining factor in the framework of algorithms we propose. We instantiate our generalised algorithm X-GGLLL with the measures $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ in place of $X(\cdot)$ to get the Pot-GGLLL and SS-GGLLL algorithms respectively. We prove that X-GGLLL outputs a δ-X-DeepLLL reduced basis and provide theoretical bounds on the runtime of X-GGLLL. We prove the concrete polynomial runtime complexities for both Pot-GGLLL and SS-GGLLL and show that they are the same as their X-DeepLLL counterparts.

We conduct extensive experiments to assess the performance of our algorithms in comparison with LLL [18], Pot-DeepLLL [8], SS-DeepLLL [31], and

BKZ (including preprocessing with LLL) [27] with blocksizes $8, 10, 12$ and 20. Our greedy global algorithms simultaneously provide excellent runtime as well as output quality on the average. They significantly outperform their respective DeepLLL counterparts on both these counts. The average runtime of SS-GGLLL is only second to LLL among all algorithms under consideration, a result that is observed consistently in all dimensions. It is the only LLL-style algorithm to outperform BKZ-12 in terms of output quality as well in our experiments. Even though it does not match the quality of BKZ-20, at dimension 150, it is almost 15 times faster than BKZ-20 while providing output quality better than BKZ-12. The excellent experimental runtimes of our algorithms are in contrast with their asymptotic runtimes. We conduct further granular analysis of the runtime performance of all LLL-style algorithms using the number of reorderings and the number of size reductions of basis vectors to provide justifications therein. Our implementations, the input bases and the output values we report are available at [12].

The outline of the paper is as follows. Section 2 details the relevant notation and gives an overview of lattices. Section 3 provides a description of LLL and generalises DeepLLL for any measure. Section 4 proposes the greedy global framework as a novel way of reducing lattice bases. Sections 5 and 6 provide theoretical analysis and experimental results respectively.

Related Works. Yamaguchi and Yasuda in [30] described an efficient algorithm for updating the GSO information in DeepLLL. Since the update of the GSO information is dominant in such an algorithm, this work is of great importance to our framework. In [32], it was proved that in LLL, the value of the squared sum $\mathrm{SS}(\mathbf{B})$ decreases with every swap. The complexity of LLL [18] and Pot-DeepLLL [8, Proposition 1] for an input basis $\mathbf{B}$ is bounded by the size of $\mathrm{Pot}(\mathbf{B})$. The complexity of SS-DeepLLL is bounded by the size of $\mathrm{SS}(\mathbf{B})$ [31]. Fukase and Kashiwabara [9] showed that a basis with a smaller squared-sum allows more short lattice vectors to be sampled using Schnorr's random sampling. This method was used in [32] to sample short vectors.

The original LLL algorithm [18] was known to run in polynomial time for the reduction parameter $\delta < 1$. For $\delta = 1$, it is known to be polynomial time, but only for fixed dimensions [1]. Although DeepLLL [27] is not known to run in polynomial time, its variants Pot-DeepLLL [8] and SS-DeepLLL [31] are both polynomial time algorithms.

In [4], the authors pointed out that $\mathrm{Pot}(\cdot)$ does not capture the typical unbalancedness demonstrated by the GSO norms. They introduced a new potential function based on the sublattice $\mathcal{L}_k$ generalising the one depending on the entire basis and demonstrated their usefulness.

An important direction in improving the efficiency of LLL has been considering the implementation details of the algorithms and the consequent optimisations. In [27], a practical LLL algorithm using floating-point arithmetic was described, which has been extended by Nguyen and Stehlé [22] in their very efficient L^2 algorithm. This is an important direction in lattice basis reduction. In [21], an asymptotically fast variant of LLL was proposed that relies on fast integer arithmetic. Another variant of LLL was introduced in [20], where the costly GSO computations are approximated by Householder transformations which are performed using floating-point arithmetic. In [3], a perturbation analysis has been performed on the $\mathbf{QR}$ factor $\mathbf{R}$ of LLL-reduced bases under columnwise perturbation. The results obtained may be applied to the floating-point implementations of LLL-type algorithms. LLL has also been adapted for use in the Information-Set Decoding algorithm for binary codes in [7] to obtain a small speed-up. Lenstra introduced the idea of a flag in lattice reduction in [19], where a flag is defined to carry a little less information than is provided by a lattice basis. The flag is then reduced within the LLL algorithm by performing successive steps which replace flags by neighbouring ones for reduction. In [15], the authors used parallelisation and recursion to improve the efficiency of LLL. Koy and Schnorr [16] introduced the Segment LLL algorithm - a variant of LLL which yields a slightly weaker reduced basis but is more efficient by a factor n. Another important direction is the application of LLL to lattices with an underlying structure or form, for example ideal lattices [25], module lattices [17] and parametric lattices [2]. It is well-understood that LLL generally provides much better output quality than the analysis of the LLL-reduced bases suggests. To this end, in [23] and [26], experimental analyses have been performed on the average-case behaviour of LLL and comparisons are drawn with the worst-case theoretical results. We note that the key ideas associated with the directions in this paragraph are more or less orthogonal to the techniques we introduce in this paper.

2 Preliminaries

Notation. The sets of integers, rational and real numbers are denoted by $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ respectively. Let $[n] = \{1, \ldots, n\}$. For $x \in \mathbb{R}$, $|x|$ denotes its absolute value. The integer closest to $x \in \mathbb{R}$ is denoted by $\lfloor x \rceil$. All vectors are column vectors. The Euclidean norm of a vector $\mathbf{x} \in \mathbb{R}^m$ is denoted by $\|x\|$. The inner product of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ is denoted by $\langle \mathbf{x}, \mathbf{y} \rangle$. All logarithms are base 2 unless denoted otherwise.

Lattice, Bases, Sublattice and Linear Span. A lattice $\mathcal{L} = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}$ specified by an ordered set of linearly independent vectors called a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in \mathbb{R}^{m \times n}$, is denoted as $\mathcal{L}(\mathbf{B})$. We call m the dimension and n the rank of the lattice $\mathcal{L}$, where $m \geq n$. The linear span of $\mathbf{B}$ is given by $\mathtt{span}(\mathbf{B}) = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{R}^n\}$. A subset of vectors in $\mathbf{B}$ give rise to a sublattice of $\mathcal{L}(\mathbf{B})$. For example, given a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ for a lattice $\mathcal{L}$, the vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_i), 1 \leq i \leq n$ form a basis of a sublattice of $\mathcal{L}$ that we denote as $\mathcal{L}_i$.

For a lattice of dimension $n \geq 2$, there are infinitely many bases. If $\mathbf{B}_1$ is a basis for a lattice $\mathcal{L}$, we may transform this into another basis $\mathbf{B}_2$ for the same lattice by $\mathbf{B}_2 = \mathbf{B}_1\mathbf{U}$, where $\mathbf{U} \in GL_n(\mathbb{Z})$ is a unimodular matrix. An invariant across the infinitely many bases of a lattice is its volume. For a basis $\mathbf{B}$ of the lattice, its volume is given by $\mathrm{Vol}(\mathcal{L}) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}$ and geometrically it represents the volume of the fundamental parallelepiped of the lattice. We generally only consider lattices with vectors in $\mathbb{Q}^m$ and by scaling we need only consider lattices in $\mathbb{Z}^m$.

Gram-Schmidt Orthogonalisation. For an ordered set of linearly independent vectors $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, $\mathbf{b}_i \in \mathbb{R}^m$, its Gram-Schmidt orthogonalisation gives the corresponding set $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$ of orthogonal vectors defined recursively as follows.

- $\mathbf{b}_1^* = \mathbf{b}_1$, and
- for $i > 1$, $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$,

where a GSO coefficient $\mu_{i,j}$ is defined for $1 \leq j \leq i \leq n$ as

$$\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\left\| \mathbf{b}_j^* \right\|^2}.$$

It is easy to see that $\mu_{i,i} = 1$ for all $1 \leq i \leq n$.

Orthogonal Projections. Given a vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$, its projections $\pi_i(\mathbf{v})$ are defined for $1 \leq i \leq n$ as

- $\pi_1(\mathbf{v}) = \mathbf{v}$, and
- for $2 \leq i \leq n$, $\pi_i(\mathbf{v})$ is the projection of $\mathbf{v}$ orthogonal to $\texttt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}))$ of the sublattice $\mathcal{L}_{i-1}$.

The projection $\pi_i(\mathbf{b}_k)$ is written in terms of the GSO vectors $(\mathbf{b}_i^*, \ldots, \mathbf{b}_k^*)$ and the GSO coefficients $\mu_{k,i}, \ldots, \mu_{k,k-1}$ as follows

$$\pi_i(\mathbf{b}_k) = \mathbf{b}_k^* + \sum_{l=i}^{k-1} \mu_{k,l} \mathbf{b}_l^*.$$

In the simplest case, $\pi_i(\mathbf{b}_i) = \mathbf{b}_i^*$.

Lovász condition. For the parameter $1/4 < \delta \leq 1$, the Lovász condition between consecutive vectors $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$ is defined as

$$\delta \cdot \left\| \pi_{k-1}(\mathbf{b}_{k-1}) \right\|^2 \leq \left\| \pi_{k-1}(\mathbf{b}_k) \right\|^2.$$

This can be written as $\left(\delta - \mu_{k,k-1}^2\right) \cdot \left\| \mathbf{b}_{k-1}^* \right\|^2 \leq \left\| \mathbf{b}_k^* \right\|^2$ in terms of the GSO vectors and coefficients. For all $1 \leq i < k \leq n$, the Lovász condition can be generalised (for deep insertions) as

$$\delta \cdot \left\| \pi_i(\mathbf{b}_i) \right\|^2 \leq \left\| \pi_i(\mathbf{b}_k) \right\|^2.$$

Algorithm 1: The size reduction algorithm for a vector $\mathbf{b}_k$

```
  Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, its GSO coefficients $\mu_{i,j}$, and an index $k$.
  Output: A basis $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}'_n)$ where $\mathbf{b}'_k$ is size reduced, and the
          updated coefficients $\mu'_{i,j}$.
1 for $j = k-1, \ldots, 1$ /* The 'reverse order' as in Remark 2 */ do
2     if $|\mu_{k,j}| > \frac{1}{2}$ then
3         $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$
4         $\mu_{k,j} \leftarrow \mu_{k,j} - \lfloor \mu_{k,j} \rceil$
5         for $i = 1, \ldots, j-1$ do
6             $\mu_{k,i} \leftarrow \mu_{k,i} - \lfloor \mu_{k,j} \rceil \mu_{j,i}$ /* As in Remark 1 part (3) */
7 return $\mathbf{B}'$ with size reduced $\mathbf{b}'_k$ and updated coefficients $\mu'_{i,j}$.
```

Size reduction. Given a basis $\mathbf{B}$ for a lattice $\mathcal{L}$, size reduction of $\mathbf{b}_i$ with $\mathbf{b}_j$ replaces $\mathbf{b}_i$ with the vector $\mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j$ while $\mathbf{b}_j$ remains unchanged. If $|\mu_{i,j}| < 1/2$, the vector $\mathbf{b}_i$ remains unchanged. Algorithm 1 describes the size reduction of a vector $\mathbf{b}_k$ with all its previous vectors $\mathbf{b}_{k-1}, \ldots, \mathbf{b}_1$ in the basis. The changes in the GSO coefficients $\mu_{i,j}$ due to size reduction have been described in Remark 1. Size reducing an entire basis $\mathbf{B}$ pertains to reducing each $\mathbf{b}_k$ for $2 \leq k \leq n$ with all previous vectors $\mathbf{b}_i, 1 \leq i < k$ in the ordering. The details are in Remark 2.

Definition 1 (Size reduced basis). *A basis* $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ *is said to be size reduced if for all* $1 \leq j < i \leq n$, $|\mu_{i,j}| \leq 1/2$.

Remark 1 (Changes in GSO Coefficients Upon Size Reduction). Based on the descriptions in [5, Chapter 2] and [10], we know that, upon a size reduction of $\mathbf{b}_i$ with $\mathbf{b}_j$ $(1 \leq i < j)$, the values of $\mu_{i,j}$ must be updated as follows for consistency.

1. We set $\mu_{i,j} \leftarrow \mu_{i,j} - \lfloor \mu_{i,j} \rceil$; as a result, upon reducing $\mathbf{b}_i$ with $\mathbf{b}_j$, we get $|\mu_{i,j}| \leq 1/2$.
2. For $j < l < i$, the values of $\mu_{i,l}$ remain unchanged. This is based on [5] and [10, Exercise 17.4.8 (3)]. The proof is as follows[2]. Let $\mathbf{b}_i$ be already size reduced with respect to the vectors $\mathbf{b}_{i-1}, \mathbf{b}_{i-2}, \ldots, \mathbf{b}_{j+1}$. So we have $\|\mu_{i,l}\| \leq 1/2$ for all l such that $j < l < i$. Now, we size reduce $\mathbf{b}_i$ with $\mathbf{b}_j$ to get $\mathbf{b}'_i = \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j$. Let $\mu'_{i,l}$ be the value of $\mu_{i,l}$ after the size reduction of $\mathbf{b}_i$ with respect to $\mathbf{b}_j$. Then we have
$$\mu'_{i,l} = \frac{\langle \mathbf{b}'_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|} - \lfloor \mu_{i,j} \rceil \frac{\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|}.$$
Note that since $l > j$, we have $\mathbf{b}_j \perp \mathbf{b}^*_l$ as $\mathbf{b}^*_l$ is (by definition) orthogonal to $\mathbf{b}_1, \ldots, \mathbf{b}_{l-1}$. Therefore, we have $\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle = 0$, and hence
$$\mu'_{i,l} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} - \lfloor \mu_{i,j} \rceil \frac{\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \mu_{i,l}.$$

[2] Although quite straight-forward, the proof is not detailed in the literature to the best of our knowledge.

3. For all $1 \le l \le j-1$, we set $\mu_{i,l} \leftarrow \mu_{i,l} - \lfloor \mu_{i,j} \rceil \mu_{j,l}$.

In summary, if we size reduce $\mathbf{b}_i$ with $\mathbf{b}_j$, then the values $\mu_{i,l}$ for $j < l \le i-1$ *do not change*. However, the values $\mu_{i,l}$ for $1 \le l \le j$ *may change*.

Remark 2 (Reducing in Reverse). Note that in Algorithm 1, while size reducing the vector $\mathbf{b}_k$ with $\mathbf{b}_1, \dots, \mathbf{b}_{k-1}$, we must reduce 'in reverse'. In other words, we first reduce $\mathbf{b}_k$ with $\mathbf{b}_{k-1}$, then $\mathbf{b}_{k-2}$, and so on, down to $\mathbf{b}_1$. This is for two reasons. First, as per point (2) of Remark 1, upon size reduction of $\mathbf{b}_k$ with $\mathbf{b}_i$, the vector $\mathbf{b}_k$ is still size reduced with respect to all $\mathbf{b}_l$ for $i < l < k$. Second, the size reduction of $\mathbf{b}_k$ with $\mathbf{b}_i$ for $1 \le i < k$ affects the size reducedness of $\mathbf{b}_k$ with respect to $\mathbf{b}_l$ for $l < i$ as per point (3) in Remark 1. So by size reducing $\mathbf{b}_k$ with $\mathbf{b}_i$, only the vectors before $\mathbf{b}_i$ are candidates for further size reduction of $\mathbf{b}_k$ and not the ones between $\mathbf{b}_i$ and $\mathbf{b}_k$.

Lattice Reduction. Given a basis for a lattice, the goal of lattice reduction is to transform it into a better quality basis consisting of shorter, more orthogonal vectors. Lattice reduction algorithms like LLL and its variants conduct size reduction as well as reordering of the input basis to improve their quality.

Basis Quality Measures. There are several measures that can be used to describe the quality of a basis. The most widely used is the Hermite factor (HF)

$$\gamma = \frac{\|\mathbf{b}_1\|}{\mathrm{Vol}(\mathcal{L})^{1/n}}$$

of a lattice. The vector $\mathbf{b}_1$ is assumed to be the shortest vector in the output basis. It has been shown that the smaller the Hermite factor of a basis, the better the basis quality [11]. Furthermore, the *root Hermite factor* (RHF) given by $\gamma^{1/n} = \left(\frac{\|\mathbf{b}_1\|}{\mathrm{Vol}(\mathcal{L})^{1/n}}\right)^{1/n}$ can be shown experimentally [11] to converge to a constant for certain basis reduction algorithms and large n.

The potential (Pot) of a basis $\mathbf{B}$ is defined in terms of its GSO vectors $\mathbf{B}^*$ as

$$\mathrm{Pot}(\mathbf{B}) = \prod_{i=1}^{n} \mathrm{Vol}(\mathcal{L}_i)^2 = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|^{2(n-i+1)}.$$

It was introduced in [18] to prove that LLL runs in polynomial time. The potential takes into account not only the vectors in a lattice basis but also their ordering. Earlier basis vectors have significantly more contribution to the value of $\mathrm{Pot}(\mathbf{B})$ than the later ones. We use the natural logarithm of the potential for easy handling of the large exponents in its computation, especially with large values of n.

$$\log_e(\mathrm{Pot}(\mathbf{B})) = \log_e\left(\prod_{i=1}^{n} \mathrm{Vol}(\mathcal{L}_i)^2\right) = 2\sum_{i=1}^{n}(n-i+1)\log_e(\|\mathbf{b}_i^*\|).$$

Another measure of basis quality is the squared sum (SS) of its GSO vectors $\mathbf{B}^*$ given by

$$\mathrm{SS}(\mathbf{B}) = \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2 .$$

Similarly to $\mathrm{Pot}(\cdot)$, the squared sum varies with changes in the lengths of the GSO vectors. However unlike $\mathrm{Pot}(\cdot)$, all GSO vectors contribute equally to its value.

Ordering of Basis Vectors. Let S_n be the group of permutations of the elements in [n]. For $\sigma \in S_n$ and a basis $\mathbf{B}$, we define $\sigma(\mathbf{B}) = \left(\mathbf{b}_{\sigma(1)}, \ldots, \mathbf{b}_{\sigma(n)}\right)$ to be a permutation of the basis vectors. Here, $\sigma(j)$ is the index of the vector in $\mathbf{B}$ that takes position j in the permuted basis $\sigma(\mathbf{B})$. In particular, we are interested in the permutations $\sigma_{i,k} \in S_n$ for $1 \leq i < k \leq n$ defined as follows.

$$\sigma_{i,k}(j) = \begin{cases} j & \text{if } \ j < i \text{ or } k < j \\ k & \text{if } \ j = i \\ j-1 & \text{if } \ i+1 \leq j \leq k. \end{cases}$$

Such a permutation of $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ essentially gives us the permuted basis

$$\sigma_{i,k}(\mathbf{B}) = (\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_i, \ldots, \mathbf{b}_{k-1}, \mathbf{b}_{k+1}, \ldots, \mathbf{b}_n)$$

where $\mathbf{b}_k$ is inserted between $\mathbf{b}_{i-1}$ and $\mathbf{b}_i$, and all vectors $\mathbf{b}_i, \ldots, \mathbf{b}_{k-1}$ are shifted up by one position. The other vectors retain their positions in the ordering.

Change in Basis Quality through Permutations. Let $X(\mathbf{B})$ be a measure of basis quality (like the HF, RHF, Pot, SS, etc.) of $\mathbf{B}$. On permuting the basis $\mathbf{B}$ to $\sigma_{i,k}(\mathbf{B})$, the difference in the measure is denoted as

$$\mathit{\Delta} X_{i,k} = X(\mathbf{B}) - X(\sigma_{i,k}(\mathbf{B})).$$

In particular, we get $\mathit{\Delta}\mathrm{Pot}_{i,k} = \mathrm{Pot}(\mathbf{B}) - \mathrm{Pot}(\sigma_{i,k}(\mathbf{B}))$ and $\mathit{\Delta}\mathrm{SS}_{i,k} = \mathrm{SS}(\mathbf{B}) - \mathrm{SS}(\sigma_{i,k}(\mathbf{B}))$[3]. We note that $\mathtt{argmax}_{1 \leq i < k \leq n}(\mathit{\Delta} X_{i,k})$ returns the pair of indices (i, k) for which the value of $\mathit{\Delta} X_{i,k}$ is maximised.

3 The LLL Algorithm, Its Variants and Generalisations

Given a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, we have its GSO $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$ and the coefficients $\mu_{i,j}$ therein.

[3] Note that even though the expression for computing the measure $\mathrm{SS}(\mathbf{B})$ itself gives equal weight to all GSO vectors (unlike $\mathrm{Pot}(\mathbf{B})$) independent of where they occur in the ordering of $\mathbf{B}^*$, the GSO vectors themselves (and hence their lengths) change upon reordering. As a result, the value of the measure $\mathrm{SS}(\mathbf{B})$ generally changes after reordering the basis.

Definition 2 (δ-LLL reduced basis). *Given* $1/4 < \delta \leq 1$, *a basis* $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ *is said to be* δ-LLL *reduced if the following two conditions are satisfied.*

1. $\mathbf{B}$ *is size reduced as in Definition 1.*
2. *For all* $2 \leq k \leq n$, *the Lovász condition holds between the consecutive vectors* $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$. *In other words,*

$$\delta \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2 .$$

Algorithm 2: The LLL Algorithm [18]

```
   Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, a threshold $1/4 < \delta \leq 1$
   Output: A basis $\mathbf{B'} = (\mathbf{b_1}', \ldots, \mathbf{b}'_n)$ which is $\delta$-LLL reduced
1  Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
2  $k \leftarrow 2$
3  while $k \leq n$ do
4      Size reduce $\mathbf{b}_k$ /* As in Algorithm 1 */
5      if $\|\mathbf{b}_k^*\|^2 < \left(\delta - \mu_{k,k-1}^2\right) \|\mathbf{b}_{k-1}^*\|^2$ /* Equivalent to the failure of the
        condition in (1) */ then
6          $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$ /* Swap vectors $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$ */
7          Update $\mathbf{b}_{k-1}^*, \mathbf{b}_k^*$ /* As in [10, Lemma 17.4.3] */
8          Update $\mu_{i,j}$'s /* As in [5, Algorithm 2.6.3] */
9          $k \leftarrow \max(k-1, 2)$
10     else
11         $k \leftarrow k + 1$
12 return $\mathbf{B'}$, a $\delta$-LLL reduced basis
```

Definition 3 (δ-DeepLLL reduced basis). *Given* $1/4 < \delta \leq 1$, *a basis* $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ *is said to be* δ-DeepLLL *reduced if the following two conditions are satisfied.*

1. $\mathbf{B}$ *is size reduced as in Definition 1.*
2. *For all* $1 \leq i < k \leq n$,

$$\delta \cdot \|\pi_i(\mathbf{b}_i)\|^2 \leq \|\pi_i(\mathbf{b}_k)\|^2 .$$

Remark 3. If the Lovász condition holds for *all* pairs (i, k), then it must certainly hold for all consecutive pairs $(k-1, k)$. A δ-DeepLLL reduced basis is hence δ-LLL reduced.

The LLL *and* DeepLLL *Algorithms.* The LLL algorithm [18] is described in Algorithm 2. The output basis $\mathbf{B'}$ is δ-LLL reduced as in Definition 2. A swap between vectors $\mathbf{b}_{k-1}$ and $\mathbf{b}_k$ in the algorithm is denoted by $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$.

This is generalised in DeepLLL [27] and its variants [8,31] to a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ where $1 \leq i < k \leq n$. All our descriptions are in terms of deep insertions. The corresponding results for swaps can be derived by substituting $i = k - 1$, where applicable.

Remark 4 (Measure for Lovász Condition). The Lovász condition is given by $\delta \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2$. This can be written as

$$(1-\delta) \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \geq \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 - \|\pi_{k-1}(\mathbf{b}_k)\|^2 .$$

Here, $\|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 - \|\pi_{k-1}(\mathbf{b}_k)\|^2$ denotes the change in $\|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 = \left\|\mathbf{b}^*_{k-1}\right\|^2$ (the square of the length of the $(k-1)^{th}$ GSO vector) that will occur if a swap step $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$ was to happen. In Algorithm 2, if the condition is not satisfied, then the change in $\left\|\mathbf{b}^*_{k-1}\right\|^2$ is large enough to go ahead with the swap and bring vector $\mathbf{b}_k$ to the earlier position $k-1$ in the basis ordering. In general, for $1 \leq i < k \leq n$, the Lovász condition for a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ is given by $\delta \cdot \|\pi_i(\mathbf{b}_i)\|^2 \leq \|\pi_i(\mathbf{b}_k)\|^2$ which can also be written similarly as

$$(1-\delta) \cdot \|\pi_i(\mathbf{b}_i)\|^2 \geq \|\pi_i(\mathbf{b}_i)\|^2 - \|\pi_i(\mathbf{b}_k)\|^2 .$$

Based on the above, we observe that *the (generalised) Lovász condition essentially uses a localised measure of the quality of the basis. For an index* $1 \leq i < n$, *the measure of quality of the basis* $\mathbf{B}$ *is given by* $\mathrm{LC}_i(\mathbf{B}) = \|\pi_i(\mathbf{b}_i)\|^2 = \|\mathbf{b}^*_i\|^2$. The change $\Delta\mathrm{LC}_i$ in the quality of the basis due to a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ is given by

$$\Delta\mathrm{LC}_i = \mathrm{LC}_i(\mathbf{B}) - \mathrm{LC}_i(\sigma_{i,k}(\mathbf{B})) = \|\pi_i(\mathbf{b}_i)\|^2 - \|\pi_i(\mathbf{b}_k)\|^2 .$$

Then, the generalised Lovász condition can be written as

$$(1-\delta) \cdot \mathrm{LC}_i(\mathbf{B}) \geq \Delta\mathrm{LC}_i \tag{1}$$

which fails if $\Delta\mathrm{LC}_i > (1-\delta) \cdot \mathrm{LC}_i(\mathbf{B})$ and calls for a deep insertion. This interpretation of the Lovász condition as a change in the measure of basis quality is not present in the literature to the best of our knowledge.

Thus the condition of the `if` statement in step 8 of Algorithm 3 is a further generalisation of the generalised Lovász condition for any measure of quality $X(\mathbf{B})$ of the basis $\mathbf{B}$. In Algorithm 3, if the condition is not satisfied, bringing a later vector $\mathbf{b}_k$ to an earlier position i in the basis ordering will result in appreciable improvement in the basis quality $X(\mathbf{B})$.

Variants of DeepLLL*: transition from a local measure to a global measure of quality* As noted above, the Lovász condition in LLL [18] and its generalisation in DeepLLL [27] are both used to check the decrease in $\|\pi_i(\mathbf{b}_i)\| = \|\mathbf{b}^*_i\|$ by inserting a later vector $\mathbf{b}_k$ at an earlier position $i < k$. The length of a GSO vector is a localised measure of quality that does not capture the quality of the whole basis. This changed in Pot-DeepLLL [8] where instead of a localised measure of quality, the potential $\mathrm{Pot}(\cdot)$ was used in DeepLLL so that the effect of permuting vectors on the entire basis is considered. In SS-DeepLLL [31], $\mathrm{Pot}(\cdot)$ was replaced by another global measure $\mathrm{SS}(\cdot)$. The basic operation of deep insertion for reordering the basis vectors is the same in all three algorithms.

Definition 4 (δ-X-DeepLLL reduced basis). *Given $0 < \delta \leq 1$, a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is said to be δ-X-*DeepLLL *reduced for a basis quality measure $X(\cdot)$ if the following two conditions are satisfied.*

1. *$\mathbf{B}$ is size reduced as in Definition 1.*
2. *For all $1 \leq i < k \leq n$,*

$$\delta \cdot X(\mathbf{B}) \leq X(\sigma_{i,k}(\mathbf{B})).$$

We omit the δ in naming our algorithms. Unless an algorithm is run for two different values of δ, this parameter is an implicit input to the algorithm. The choice of δ is however crucial in determining the quality of the basis. Larger the value of δ, the better the output quality in general. Hence we include it in the notation used in the definition of reducedness of a basis.

Using basis quality measures $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ in place of the generic $X(\cdot)$, Definition 4 is instantiated to that of a Pot-DeepLLL [8] reduced basis and a SS-DeepLLL [31] reduced basis. We know from [8, Lemma 2] that a Pot-DeepLLL reduced basis is LLL reduced. Also from [8, Lemma 3], for $1/4^{n-1} < \delta \leq 1$, a δ-DeepLLL reduced basis is δ^{n-1}-Pot-DeepLLL reduced. From [31, Proposition 1] we know that any 1-SS-DeepLLL reduced basis is also δ-LLL reduced for any $1/4 < \delta < 1$. However, there are no known relationships between δ-SS-DeepLLL reduced bases and δ-DeepLLL reduced bases to the best of our knowledge. We remark that both Pot-DeepLLL and SS-DeepLLL have polynomial-time complexity by construction, but their output quality cannot be covered by [31, Theorem 1] since their output bases are not DeepLLL-reduced.

Remark 5. In general, for two different basis quality measures X_1 and X_2, a δ-X_1-DeepLLL reduced basis may or may not be δ-X_2-DeepLLL reduced. In particular a δ-Pot-DeepLLL reduced basis is also δ-LLL reduced whilst a δ-SS-LLL reduced basis is not necessarily so for $\delta < 1$. Therefore, there exist bases which are δ-SS-LLL reduced but not necessarily δ-LLL reduced or δ-Pot-DeepLLL reduced.

A Generalisation of DeepLLL*,* Pot-DeepLLL *and* SS-DeepLLL*.* We provide a generalised description of DeepLLL and its variants Pot-DeepLLL and SS-DeepLLL in the X-DeepLLL algorithm 3. The X in the name X-DeepLLL corresponds to the general measure $X(\mathbf{B})$ of the quality of $\mathbf{B}$. The generalisation is instantiated for different local and global quality measures of a basis $\mathbf{B}$ that are all based on the GSO vectors $\mathbf{B}^*$. The localised measure $\mathrm{LC}_i(\mathbf{B}) = \|\mathbf{b}_i^*\|^2$ (used in DeepLLL [27]) is only for a single GSO basis vector, while the measures $\mathrm{Pot}(\cdot)$ [8] and $\mathrm{SS}(\cdot)$ [31] are on the entire GSO basis $\mathbf{B}^*$. In Remark 4 we have

Algorithm 3: The X-DeepLLL Algorithm

```
   Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, a threshold $0 < \delta \leq 1$
   Output: $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}'_n)$ which is $\delta$-$X$-DeepLLL reduced
 1 Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
 2 Size reduce $\mathbf{b}_2, \ldots, \mathbf{b}_n$ /* As in Algorithm 1 */
 3 Initialise other bookkeeping data structures, if required for $X(\mathbf{B})$
 4 $k \leftarrow 2$
 5 while $k \leq n$ do
 6     Size reduce $\mathbf{b}_k$ /* As in Algorithm 1 */
 7     Find $i$ such that $i = \mathtt{argmax}_{1 \leq j < k}(\Delta X_{j,k})$ and set $\Delta X = \Delta X_{i,k}$
 8     if $\Delta X > (1 - \delta) \cdot X(\mathbf{B})$ then
 9         $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ /* Deep insert $\mathbf{b}_k$ before $\mathbf{b}_i$ */
10         Update $\mathbf{B}^*$ and $\mu_{l,j}$ /* As in [30, Theorem 1 and Proposition 1] */
11         $k \leftarrow \max(i, 2)$
12     else
13         $k \leftarrow k + 1$
14 return $\mathbf{B}'$, a $\delta$-$X$-DeepLLL reduced basis
```

argued that the (generalised) Lovász condition can be interpreted as a condition on the change in the quality of the basis assessed based on the localised measure $\mathrm{LC}_i(\mathbf{B}) = \|\pi_i(\mathbf{b}_i)\|^2 = \|\mathbf{b}_i^*\|^2$. Hence, the X-DeepLLL algorithm 3 is a generalisation of the DeepLLL algorithm of [27]. Pot-DeepLLL and SS-DeepLLL are both variants of DeepLLL. It should be easy to see that the X-DeepLLL algorithm 3 for the measure $X(\cdot) = \mathrm{Pot}(\cdot)$ is Pot-DeepLLL and for the measure $X(\cdot) = \mathrm{SS}(\cdot)$, it is SS-DeepLLL.

In the generalised X-DeepLLL algorithm 3, we note that the threshold value δ represents a fraction of the measure $X(\mathbf{B})$. If a reordering of the basis $\mathbf{B}$ can improve its quality by *more than* $(1 - \delta) \cdot X(\mathbf{B})$, the algorithm has scope for such a reordering. In fact, when there is no way to decrease the measure (thus improving the quality) to less than the fraction $\delta \cdot X(\mathbf{B})$ of the measure, that is when the basis is considered to be δ-X-DeepLLL reduced as in Definition 3. As may be expected, the threshold δ depends on the measure X in the context. The notation δ is commonly used [18,27,8] to denote the fraction in the context of algorithms based on the localised measure $\mathrm{LC}(\cdot)$ (when using the Lovász condition) and the measure $\mathrm{Pot}(\cdot)$ for the whole basis. The notation η has been used in [31] to denote the threshold in the context of the measure $\mathrm{SS}(\cdot)$. In our generalisations of the algorithms and their analysis, we continue using the more common notation δ with the awareness that for two different measures $X_1(\cdot)$ and $X_2(\cdot)$, two different thresholds δ_1 and δ_2 may have to be considered, respectively. The relationship between threshold values δ_1, δ_2 of the algorithms may be derived from the relationship between their measures X_1, X_2 as in [8,31].

It should be clear that the value of the threshold δ in the X-DeepLLL algorithm 3 should be upper bounded by $\delta \leq 1$ (and consequently $(1 - \delta) \geq 0$) due to the algorithm's key principle of trying to reduce the measure $X(\mathbf{B})$ in every iteration as explained above. In particular, for $\delta = 1$, a deep insertion is allowed for any decrease $\Delta X > 0$ in the measure. Assuming the measure $X(\mathbf{B}) > 0$ for any basis $\mathbf{B}$, since the decrease in the measure ΔX can not be more than or equal to the measure $X(\mathbf{B})$ itself, hence we necessarily have $\delta > 0$. For the algorithms using the measure $\mathrm{LC}(\cdot)$ (based on the Lovász condition), the threshold must further satisfy $\delta > 0.25$ [18]. In general, the threshold δ and a tighter lower bound thereof may be determined by the termination condition for the loop.

All LLL-style algorithms work in a manner where a single iteration of the loop works only with a sublattice $\mathcal{L}_k$ generated by $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. The rest of the vectors $(\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n)$ remain "untouched" in that iteration. Hence, after a deep insertion step, $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ the sublattice under consideration in the next iteration would just be $(\mathbf{b}_1, \ldots, \mathbf{b}_i)$. The newly inserted vector $\mathbf{b}_i$ will have already been size reduced with respect to the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ in the previous iteration of the loop when considering the index k. The vectors $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$ have all been "shifted" up by one position. They will now require further size reduction since they will not have been reduced with respect to the newly inserted vector $\mathbf{b}_i$. However, this does not need to be done immediately; these vectors will be size reduced again when they enter the sublattice under consideration in a subsequent iteration of the loop. So these algorithms size reduce only one vector in an iteration and not the whole basis.

Deep Insertions. A deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ only changes the vectors $\mathbf{b}_i, \ldots, \mathbf{b}_k$ in the basis. The vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ remain unchanged. The corresponding changes in the GSO basis $\mathbf{B}^*$ and the lengths of the vectors therein is given by [30, Theorem 1]. The corresponding changes in the GSO coefficients is given by [30, Proposition 1].

Remark 6. From [30, Theorem 1], we note that due to a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ the only GSO vectors that change are $\mathbf{b}_i^*, \ldots, \mathbf{b}_k^*$. Hence, the only GSO coefficients that change are $\mu_{l,j}$ for $j < l$, $i \le j \le k$, and $i + 1 \le l \le n$.

Algorithm 4: The X-GGLLL Algorithm

```
   Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, a threshold $0 < \delta \le 1$
   Output: $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}_n')$ which is a $\delta$-$X$-GGLLL reduced basis
1  Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
2  Size reduce $\mathbf{b}_2, \ldots, \mathbf{b}_n$ in this order /* As in Algorithm 1 for each $\mathbf{b}_k$ */
3  Find $(i', k')$ such that $(i', k') = \mathtt{argmax}_{1 \le i < k \le n}(\Delta X_{i,k})$ and set $\Delta X = \Delta X_{i',k'}$
4  while $\Delta X > (1 - \delta) \cdot X(\mathbf{B})$ do
5      $\mathbf{B} \leftarrow \sigma_{i',k'}(\mathbf{B})$ /* Deep insert $\mathbf{b}_{k'}$ before $\mathbf{b}_{i'}$ */
6      Update $\mathbf{B}^*$ and $\mu_{l,j}$ /* As in [30, Theorem 1 and Proposition 1] */
7      Size reduce $\mathbf{b}_{i'+1}, \ldots, \mathbf{b}_n$ /* As in Algorithm 1 and proof of Lemma 1*/
8      Find $(i', k')$ such that $(i', k') = \mathtt{argmax}_{1 \le i < k \le n}(\Delta X_{i,k})$ and $\Delta X = \Delta X_{i',k'}$
9  end
10 return $\mathbf{B}'$, a $\delta$-$X$-DeepLLL reduced basis.
```

4 The X-GGLLL Algorithm

The generalisation of DeepLLL, Pot-DeepLLL and SS-DeepLLL in the form of the X-DeepLLL algorithm 3 sets the stage for our new framework of algorithms.

The Greedy Global Framework. The greedy global framework described as the X-GGLLL algorithm 4 provides a general description of algorithms realised by specifying a basis quality measure X. The algorithm starts by finding the GSO in step 1 and then size reducing the input basis $\mathbf{B}$ in step 2. In step 3, it finds a pair of indices (i', k') that may be suitable for a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$. It then runs a loop performing a deep insertion and associated bookkeeping in steps 5-6, the consequent size reductions using Algorithm 1 in step 7 and finding an appropriate pair (i', k') for the next iteration in step 8. By the end of each iteration of the loop, the algorithm produces a size reduced basis and the associated bookkeeping information like the values of $\mu_{i,j}$, etc. are all updated to be consistent with the new basis. The loop runs as long as there is a pair of indices (i', k') such that if $\mathbf{b}_{k'}$ is deep inserted before $\mathbf{b}_{i'}$, the change in the measure $\mathit{\Delta} X = X(\mathbf{B}) - X(\sigma_{i',k'}(\mathbf{B}))$ is at least a fraction $(1-\delta)$ of the current measure $X(\mathbf{B})$. Note that every time the loop runs, a deep insertion is certainly conducted. For δ close to 1, $(1-\delta)$ is a small value. So the algorithm essentially terminates when there is no possible deep insertion step in the entire basis $\mathbf{B}$ that can reduce the measure $X(\mathbf{B})$ by a fraction $(1-\delta)$ that may be considered as a substantial change to the quality of the basis. Thus X-GGLLL returns a δ-X-DeepLLL reduced basis as in Definition 4. We prove this in Lemma 2.

LLL, DeepLLL, Pot-DeepLLL and SS-DeepLLL all linearly increase or decrease the index $2 \le k \le n$ in an iteration. In the process, they only work with the sublattice $\mathcal{L}_k$ generated by a subset $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. The key novelty of our framework and the algorithms therefrom lies in not restricting the choice of the vector $\mathbf{b}_k$ that is investigated for a possible insertion at an earlier position to only a sublattice (unlike all previous LLL-style algorithms). Instead, our algorithm works with the whole basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ and hence the entire lattice in every iteration throughout the algorithm. As a result, a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ in our algorithm has to be immediately followed by reductions of $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_n$ to ensure that the entire basis is size reduced and ready for the next iteration. Even though this is $\mathcal{O}(n)$ more operations than that of X-DeepLLL, it creates avenues for smarter choices of the indices for size reduction. In asymptotic terms, this loss is compensated by the $\mathcal{O}(n)$ gain for not having to increment the index k for each deep insertion $\mathcal{O}(n)$ times in the worst case.

Apart from working with the whole basis in every iteration, we introduce a greedy technique to select the indices (i, k). In particular, the algorithm finds a pair (i', k') such that the consequent change in the measure $\mathit{\Delta} X(\mathbf{B})$ is maximised. Step 3 of Algorithm 4 does this for the first time before entering the loop and step 8 does it subsequently for each iteration of the loop. The algorithm starts with a certain value of $X(\mathbf{B})$ that can be at most I_X and attempts to reach a minimum value Z_X. By choosing the maximum decrease in each step, it gets closer to

Z_X by reaching a δ-X-DeepLLL state very quickly (if not the quickest[4]) by taking the largest possible leaps at each point. The asymptotic analysis assumes the least possible change in the measure in every iteration and hence does not capture the effect of the greedy choice. However, the gains due to the greedy choice gets reflected in the experimental results provided in Section 6 where our algorithms perform exceedingly well in terms of their runtimes, total number of deep insertions and total number of size reductions with respect to previous LLL-style algorithms.

The greedy choice is not necessarily the best long-term choice though. There could be other pairs (i, k) in an iteration that do not decrease the measure as much as the greedy choice (i', k') (but more than δ fraction) in that iteration, but creates the scope for larger decrease in the measure in subsequent iterations. We do not consider such strategies in this work and leave them for future considerations. Our focus is on the greedy choice only.

Every new measure gives us a unique new lattice reduction algorithm. For the potential, we get Pot-GGLLL and for the squared sum, we get SS-GGLLL. Like X-DeepLLL, the values of δ to be used to get output bases of sufficiently good quality, will depend on the measure X in the context. We assume that any other measure X will be calculable from the basis vectors and the GSO information. If necessary, the steps in Algorithm 4 can be modified to take into account possible additional bookkeeping steps that a measure may require if it is not calculable from the stored information. We note that the change in the measure X may require additional computation; for instance, the change in potential requires the calculation of projections[5]. However, these computations can be done on the fly, and are covered by step 8 of Algorithm 4.

Remark 7 (Pre-processing Reduction). The description of Pot-DeepLLL in [8, Algorithm 1] includes a pre-processing of the basis $\mathbf{B}$ by LLL. In case of the SS-DeepLLL algorithm in [31, Algorithm 2], the description itself does not include the pre-processing step. However, they have included the pre-processing step with 0.99-LLL while reporting the performance results [31, Section 4.3.3]. We note here from [31] that the quality of the output basis from a reduction algorithm is often key to their subsequent use in other algorithms for finding short vectors in the lattice. We believe that the pre-processing step is an interesting idea that is independent of our generalisations. In particular, any lattice reduction algorithm can be used for pre-processing the basis before being fed into a second algorithm for further reduction. Given that the efficiency of our algorithms (especially SS-GGLLL), are in practice almost as good as LLL in many

[4] It is well known that an immediate greedy choice is not necessarily always the best in terms of the overall result of an algorithm.

[5] In Pot-DeepLLL, when computing the position i for deep inserting a vector $\mathbf{b}_k$, it is necessary to compute the projections $\pi_i(\mathbf{b}_k)$ in order to check if the insertion is viable. However, this is not essential in the computation of the measure SS since the change in SS due to an insertion can be computed directly using the GSO information that was updated in a previous step without computing a projection [31, Equation 5].

cases, *while providing much better output quality*, we believe the pre-processing can be done using any algorithm that would be suitable in the context depending on an efficiency versus output quality trade-off for the given input parameters, basis types, etc. Hence, we have excluded the pre-processing step from our theoretical descriptions, asymptotic analysis and experiments of the LLL-style algorithms and have focused on their independent performances.

The BKZ algorithm [27] runs LLL as a preprocess before applying the blockwise reduction. We use the NTL library implementation of BKZ that inherits this feature in our experiments.

5 Theoretical Results

Lemma 1. *Let $X(\mathbf{B})$ be lower bounded by $Z_X > 0$. Algorithm 4, outputs a size reduced basis as in Definition 1.*

Proof. In every iteration of Algorithm 4, the measure decreases by $\Delta X(\mathbf{B}) = X(\mathbf{B}) - X(\sigma_{i,k}(\mathbf{B}))$. Since it can keep decreasing only until $Z_X > 0$, the algorithm terminates and outputs a basis.

To prove that the output basis is size reduced, it is sufficient to show that the basis vectors are all size reduced by the end of each iteration of the `while` loop in Algorithm 4. We prove this by induction on the number of loop iterations. We note that step 2 of Algorithm 4 size reduces the whole basis before the first iteration. In general, we assume that the basis is size reduced at the start of iteration r. By Remark 6, a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ only changes the GSO vectors $\mathbf{b}_i^*, \ldots, \mathbf{b}_k^*$. From [30, Theorem 1, Proposition 1] and point (2) of Remark 1, we know the following.

- *The vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ do not need further size reduction.* In fact, the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ have not changed. Since their orders have not changed either, their GSO vectors also remain the same.
- *The vector $\mathbf{b}_k$ upon being inserted in position i does not need further size reduction.* In the deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$, the vector $\mathbf{b}_k$ is inserted in position i. This vector has already been size reduced with respect to $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ in a previous iteration $< r$ (or before the loop starts). However, its GSO changes from $\pi_k(\mathbf{b}_k)$ to $\pi_i(\mathbf{b}_k)$ due to the reordering.
- *Vectors $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$ need to be size reduced by all earlier vectors, but $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ need only to be reduced by $\mathbf{b}_k, \ldots \mathbf{b}_1$.* By Remark 6, the only GSO coefficients that change upon a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ are $\mu_{l,j}$ for $j < l$, $i \leq j \leq k$, and $i+1 \leq l \leq n$.
  - In particular, for vectors $\mathbf{b}_l, i+1 \leq l \leq k$, the following things change.
    - ∗ The GSO of vector $\mathbf{b}_l$ changes from being a projection of $\mathbf{b}_l$ orthogonal to $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{l-1}))$ to being orthogonal to $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_{i+1}, \ldots, \mathbf{b}_{l-1}))$.
    - ∗ Also, $\mathbf{b}_l$ may not be size reduced with respect to this newly inserted vector $\mathbf{b}_k$.

Hence, we start with $\mathbf{b}_{i+1}$ and size reduce it with the newly inserted vector $\mathbf{b}_k$. Upon this size reduction, for $1 \le l < k$, the values of $\mu_{i+1,l}$ will be updated by part (3) of Remark 1. Hence, we must size reduce $\mathbf{b}_{i+1}$ with all vectors $\mathbf{b}_i, \ldots, \mathbf{b}_1$ as explained in Remark 2. We similarly reduce all vectors $\mathbf{b}_{i+2}, \ldots, \mathbf{b}_k$.

- Reordering the vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ does not change $\texttt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_k))$. The vectors $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ have not been changed due to the deep insertion step. Hence, their projections orthogonal to $\texttt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_k))$ remain the same. Thus their GSO remains the same. Furthermore, the vectors $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ may not be size reduced with respect to $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$. Therefore, vectors $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ must be size reduced only with the vectors $\mathbf{b}_k, \ldots, \mathbf{b}_1$ as in Remark 2.

We therefore must reduce $\mathbf{b}_{i'+1}, \ldots \mathbf{b}_n$, due to the change in GSO of the vector in position i'. This is done in step 7 of Algorithm 4.

□

Lemma 2. *Algorithm 4 returns a δ-X-*DeepLLL *reduced basis as in Definition 4.*

Proof. Algorithm 4 outputs a basis $\mathbf{B}'$. The condition in the `while` statement in step 4 of the algorithm ensures that upon termination of the algorithm, no possible reordering $\sigma_{i,k}(\mathbf{B}')$ for all $1 \le i < k \le n$ results in a $\Delta X = X(\mathbf{B}') - X(\sigma_{i,k}(\mathbf{B}'))$ which is greater than $(1-\delta) \cdot X(\mathbf{B}')$. In other words,

$$\Delta X = X(\mathbf{B}') - X(\sigma_{i,k}(\mathbf{B}')) \le (1-\delta) \cdot X(\mathbf{B}')$$

for all $1 \le i < k \le n$. Equivalently, $\delta \cdot X(\mathbf{B}') \le X(\sigma_{i,k}(\mathbf{B}'))$ for all $1 \le i < k \le n$. Also by Lemma 1, the basis $\mathbf{B}'$ on output is size reduced. Hence, the output of X-GGLLL is a δ-X-DeepLLL reduced basis as per Definition 4. □

In Algorithm 4, the basis quality measure $X(\mathbf{B})$ being a function of the basis $\mathbf{B}$, may be computed using the values of the associated parameters like $\|\mathbf{b}_i\|^2$, $\|\mathbf{b}_i^*\|^2$ and $\mu_{i,k}$, for all $1 \le i \le k \le n$. Let C be an upper bound on the square of the norm of the vectors in $\mathbf{B}$. The following result is on the computational complexity of the general X-GGLLL algorithm.

Lemma 3. *Let $\|\mathbf{b}_i\|^2 \le C$ for all $1 \le i \le n$ in a basis $\mathbf{B}$. In Algorithm 4[Step 8], let the number of bit operations required for finding the pair of indices (i', k') for the maximum $\Delta X_{i,k}$ be $\mathcal{O}(f_X(C, m, n))$ using exact $\mathbb{Q}$ arithmetic but without fast integer arithmetic. Let I_X and Z_X respectively denote upper and lower bounds on $X(\mathbf{B})$ and let $0 < \delta < 1$[6]. Then the total number of bit operations performed by the X-*GGLLL *algorithm 4 is given by*

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + f_X(C,m,n)\right) \log_{1/\delta}\left(\frac{I_X}{Z_X}\right)\right) \qquad (2)$$

[6] We note that Algorithm 4 can work with $\delta = 1$ because it can (theoretically) allow very small changes in the measure X. However, an arbitrarily small change in the measure cannot be captured by a fixed value of δ in the expression for the number of iterations. Hence, $\delta < 1$ in the analysis.

using exact $\mathbb{Q}$ arithmetic but without fast integer arithmetic.

Proof. We assume all arithmetic operations in Algorithm 4 are using exact $\mathbb{Q}$ arithmetic but without fast integer arithmetic. We first note that the size reduction step within the `while` loop ensures that length of the vectors in the basis $\mathbf{B}$ do not increase throughout the algorithm [18]. All arithmetic operations are on integers of size $O(n \log C)$ bits by the same argument as in [18][Proposition 1.26].

At each iteration of the `while` loop, we reduce the measure X by a factor of at least δ. So after i iterations, the measure $X^{(i)} = \frac{1}{\delta^i} X$ satisfies

$$Z_X \leq \frac{1}{\delta^i} X \leq I_X .$$

For a given δ, the iteration number i is maximised for $Z_X = \frac{1}{\delta^i} X$. Thus, the number of deep insertions (iterations of the `while` loop) is bounded above by

$$\log_{1/\delta} \left(\frac{I_X}{Z_X} \right) .$$

Within each iteration of the `while` loop, we do the following operations.

- *Deep insertions*: A deep insertion and its associated bookkeeping are done in steps 5-6 of the algorithm. This results in updates of the basis parameters like the GSO vectors and the GSO coefficients. We know from the analysis of [30, Algorithm 4] that the total bit-complexity of the GSO updates is $\mathcal{O}\left(n^4 \log^2 C\right)$.
- *Size reductions*: Step 7 of the algorithm size reduces the basis and performs the associated bookkeeping updates. A vector in the basis contains m integers each of size $\mathcal{O}(n \log C)$. So the size reduction of a vector $\mathbf{b}_k$ with $\mathbf{b}_i, 1 \leq i < k$ requires $\mathcal{O}\left(mn^2 \log^2 C\right)$ bit operations. So the size reduction of $\mathbf{b}_k$ with all such $\mathbf{b}_i$ as in Algorithm 1 requires $\mathcal{O}\left(mn^3 \log^2 C\right)$ bit operations. Hence size reducing all basis vectors will require $\mathcal{O}\left(mn^4 \log^2 C\right)$ bit operations.
- *Index search*: In step 8, we search for the pair of indices (i', k') for which the measure $\Delta X_{i',k'}$ is the minimum among all possible pairs. We assume this step requires $\mathcal{O}(f_X(C, m, n))$ bit operations in every iteration.

So Algorithm 4 needs a total of $\mathcal{O}\left(n^4 \log^2 C + mn^4 \log^2 C + f_X(C, m, n)\right)$ bit operations in each iteration of the `while` loop. Considering all iterations, the total number of bit operations performed by the algorithm is given by

$$\mathcal{O}\left(\left(\underbrace{n^4 \log^2 C}_{\text{deep insertion}} + \underbrace{mn^4 \log^2 C}_{\text{size reduction}} + \underbrace{f_X(C, m, n)}_{\text{index search}} \right) \underbrace{\log_{1/\delta} \left(\frac{I_X}{Z_X} \right)}_{\#\text{iterations}} \right) .$$

□

The proof of Lemma 3 shows that the asymptotic complexity of Algorithm 4 does not capture the *value* by which the measure X decreases in each iteration. Any deep insertion strategy which decreases X by a fraction at least $(1-\delta)$ will result in an algorithm with asymptotic complexity at most as in 2 of Lemma 3. In practice, the greedy choice of an insertion that results in the maximum possible decrease in the measure makes the algorithm very efficient.

We use Lemma 3 corresponding to the general framework to find the computational complexities of the concrete algorithms Pot-GGLLL and SS-GGLLL.

5.1 Computational Complexity of Pot-GGLLL

With $\mathrm{Vol}(\mathcal{L}_i)^2 = \prod_{j=1}^{i} \left\|\mathbf{b}_j^*\right\|^2$, the potential is given by

$$\mathrm{Pot}(\mathbf{B}) = \prod_{i=1}^{n} \mathrm{Vol}(\mathcal{L}_i)^2 = \prod_{i=1}^{n-1} \mathrm{Vol}(\mathcal{L}_i)^2 \cdot \mathrm{Vol}(\mathcal{L})^2.$$

From [8, Proof of Proposition 1] we know that an upper bound on the value of $\mathrm{Pot}(\mathbf{B})$ is

$$I_{\mathrm{Pot}} = \prod_{i=1}^{n-1} \mathrm{Vol}(\mathcal{L}_i)^2 \mathrm{Vol}(\mathcal{L})^2 \leq \prod_{i=1}^{n-1} C^i \mathrm{Vol}(\mathcal{L})^2 \leq \prod_{i=1}^{n-1} C^{\frac{n(n-1)}{2}} \mathrm{Vol}(\mathcal{L})^2$$

and a lower bound is $Z_{\mathrm{Pot}} \geq \mathrm{Vol}(\mathcal{L})^2$. In 2, we substitute the expressions for the number of iterations and the complexity of index search to find the overall complexity of Pot-GGLLL. From Lemma 3, the maximum number of iterations in Pot-GGLLL is

$$\log_{1/\delta}\left(\frac{I_{\mathrm{Pot}}}{Z_{\mathrm{Pot}}}\right) = \log_{1/\delta}\left(C^{n(n-1)/2}\right) = \mathcal{O}(n^2 \log_{1/\delta} C).$$

For a pair (i,k) of indices, computing the value of $\Delta\mathrm{Pot}_{i,k}$ as described in [8, Equation 3.1] requires $\mathcal{O}(n^2)$ arithmetic operations or equivalently $\mathcal{O}(n^4 \log^2 C)$ bit operations. A straight-forward extension of this to find $\texttt{argmax}_{1\leq i<k\leq n}(\Delta\mathrm{Pot}_{i,k})$ would require the computation of $\Delta\mathrm{Pot}_{i,k}$ for each pair (i,k) with a total of $\mathcal{O}(n^6 \log^2 C)$ bit operations. This computation can be improved by $\mathcal{O}(n)$ time. For a fixed index k, the values of $\Delta\mathrm{Pot}_{i,k}$ can be computed incrementally for all $i \in \{k-1,\ldots,1\}$ where $\Delta\mathrm{Pot}_{i-1,k}$ is computed using the value of $\Delta\mathrm{Pot}_{i,k}$. This optimisation is applicable to Pot-DeepLLL as well as Pot-GGLLL. Hence Pot-GGLLL computes $\texttt{argmax}_{1\leq i<k\leq n}(\Delta\mathrm{Pot}_{i,k})$ using $\mathcal{O}(n^5 \log^2 C)$ bit operations in each iteration. In total, Pot-GGLLL requires

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + n^5 \log^2 C\right) n^2 \log_{1/\delta} C\right) = \mathcal{O}\left((m+n)\,\frac{n^6 \log^3 C}{\log 1/\delta}\right)$$

bit operations. From [8][Proof of Proposition 1] we know that Pot-DeepLLL requires $\mathcal{O}\left((m+n)n^4 \log_{1/\delta} C\right)$ arithmetic operations or equivalently

$\mathcal{O}\left((m+n)\frac{n^6 \log^3 C}{\log 1/\delta}\right)$ bit operations, which is the same as Pot-GGLLL. The number of bit operations for each part of the Pot-DeepLLL algorithm has been listed in Table 1.

5.2 Computational Complexity of SS-GGLLL

An upper bound on the value of SS($\mathbf{B}$) is given by

$$I_{\text{SS}} = \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2 \leq n \cdot C$$

and a lower bound is $Z_{\text{SS}} \geq n$ which occurs when $C = 1$. From Lemma 3, the maximum number of iterations in SS-GGLLL is

$$\log_{1/\delta}\left(\frac{I_{\text{SS}}}{Z_{\text{SS}}}\right) = \log_{1/\delta}(C).$$

as was noted in [31, Proposition 2]. From [31, Equation 5], we know that $\varDelta\text{SS}_{i,k}$ can be computed in $\mathcal{O}(n^3 \log^2 C)$ bit operations. Using similar techniques as in Pot-GGLLL, the computation of $\texttt{argmax}_{1\leq i<k\leq n}(\varDelta\text{SS}_{i,k})$ requires $\mathcal{O}(n^4 \log^2 C)$ bit operations. Hence SS-GGLLL requires a total of

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + n^4 \log^2 C\right)\log_{1/\delta}(C)\right) = \mathcal{O}\left(\frac{mn^4 \log^3 C}{\log 1/\delta}\right)$$

bit operations. In comparison, the number of bit operations of SS-DeepLLL is

$$\mathcal{O}\left(\frac{mn^4 \log^3 C}{\log 1/\delta}\right)$$

which is again the same as SS-GGLLL. The number of bit operations for each part of the SS-DeepLLL algorithm has been listed in Table 1.

Algorithm Name	Deep Insertion	Size Reduction	Index Search	Number of Iterations
Pot-DeepLLL	$mn^3 \log^2 C$	$mn^3 \log^2 C$	$n^4 \log^2 C$	$n^3 \log_{1/\delta} C$
Pot-GGLLL	$n^4 \log^2 C$	$mn^4 \log^2 C$	$n^5 \log^2 C$	$n^2 \log_{1/\delta} C$
SS-DeepLLL	$mn^3 \log^2 C$	$mn^3 \log^2 C$	$n^3 \log^2 C$	$n \log_{1/\delta} C$
SS-GGLLL	$n^4 \log^2 C$	$mn^4 \log^2 C$	$n^4 \log^2 C$	$\log_{1/\delta} C$

Table 1: Complexity comparison of X-DeepLLL and X-GGLLL.

*Remark 8 (Comparison between X-*DeepLLL *and X-*GGLLL*).* A comparison between the number of bit operations required in different parts of the X-DeepLLL and X-GGLLL algorithms is shown in Table 1. It provides a better understanding of where a greedy global algorithm makes gains and losses when compared with the corresponding DeepLLL algorithm. For deep insertion, X-DeepLLL requires $\mathcal{O}(mn^3 \log^2 C)$ bit operations. This involves a reordering of the basis followed by an update of the relevant GSO information. In comparison, X-GGLLL requires $\mathcal{O}(n^4 \log^2 C)$ bit operations using [30, Algorithm 4]. For size reductions, X-DeepLLL requires $\mathcal{O}(n)$ fewer bit operations than X-GGLLL because the greedy global algorithms need the basis to be completely size reduced before performing the index search. In contrast, X-DeepLLL needs the basis to be size reduced only up to index k being considered in an iteration. X-DeepLLL also requires $\mathcal{O}(n)$ fewer bit operations for index search than X-GGLLL. In X-DeepLLL, the index k is fixed, and so only a search for the best index i for insertion is required. However, for X-GGLLL, the search covers all pairs (i, k) for $1 \leq i < k \leq n$, and so $\mathcal{O}(n)$ more operations are required. The increase in complexity due to index search is compensated in the number of iterations of the `while` loop that requires $\mathcal{O}(n)$ fewer operations in X-GGLLL than in X-DeepLLL. This is because X-DeepLLL maintains the index k which must reach $k = n + 1$ for the algorithm to terminate. If N is the number of deep insertions in X-DeepLLL, the number of times k is incremented in step 13 of Algorithm 3 is upper bounded by $N(n-1)+n$ as argued in [18]. In other words, there are at most $\mathcal{O}(n)$ more iterations of the `while` loop than the number of deep insertions. Since there is no such incremental change in the indices in X-GGLLL, hence it requires $\mathcal{O}(n)$ fewer iterations.

Remark 9. The output basis of X-GGLLL is δ-X-DeepLLL reduced just like in X-DeepLLL. In practice, the output basis of X-GGLLL is usually better than X-DeepLLL. For example, for the first basis we tested at dimension 40, the values of RHF are 1.0127 for SS-GGLLL and 1.0151 for SS-DeepLLL. However, X-GGLLL is not necessarily guaranteed to reduce the basis quality measure X more than X-DeepLLL. For instance, for the second basis we tested at dimension 40, the values of RHF are 1.0151 for SS-DeepLLL and 1.0155 for SS-GGLLL.

6 Experimental Results

We conduct concrete comparative analysis of a number of relevant LLL-style algorithms including LLL [18], Pot-DeepLLL [8] and SS-DeepLLL [31]; the BKZ algorithm [27] with blocksizes $8, 10, 12$ and 20; and our two new proposals Pot-GGLLL and SS-GGLLL. For LLL, Pot-DeepLLL and Pot-GGLLL we use the threshold value $\delta = 0.999$. For SS-DeepLLL and SS-GGLLL we use the threshold $\delta = (1 - 10^{-6})$ following the rationale provided in the discussion in [31, Section 4.3.1].

To the best of our knowledge, there is no publicly available implementation of SS-DeepLLL [31, Algorithm 2]. Hence, for the sake of uniformity and fairness, we have used our own implementations of all LLL-style algorithms using floating point arithmetic. We have used the NTL library datatypes `ZZ` for integers and `RR` for real numbers. For each dimension we fixed a precision which is a deliberate overestimate so that we did not encounter anomalies due to floating-point arithmetic. We ran all algorithms with the same precision at a given dimension. We have used the `BKZ_RR` algorithm of NTL [29] in our comparisons. We do not apply any preprocessing to our implementations of the LLL-style algorithms. However, the BKZ implementation in the NTL library [29] includes preprocessing with LLL. Our implementations, the input lattice bases we have used in our experiments and the outputs of our algorithms are available at [12].

Each algorithm ran on a single Intel® Xeon® CPU E7-4830 v2 at 2.20 GHz on a shared memory machine. Our input bases are random in the sense of Goldstein and Mayer [13] and are akin to those provided by the SVP Challenge [6]. These bases have the form

$$\mathbf{B} = \begin{bmatrix} q & \mathbf{0} \\ \mathbf{x} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} q & 0 & 0 & \dots & 0 \\ x_1 & 1 & 0 & \dots & 0 \\ x_2 & 0 & 1 & \dots & 0 \\ \vdots & & & \ddots & \\ x_{n-1} & 0 & \dots & 0 & 1 \end{bmatrix}$$

where q is a $10n$-bit prime, $\mathbf{x} = (x_1, \dots, x_{n-1})^T$ is a column vector of integers modulo q chosen uniformly at random and $\mathbf{I}$ is the $(n-1) \times (n-1)$ identity matrix. For dimensions $n = 40, 50, 60, 70, 80$ and 90, we tested the algorithms by generating 300 such bases and for dimensions $n = 100, 110, 120, 130, 140$ and 150, we tested on 50 bases.

Our comparisons of the aforementioned algorithms are based on three efficiency parameters, namely, (1) average running time (Table 2 and Figure 1), (2) number of reorderings (swaps in LLL and deep insertions in the rest; Table 3 and Figure 2), and (3) number of size reductions of the basis vectors as in step 3 of Algorithm 1 (Table 4 and Figure 3). We measure the output quality using the root Hermite factor (RHF) (Table 5 and Figure 4).

	Dimension											
Algorithm	40	50	60	70	80	90	100	110	120	130	140	150
LLL	2.03	5.74	12.6	22.7	41.9	85.0	148	248	348	580	831	1134
SS-DeepLLL	8.54	25.3	63.2	124	244	490	954	1777	2479	4103	6268	8753
SS-GGLLL	2.20	6.06	15.1	31.1	63.2	120	275	497	818	1695	2554	3873
Pot-DeepLLL	14.2	48.1	136	295	645	1323	3022	5644	8918	15929	25617	37328
Pot-GGLLL	6.37	20.7	59.1	134	301	649	1555	3043	5130	9865	16886	26739
BKZ-08	6.62	19.4	47.9	93.9	187	370	775	1492	2141	3730	5889	8341
BKZ-10	6.85	20.2	50.8	100.0	202	417	829	1630	2393	4205	6804	9535
BKZ-12	7.20	21.6	54.8	109	221	443	941	1861	2655	4746	7683	11014
BKZ-20	9.23	30.4	90.4	201	458	1069	2645	5422	9742	17614	32865	57624

Table 2: Average runtime in seconds (rounded to most significant 3 digits for smaller values).

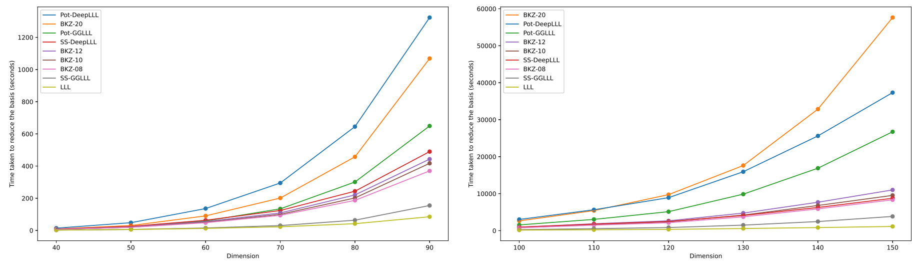

Fig. 1: Average runtime in seconds.

Comparison between LLL, X*-*DeepLLL *and* X*-*GGLLL*.* We first compare the LLL-style algorithms. LLL is, as expected, the quickest in every dimension. However, again as one would expect, it produces the worst quality in terms of average RHF in every dimension.

While the asymptotic runtime complexities of our greedy global algorithms Pot-GGLLL and SS-GGLLL are the same as the corresponding X-DeepLLL algorithms, we see from Table 2 (Figure 1) that *our algorithms run in much less time on average in every dimension. As the dimension grows, they become even better in comparison.* At dimension 150, SS-GGLLL is around 2.27 times faster than SS-DeepLLL and Pot-GGLLL is about 1.40 times faster than Pot-DeepLLL. In fact, *the average runtime of* SS-GGLLL *is only second to* LLL *among all algorithms under consideration, consistently in all tested dimensions.* Compared with LLL, at dimension 40, SS-GGLLL is just 1.09 times slower, and at dimension 150, SS-GGLLL is around 3.39 times slower. However Pot-GGLLL does not compare as well with LLL in terms of time. At dimension 40, Pot-GGLLL is roughly 3.14 times slower and is around 23.6 times slower at dimension 150.

	Dimension							
Algorithm	40	50	60	80	90	100	130	150
LLL	27355	47667	73652	142102	184177	231595	400127	535176
SS-DeepLLL	12512	23996	39682	83783	112048	144138	258387	346082
SS-GGLLL	362	517	705	1206	1525	1938	3626	5442
Pot-DeepLLL	11891	22251	36332	75783	101189	130547	239663	329958
Pot-GGLLL	300	429	571	914	1128	1364	2186	2902

Table 3: Average number of reorderings (swaps and deep insertions).

The number of reorderings of basis vectors (swaps and deep insertions) is crucial to the runtime analysis of our X-GGLLL algorithms. It is a strong indicator of the reason why our algorithms do so well in practice, even though their asymptotic behaviour is similar to their DeepLLL counterparts as shown

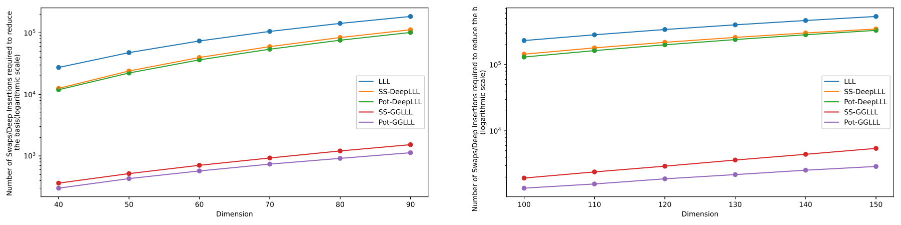

Fig. 2: Average number of reorderings.

in Table 1. The *average number of reorderings* in the LLL-style algorithms is provided in Table 3 (Figure 2). At dimension 40, the number in SS-GGLLL is only around 1.32% of LLL and around 2.90% of SS-DeepLLL. Pot-GGLLL does even better. The number in Pot-GGLLL at dimension 40 is only 1.10% of LLL and 2.52% of Pot-DeepLLL. At dimension 150, the number in SS-GGLLL is around 1.02% of LLL and around 1.57% of SS-DeepLLL. The number in Pot-GGLLL at dimension 150 is only 0.54% of LLL and 0.88% of Pot-DeepLLL.

Remark 10. The comparisons of the number of reorderings in LLL-style algorithms provide strong intuitive justification for our greedy global approach in terms of improving the efficiency. One would expect that fewer reorderings of the basis (and hence fewer GSO updates and size reductions) would result in a more efficient algorithm. However, we must note that upon a reordering in the X-GGLLL algorithm, there is more that needs to be done compared to LLL or X-DeepLLL to ensure that the basis is fully size reduced for the next iteration. Hence, recording the number of size reductions of $\mathbf{b}_k$ with $\mathbf{b}_j$ (Algorithm 1[Step 3]) provides a more granular measure of efficiency for fairer comparison between the LLL-style algorithms.

	Dimension							
Algorithm	40	50	60	80	90	100	130	150
LLL	81180	185909	359996	992472	1485987	2123061	4982003	7815673
SS-DeepLLL	143755	371094	790779	2522662	4010701	6044032	16242772	27623812
SS-GGLLL	70705	164548	337483	1106309	1827259	2903142	9523874	18822813
Pot-DeepLLL	127230	320840	669874	2052506	3199109	4732811	12083865	19710300
Pot-GGLLL	67410	155502	312213	963137	1541376	2353368	6759215	12158151

Table 4: Average number of size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ (Algorithm 1[Step 3]).

Table 4 provides the *average number of size reductions* (Algorithm 1[Step 3]) for each LLL-style algorithm in some of the representative dimensions. Pot-GGLLL requires fewer size reductions on average than SS-GGLLL in all dimensions in our tests. Also, the greedy global variants always perform fewer

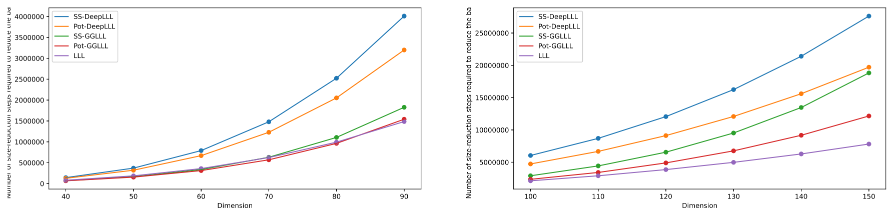

Fig. 3: Average number of size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ (Algorithm 1[Step 3]).

size reductions than their DeepLLL counterparts. At dimension 150, the number in SS-GGLLL is 68.1% of SS-DeepLLL and the number in Pot-GGLLL is 61.7% of Pot-DeepLLL. The X-GGLLL algorithms require smaller number of size reductions than LLL in smaller dimensions. At dimension 40, the number in Pot-GGLLL is around 83.0% of LLL while that of SS-GGLLL is around 87.1% of LLL. As the dimension increases, the increase in the number of size reductions in LLL is lesser than the other algorithms. At dimension 150, the number in SS-GGLLL is 2.41 times of LLL and the number in Pot-GGLLL is 1.56 times of LLL. A graphical representation of the average number of size reductions in Figure 3 shows the diverging curves of X-GGLLL and the respective X-DeepLLL counterparts showing that *the difference in the number keeps growing as the dimension increases.*

	Dimension						
Algorithm	40	60	80	90	100	130	150
LLL	1.01664	1.01829	1.01938	1.01955	1.01957	1.02036	1.02049
SS-DeepLLL	1.01366	1.01392	1.01408	1.01403	1.01416	1.01409	1.01405
SS-GGLLL	1.01335	1.01377	1.01375	1.01374	1.01382	1.01366	1.01368
Pot-DeepLLL	1.01372	1.01430	1.01456	1.01468	1.01472	1.01503	1.01506
Pot-GGLLL	1.01349	1.01405	1.01437	1.01448	1.01453	1.01476	1.01491
BKZ-08	1.01324	1.01399	1.01427	1.01440	1.01454	1.01462	1.01475
BKZ-10	1.01315	1.01367	1.01383	1.01395	1.01403	1.01411	1.01422
BKZ-12	1.01290	1.01327	1.01344	1.01351	1.01359	1.01376	1.01386
BKZ-20	1.01242	1.01228	1.01235	1.01237	1.01231	1.01246	1.01248

Table 5: Average Root Hermite Factor (RHF).

In addition to having better runtime, the X-GGLLL algorithms achieve a smaller *average RHF* than their corresponding X-DeepLLL algorithms across the tested dimensions, as shown in Table 5 (Figure 4). At dimension 150, SS-GGLLL outperforms SS-DeepLLL achieving average RHFs 1.01368 and 1.01405 respectively. Similarly, Pot-GGLLL and Pot-DeepLLL achieve average RHFs 1.01491 and 1.01506 respectively. To summarise,

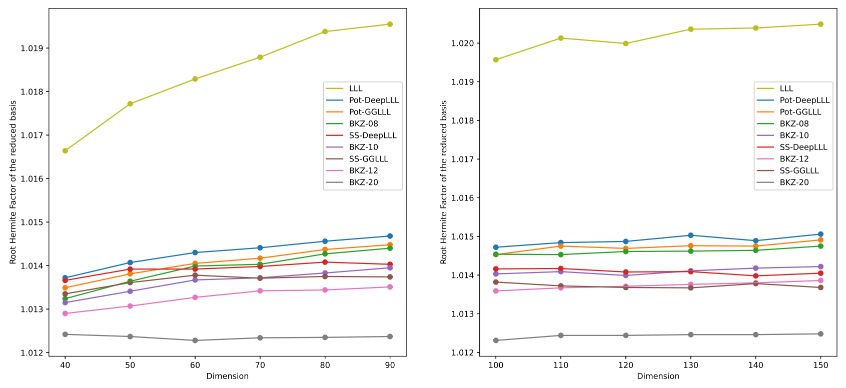

Fig. 4: Average Root Hermite Factor (RHF).

> X-GGLLL *on the average outperforms* X-DeepLLL *both in terms of efficiency and output quality*

whilst achieving the same theoretical notion of reduction as proved in Lemma 2 (by producing δ-X-DeepLLL reduced bases) and the same asymptotic runtime complexity as explained in Section 5.1 and Section 5.2.

Comparing SS-GGLLL *and* BKZ*.* SS-GGLLL clearly outperforms Pot-GGLLL both in terms of efficiency and output quality throughout our tests. So we next compare SS-GGLLL and BKZ with blocksizes $8, 10, 12$ and 20.

We first note that SS-GGLLL *is unilaterally faster than* BKZ *for all* 4 *blocksizes and all dimensions in our tests.* At dimension 150, BKZ-8, 10, 12 and 20 took around 2.15, 2.46, 2.84 and 14.9 times longer than SS-GGLLL respectively. It is clear from the diverging curves in Figure 1 that BKZ *becomes slower compared to* SS-GGLLL *as the dimension increases.* In particular, the factor by which the runtime of BKZ-20 grows relative to SS-GGLLL is also increasing.

While SS-GGLLL starts below BKZ-8, 10 and 12 in terms of its output quality at dimension 40, it eventually outperforms at higher dimensions. In Figure 5, the ratios of the average RHFs of SS-GGLLL and BKZ-$8, 10, 12$ and 20, are provided for all dimensions tested. A value above the dotted line at $y = 1$ implies that the RHF of SS-GGLLL is smaller, whereas a value below it implies that the RHF of BKZ is smaller. SS-GGLLL *starts outperforming* BKZ-8 *before dimension* 50*,* BKZ-10 *before dimension* 70 *and* BKZ-12 *before dimension* 120. When one also considers that SS-GGLLL is significantly faster than BKZ (especially at higher dimensions), it is clear that SS-GGLLL *is indeed an improvement on* BKZ for blocksizes $8, 10$ and 12.

One can notice from Figure 5 that the RHF of SS-GGLLL keeps getting better than BKZ-8, 10 and 12 with increasing dimension and the gap between them continues to widen. From Figure 4, we also notice that the RHFs for BKZ-8, 10 and 12 are generally increasing with the dimension, while that of SS-GGLLL is generally decreasing. This explains the reason behind SS-GGLLL eventually outperforming BKZ-8, 10 and 12 in Figure 5.

Even though BKZ-20 is much slower than SS-GGLLL, it provides better output quality across all dimensions that we tested. However, the behaviour of BKZ-20 in larger dimensions may be expected to emulate that of BKZ-8, 10 and 12 with respect to SS-GGLLL. It seems that the runtime of SS-GGLLL will continue to improve when compared with BKZ-20, whilst the gap in quality slowly decreases. It is hard to say how narrow the gap in RHF between SS-GGLLL and BKZ-20 will become in higher dimensions due to the subtle changes that can be noticed thus far. Whilst there is a possibility that the RHF of SS-GGLLL eventually gets better than BKZ-20, that may not happen as well.

Our focus in this work has been the core performance of algorithms and its analysis rather than their specific applications and use in engineering solutions like preprocessing. Hence we compare SS-GGLLL directly with BKZ. We note that SS-GGLLL can be used to replace LLL as a preprocessing step in BKZ. While preprocessing with SS-GGLLL will run longer than LLL, it will provide a far better quality basis for BKZ to begin with, which may in turn reduce the runtime of BKZ.

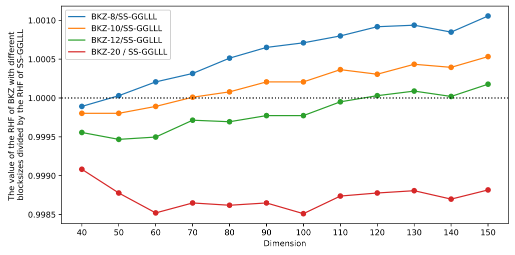

Fig. 5: The gap between the RHF of SS-GGLLL and BKZ with blocksize $\beta = 8$, 10, 12 and 20, expressed as $\frac{\mathrm{RHF}_{\text{BKZ-}\beta}}{\mathrm{RHF}_{\text{SS-GGLLL}}}$ (which is $= 1$ at the dotted line).

7 Conclusion

In this work, we have presented a greedy global framework as a generic algorithm X-GGLLL based on a measure of quality X of a lattice basis. The key novelty in is framework is to work with the whole lattice in every iteration in place of a sublattice as in all previous LLL-style algorithms. The basis vectors are reordered using a greedy approach towards improving their quality that results in very efficient algorithms. We have proved results on the efficiency of the general framework and on the two new algorithms we propose using the basis quality measures - potential Pot and squared sum SS. Furthermore, we have shown that the bases produced by our algorithms are of provable quality. Experimentally we have found that our algorithms do very well in terms of both efficiency and basis quality when compared to their counterparts introduced in [8] and [31] respectively. Our squared-sum based algorithm SS-GGLLL is second only to

LLL in terms of efficiency while producing output bases with quality that gets generally better than BKZ of increasing blocksize as the dimension increases.

Our design principle has been to achieve the best possible efficiency in reaching an assured quality by reducing the measure X as much as possible in each iteration. The result is quick improvements in the basis quality. Our framework could be altered to not make the most greedy choice resulting in a slower algorithm which performs more iterations to help close the gap in quality with BKZ (say with blocksize 20) at smaller dimensions. There could be other strategies that may as well decrease the overall runtime without compromising on the quality or even improving it, as X-GGLLL did over X-DeepLLL. We believe our framework has opened up avenues for designing interesting new lattice reduction algorithms.

Acknowledgement. We thank the anonymous reviewers for their detailed comments on an earlier version of this paper that helped in improving it significantly. We also thank Palash Sarkar for his comments on the paper.

References

1. Akhavi, A.: The optimal LLL algorithm is still polynomial in fixed dimension. Theoretical Computer Science **297**(1), 3–23 (2003). https://doi.org/10.1016/S0304-3975(02)00616-3
2. Bogart, T., Goodrick, J., Woods, K.: A Parametric Version of LLL and Some Consequences: Parametric Shortest and Closest Vector Problems. SIAM J. Discret. Math. **34**(4), 2363–2387 (jan 2020). https://doi.org/10.1137/20M1327422
3. Chang, X.W., Stehlé, D., Villard, G.: Perturbation analysis of the QR factor R in the context of LLL lattice basis reduction. Mathematics of Computation **81**(279), 1487–1511 (2012)
4. Chen, J., Stehlé, D., Villard, G.: Computing an LLL-reduced basis of the orthogonal lattice. In: Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation. p. 127–133. ISSAC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3208976.3209013
5. Cohen, H.: A course in computational algebraic number theory. Springer (1993)
6. Darmstadt T.U.: SVP challenge. https://www.latticechallenge.org/svp-challenge/
7. Debris-Alazard, T., Ducas, L., van Woerden, W.P.J.: An algorithmic reduction theory for binary codes: LLL and more. IEEE Transactions on Information Theory **68**(5), 3426–3444 (2022). https://doi.org/10.1109/TIT.2022.3143620
8. Fontein, F., Schneider, M., Wagner, U.: PotLLL: a polynomial time version of LLL with deep insertions. Designs, Codes and Cryptography **73**(2), 355–368 (2014). https://doi.org/10.1007/s10623-014-9918-8
9. Fukase, M., Kashiwabara, K.: An accelerated algorithm for solving svp based on statistical analysis. J. Inf. Process. **23**, 67–80 (2015)
10. Galbraith, S.D.: Mathematics of public key cryptography. Cambridge University Press, USA, 1st edn. (2012)
11. Gama, N., Nguyen, P.Q.: Predicting lattice reduction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 31–51. Springer, Istanbul, Turkey (apr 2008). https://doi.org/10.1007/978-3-540-78967-3_3

12. GGLLL: Our implementations of all lll-style algorithms (2023), https://github.com/GG-LLL/Greedy-Global-LLL
13. Goldstein, D., Mayer, A.: On the equidistribution of hecke points. Forum Mathematicum **15**(2), 165–189 (2003). https://doi.org/10.1515/form.2003.009
14. Howgrave-Graham, N.A.: Isodual reduction of lattices. Cryptology ePrint Archive, Paper 2007/105 (2007), https://eprint.iacr.org/2007/105
15. Kirchner, P., Espitau, T., Fouque, P.A.: Towards faster polynomial-time lattice reduction. In: Malkin, T., Peikert, C. (eds.) 2021, Part II. LNCS, vol. 12826, pp. 760–790. Springer, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84245-1_26
16. Koy, H., Schnorr, C.P.: Segment LLL-reduction of lattice bases. In: Silverman, J.H. (ed.) Cryptography and Lattices. pp. 67–80. Springer Berlin Heidelberg (2001). https://doi.org/10.1007/3-540-44670-2_7
17. Lee, C., Pellet-Mary, A., Stehlé, D., Wallet, A.: An LLL algorithm for module lattices. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019, Part II. LNCS, vol. 11922, pp. 59–90. Springer, Kobe, Japan (Dec 8–12, 2019). https://doi.org/10.1007/978-3-030-34621-8_3
18. Lenstra, A.K., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**, 515–534 (1982). https://doi.org/10.1007/BF01457454
19. Lenstra, H.W.: Flags and lattice basis reduction. In: Casacuberta, C., Miró-Roig, R.M., Verdera, J., Xambó-Descamps, S. (eds.) European Congress of Mathematics. pp. 37–51. Birkhäuser Basel, Basel (2001)
20. Morel, I., Stehlé, D., Villard, G.: H-LLL: using householder inside LLL. In: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation. p. 271–278. ISSAC '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1576702.1576740
21. Neumaier, A., Stehlé, D.: Faster LLL-type reduction of lattice bases. In: Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation. p. 373–380. ISSAC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2930889.2930917
22. Nguyen, P.Q., Stehlé, D.: Floating-point LLL revisited. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 215–233. Springer, Aarhus, Denmark (May 22–26, 2005). https://doi.org/10.1007/11426639_13
23. Nguyen, P.Q., Stehlé, D.: LLL on the average. In: Proceedings of the 7th International Conference on Algorithmic Number Theory. p. 238–256. ANTS'06, Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11792086_18
24. Nguyen, P.Q., Stern, J.: The two faces of lattices in cryptology. In: Silverman, J.H. (ed.) Cryptography and Lattices. pp. 146–180. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44670-2_12
25. Plantard, T., Susilo, W., Zhang, Z.: LLL for ideal lattices: re-evaluation of the security of Gentry—Halevi's FHE scheme. Des. Codes Cryptography **76**(2), 325–344 (aug 2015). https://doi.org/10.1007/s10623-014-9957-1
26. Schneider, M., Buchmann, J., Lindner, R.: Probabilistic analysis of LLL reduced bases. In: in Proc. WEWoRC 2009 (2010)
27. Schnorr, C.P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Mathematical programming **66**(1), 181–199 (1994). https://doi.org/10.1007/BF01581144
28. Schnorr, C.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science **53**(2), 201–224 (1987). https://doi.org/10.1016/0304-3975(87)90064-8

29. Shoup, V.: NTL: a library for doing number theory (2021), available at https://github.com/libntl/ntl
30. Yamaguchi, J., Yasuda, M.: Explicit formula for Gram-Schmidt vectors in LLL with deep insertions and its applications. In: Kaczorowski, J., Pieprzyk, J., Pomykała, J. (eds.) Number-Theoretic Methods in Cryptology. pp. 142–160. Springer (2018). https://doi.org/10.1007/978-3-319-76620-1_9
31. Yasuda, M., Yamaguchi, J.: A new polynomial-time variant of LLL with deep insertions for decreasing the squared-sum of Gram–Schmidt lengths. Designs, Codes and Cryptography **87**(11), 2489–2505 (2019). https://doi.org/10.1007/s10623-019-00634-9
32. Yasuda, M., Yokoyama, K., Shimoyama, T., Kogure, J., Koshiba, T.: Analysis of decreasing squared-sum of gram-schmidt lengths for short lattice vectors. Journal of Mathematical Cryptology **11**(1), 1–24 (2017). https://doi.org/10.1515/jmc-2016-0008