A Greedy Global Framework for Lattice Reduction Using Deep Insertions

Sanjay Bhattacherjee[1], Julio Hernandez-Castro[2], and Jack Moyler[1]

[1] Institute of Cyber Security for Society and School of Computing, University of Kent, Canterbury, Kent, United Kingdom
{s.bhattacherjee,jdm58}@kent.ac.uk
[2] ETSISI, Universidad Politecnica de Madrid, Madrid, Spain
jc.hernandez.castro@upm.es

Abstract. LLL-style lattice reduction algorithms iteratively employ size reduction and reordering on ordered basis vectors to find progressively shorter, more orthogonal vectors. DeepLLL reorders the basis through deep insertions, yielding much shorter vectors than LLL. DeepLLL was introduced alongside BKZ, however, the latter has received greater attention and has emerged as the state-of-the-art. We first show that LLL-style algorithms work with a designated measure of basis quality and iteratively improves it; specifically, DeepLLL improves a sublattice measure based on the generalised Lovász condition. We then introduce a new generic framework X-GG for lattice reduction algorithms that work with a measure X of basis quality. X-GG globally searches for deep insertions that minimise X in each iteration. We instantiate the framework with two quality measures – basis potential (Pot) and squared sum (SS) – both of which have corresponding DeepLLL algorithms. We prove polynomial runtimes for our X-GG algorithms and also prove their output to be X-DeepLLL reduced. Our experiments on non-preprocessed bases show that X-GG produces better quality outputs whilst being much faster than the corresponding DeepLLL algorithms. We also compare SS-GG and the FPLLL implementation of BKZ with LLL-preprocessed bases. In small dimensions (40 to 210), SS-GG is significantly faster than BKZ with block sizes 8 to 12, while simultaneously also providing better output quality in most cases. In higher dimensions (250 and beyond), by varying the threshold δ for deep insertion, SS-GG offers new trade-offs between the output quality and runtime. On the one hand, it provides significantly better runtime than BKZ-5 with worse output quality; on the other hand, it is significantly faster than BKZ-21 while providing increasingly better output quality after around dimension 350.

Keywords: Lattice reduction, LLL, deep insertion, greedy global framework, potential, squared sum.

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

1 Introduction

A Euclidean lattice (or just a lattice) $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^m$. It can 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 to 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 major importance. The process of transforming a given basis into one of better quality is generally called lattice reduction. Lattice reduction algorithms have a wide variety of uses, including in the cryptanalysis of lattice-based cryptosystems [3,4], attacks on knapsack cryptosystems [38], and finding small roots of systems of modular equations [13,14,23].

In 1982, Lenstra, Lenstra, and Lovász [29] 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 the 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 and $1/4 < \delta \leq 1$ is a parameter called the *threshold* which determines the quality of reduction. The quality improves as the threshold δ increases. After LLL terminates, the vector $\mathbf{b}_i$ in the output basis is an exponential approximation of the i^{th} shortest linearly independent vector in the lattice. In [29], LLL was shown to run in polynomial time using an argument surrounding a quantity known as the *potential* of the basis – a measure of basis quality that we will describe soon. LLL has many applications including in cryptology [36], algorithmic number theory [12], factoring polynomials [26], Diophantine approximation [21], etc.

Schnorr and Euchner [43] introduced a variant of the LLL algorithm called LLL with deep insertions, or DeepLLL. The key algorithmic novelty was in the reordering of the vectors. They introduced the notion of *deep insertion*, whereby instead of just swapping a 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[3]. This introduced more constraints on the output basis and as a result, the quality of the output basis is

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

provably better than in LLL. In particular, 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 [48, Theorem 1]. However, DeepLLL requires additional size reduction steps and bookkeeping, which makes it significantly more time-consuming than LLL.

In the same paper [43], the authors introduced an algorithm which performs the block Korkin-Zolotarev (BKZ) reduction. Since then, BKZ has been more researched than DeepLLL-style algorithms and has become state-of-the-art in lattice reduction. BKZ takes as input a parameter β denoting the *block size*. It iteratively reduces consecutive projected blocks of size β by calling a shortest vector problem (SVP) oracle on these blocks and inserting the resultant vector into the basis. As β increases, the algorithm outputs bases of better quality, however the runtime also increases. Improvements were made to BKZ in [11] by incorporating a pruning technique on the enumeration SVP subroutine, which decreases the runtime without any decrease in basis quality on output. Furthermore, the authors introduced preprocessing of the local bases and reducing the enumeration radius; both to reduce the runtime of the enumeration subroutine. The FPLLL library [46] provides the state-of-the-art implementation of BKZ.

Since Schnorr and Euchner introduced DeepLLL, there have been two new deep-insertion-based algorithms – Pot-LLL [16] and SS-LLL [48]. 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 $\mathbf{B}^*$. 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 provide 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 each iteration of DeepLLL and its variants Pot-DeepLLL and SS-DeepLLL, the algorithms only work with the sublattice $\mathcal{L}_k$ generated by the subset of vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. Each of these algorithms attempts to iteratively improve some basis quality measure. The use of Pot and SS 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 the quality of the basis. In contrast, Pot and SS 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.

The above generalisation leads us to our new generic framework of algorithms. We ask the following question – *if the general principle of LLL-style algorithms* *is to iteratively improve the basis quality, is there a greedy approach to improve it as much as possible through deep insertions in every iteration?*

Previous LLL-style algorithms [29,43,34,16,48] maintain an index k of the vector to be inserted at a previous position $i \in \{1, \ldots, k-1\}$ in the basis ordering, to improve the basis quality. In LLL [29] and L^2 [34], $i = k-1$ is a fixed previous position for a certain k, while in algorithms using deep insertions [43,16,48], the choices for the deep insertion position i are restricted within the sublattice $\mathcal{L}_k$ in an iteration. We observe that a deep insertion of a pre-determined vector $\mathbf{b}_k$ at a position i can be substituted by a deep insertion of $\mathbf{b}_{k'}$ at position i' (for some $2 \leq k' \leq n$ and $i' \in \{1, \ldots, k'-1\}$) such that the basis quality improvement is better. *In fact, there is a pair of indices (i', k') for which the quality improvement is the maximum possible at that point.* This observation is the basis of our greedy choice of vectors for reordering the basis.

In this work, we move away from the technique of maintaining an index k and working with a sublattice $\mathcal{L}_k$. We propose a new generic framework for lattice reduction through an algorithm X-GG. Our algorithm works with a general quality measure $\mathrm{X}(\mathbf{B})$ of the basis $\mathbf{B}$. To iteratively improve the quality of the basis, *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. Such deep insertions are carried out *in each iteration* as long as the measure of the reordered basis decreases by *at least* a fraction $(1-\delta)$ of its previous value, where δ is the *threshold for deep insertion.* When the algorithm terminates, the output basis is guaranteed to have a measure that cannot be reduced appreciably (by a fraction $1-\delta$ or more) any further through deep insertions. For a measure X, we call such a basis δ-X-DeepLLL reduced. *By choosing the maximum change in* X *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 is a key determining factor in the framework of algorithms we propose. We instantiate our generalised algorithm X-GG with the measures Pot and SS in place of X to get the Pot-GG and SS-GG algorithms respectively. We prove that X-GG outputs a δ-X-DeepLLL reduced basis and provide theoretical bounds on the runtime of X-GG. We prove the concrete polynomial runtime bit-complexities for both Pot-GG and SS-GG using exact $\mathbb{Q}$ arithmetic and show that they are the same as their X-DeepLLL counterparts.

We conduct extensive experiments to compare the performances of LLL, Pot-DeepLLL, SS-DeepLLL and BKZ with our Pot-GG and SS-GG algorithms, on SVP Challenge style bases [20,15], using floating-point implementations of all algorithms. Our implementations, the input bases we have used in our experiments and the outputs of said experiments are available at [7]. Other than our own algorithms, this repository has the only publicly available implementation of Pot-DeepLLL and SS-DeepLLL with the incorporation of the techniques from [47] for efficiently computing the GSO coefficients $\mu_{i,j}$ and the squared lengths of the GSO basis vectors $\|\mathbf{b}_i^*\|^2$, to the best of our knowledge.

We assess the algorithms in two ways – (1) as *standalone* algorithms running on bases of dimensions 40 to 150 that have not been preprocessed in any way, and (2) by running them on bases of dimensions 40 to 600 that have been *preprocessed* with 0.99-LLL (i.e. LLL with $\delta = 0.99$), for fair comparison with the FPLLL [46] implementation of BKZ. For standalone comparisons in dimensions 40 to 150, we use the NTL library for the multi-precision arithmetic computations. For comparisons with LLL-preprocessed bases in dimensions 40 to 210, we use standard data type implementations. We additionally compare SS-GG and BKZ at dimensions 250 to 600 in detail. Multi-precision floating-point arithmetic is required at these dimensions both in our implementations and the FPLLL implementation of BKZ to ensure correctness of the computations. Since the FPLLL library [46] in turn uses the GNU MPFR library [22] for the multi-precision arithmetic computations, we use an MPFR implementation of SS-GG for fairness in this comparison.

For all experiments requiring multi-precision implementations, we use experimentally estimated values of precision. We observe that at certain smaller precisions, the output bases of X-GG get very close to being X-DeepLLL reduced, but are not quite there yet. A simple trick of recomputing the values of $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$ from the integer vectors just once on the output basis and then running the SS-GG algorithm again quickly makes it X-DeepLLL reduced. Recomputing these values help to get rid of the floating-point errors accumulated thus far by the algorithm. We use this trick for the higher dimension (250 to 600) experiments to be able to run X-GG with a smaller precision, providing better runtime.

In the detailed comparison of the behaviour of δ-SS-GG and BKZ-β in terms of efficiency and output quality, we vary the values δ and β. To provide deeper understanding of the algorithms, we also compare the average number of deep insertions in SS-GG with the average number of SVP calls in BKZ, as well as the depth of deep insertions of SS-GG with the BKZ block size. Taking δ closer to 1 outputs a basis with a greatly improved output quality but with slower runtime, whilst reducing the value of δ yields a weaker reduction with a vast improvement in runtime. This allows for one to choose a value for the parameter δ which suits a particular application.

Here, we present the key findings from our experiments. The output quality is measured uniformly[4] for all algorithms with the Root Hermite Factor (RHF) and the length of the shortest vector in the output basis. We observe that the output quality of an algorithm does not vary between the standalone and the preprocessed experiments. *X-GG has better output quality than the corresponding X-DeepLLL algorithm* (Figures 1 and 3). At dimension 210, SS-GG outputs shortest vectors that are on average 11.6% shorter than those obtained by SS-DeepLLL (Table 2).

[4] Note that Pot, SS, etc. are various measure of output quality. However, the RHF and the length of the shortest vector are the common scales to uniformly measure the quality of all algorithms.

The standalone comparison of LLL, X-DeepLLL and X-GG algorithms shows that the *runtime of SS-GG is only second to LLL* (Figure 4) while providing significantly better output quality than LLL (Table 2). *X-GG algorithms are much faster than the corresponding X-DeepLLL algorithms*, especially as the dimension grows (Figure 4). At dimension 150, SS-GG is around 2.3 times faster than SS-DeepLLL and Pot-GG is about 1.4 times faster than Pot-DeepLLL (Table 4).

The preprocessed comparisons are split into two parts. First, we compare X-DeepLLL, X-GG and BKZ using standard data types at dimensions 40 to 210. Our experiments show the surprising result that *X-DeepLLL and X-GG algorithms are faster than BKZ with* $\beta \geq 8$ *for all dimensions* 40-210 (Figure 5). We conjecture that incorporating the techniques from [47] in our implementations is perhaps the main reason behind this excellent runtime performance of X-DeepLLL and subsequently X-GG. (Whilst it was reported in [16] that Pot-DeepLLL has a runtime comparable to BKZ-5, there was no claim in [48] regarding the runtime performance of SS-DeepLLL compared with BKZ.) We also note that in preprocessed comparison, X-GG algorithms are slower than the corresponding X-DeepLLL algorithms (Figure 5), while still providing better output quality (Figure 1). In Section 6.3, we provide intuitive experimental justification for this change in the runtime compared to standalone results by showing that *even though X-GG requires significantly fewer deep insertions than X-DeepLLL, it cannot compensate for the increased number of size reductions.*

In preprocessed comparison, *SS-GG with* $\delta = 1 - 10^{-6}$ *is significantly faster than BKZ with block sizes* $8, 10, 12$ *in dimensions* 40 *to* 210 (Figures 1 and 5). In these dimensions, *the RHF of SS-GG with* $\delta = 1 - 10^{-6}$ *is better than BKZ-*8 *throughout, BKZ-*10 *from around* $n = 60$ *and BKZ-*12 *from around* $n = 100$. Moreover, SS-GG is around 20 times faster than BKZ-12 at dimension 40, around 16 times faster at dimension 100, and around 6 times faster at dimension 210 (Table 5).

In Section 6.4, we further compare the behaviour of SS-GG and BKZ in much more detail at dimensions 250 to 600 (Table 6; Figures 6 and 7). We vary the values of the threshold δ for deep insertion in SS-GG from $1 - 10^{-4}$ to 1, while we run BKZ with block sizes $\beta = 5, 8, 10, 12, 14, 18, 20$ and 21. These extensive higher dimension results on SVP challenge bases confirm previous results on the performance of BKZ [19,41,2] at higher dimensions. These higher dimension results also contrast the observation of [48] that increasing the threshold beyond $\delta = 1 - 10^{-6}$ in an SS-based algorithm does not cause any significant improvements to the RHF at lower dimensions. Figures 6 and 7 show the runtime and output quality of δ-SS-GG and BKZ-β for these various values of δ and β. The most striking aspect of the comparison is in Figure 7. *While the RHF of BKZ remains almost constant as the dimension increases*[5]*, the RHF of SS-GG is not constant. It varies with the dimension* n *and the threshold* δ *for deep insertion.* With smaller values of δ, SS-GG is much faster and the RHF can compete with

[5] This behaviour was previously noted in [19] for $n \leq 200$ in all lattice reduction algorithms they tested, including DeepLLL.

small block size BKZ. In fact, SS-GG with $\delta = 1 - 10^{-5}$ beats BKZ-5 both in runtime and output quality in dimensions $250 - 300$. SS-GG with $\delta = 1 - 10^{-4}$ is much faster than BKZ-5 for all dimensions. At dimension 600, SS-GG with $\delta = 1 - 10^{-4}$ is around 5.39 times faster than BKZ-5, while their RHFs are 1.01804 and 1.01632, respectively. Therefore, SS-GG with $\delta = 1 - 10^{-4}$ achieves a middle ground between LLL and BKZ-5. Using values of δ closer to the extreme value of 1 yields much stronger reduction, with output quality getting closer to BKZ-$18, 20, 21$. In particular, running SS-GG with $\delta = 1$ can output bases with a smaller RHF than BKZ-$18, 20, 21$ as the dimension grows (Figure 7). However, the SS-GG runtime also increases (Figure 6). Nevertheless, due to the drastic runtime increase of BKZ-β for $\beta > 20$, SS-GG with $\delta = 1$ runs significantly faster than BKZ-21 in dimensions 250 to 400 that we could test up to. Furthermore, from around dimension 350, the RHF of SS-GG keeps getting significantly better than BKZ-21 (Figure 7). So *choosing different values of δ in SS-GG provides different trade-offs between runtime and output quality.* Finally, we conclude that SS-GG and BKZ are two very different algorithms. *In SS-GG, the shallower insertions lead to better RHF, while in BKZ, the larger block size leads to better RHF.*

The outline of the paper is as follows. Section 2 details the relevant notation and gives an overview of lattices. Section 3 describes 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. Appendix A has additional data and plots for deeper insights on the behaviour of the algorithms.

Related Works. Yamaguchi and Yasuda in [47] described an efficient algorithm for updating the GSO information in DeepLLL. Since the update of the GSO coefficients $\mu_{i,j}$ and the values of $\|\mathbf{b}_i^*\|^2$ is dominant in algorithms using deep insertions, this work [47] is of great importance to our framework.

The original LLL algorithm [29] was known to run in polynomial time for the threshold $\delta < 1$. For $\delta = 1$, it is polynomial time for fixed dimensions [1]. Although DeepLLL [43] is not known to run in polynomial time, its variants Pot-DeepLLL [16] and SS-DeepLLL [48] are both polynomial time algorithms. The runtime of LLL [29] and Pot-DeepLLL [16, Proposition 1] for an input basis $\mathbf{B}$ is bounded by $\mathrm{Pot}(\mathbf{B})$ that decreases every time the basis is reordered (respectively through swaps and deep insertions) by these algorithms. In [49], it was proved that the squared sum $\mathrm{SS}(\mathbf{B})$ also decreases with every swap of LLL. The runtime of SS-DeepLLL is similarly bounded by $\mathrm{SS}(\mathbf{B})$ [48]. The proofs for runtime complexity of our algorithms follow similar techniques, using bounds on the quality measure X and that it decreases in every iteration.

The basis potential $\mathrm{Pot}(\mathbf{B})$ has been used for constructing new algorithms and their analyses. In [24], LLL was examined based on maximally reducing the basis potential for a given lattice. Whilst LLL continues until the potential can not be further reduced by a factor of δ, this does not mean that LLL reduces the potential maximally. Instead, a new notion of basis reduction was introduced in [24] with the aim to find a basis $\mathbf{B}$ with potential smaller than the potential of all other bases $\mathbf{B}'$ for the same lattice. This technique is different from our greedy deep insertion minimising a measure at each iteration. In [10], it was pointed out that the potential does not capture the typical unbalancedness demonstrated by the GSO norms. The authors generalised $\mathrm{Pot}(\mathbf{B})$ for the whole lattice $\mathcal{L}$ to $\mathrm{Pot}_k(\mathbf{B})$ for a sublattice $\mathcal{L}_k$ with only the first k basis vectors, and demonstrated the usefulness of this more granular measure.

The squared-sum $\mathrm{SS}(\mathbf{B})$ has also been demonstratively useful. Fukase and Kashiwabara [17] showed that a basis with a smaller squared-sum (SS) allows more short lattice vectors to be sampled using Schnorr's random sampling. This method was used in [49] to sample short vectors.

Some greedy approaches have been used in lattice reduction prior to this work. Lenstra [30] introduced the idea of a flag to choose the next basis in the reduction algorithm. Two bases have the same flag if their GSO vectors are the same, or only differ by their sign. The choice for the next flag to move to is greedy in some sense. In [35], the authors used a greedy technique to size reduce basis vectors. A basis vector $\mathbf{b}_k$ is size reduced using a vector from the sublattice $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$ that is *closest* (hence greedy) to $\mathbf{b}_k$. Our greedy approach is different from the above as it minimises a basis quality measure X through deep insertions in each iteration.

An important direction in improving lattice reduction algorithms is their floating-point arithmetic (*fpa*) considerations and their consequent optimisations. Schnorr and Euchner [43] described the LLL algorithm using *fpa* that was used as a subroutine in their description of BKZ. Nguyen and Stehlé [34] significantly improved the precision handling of LLL using the Cholesky Factorisation Algorithm (CFA) and the Iterative Babai Nearest Plane Algorithm [5,6]. Their L^2 algorithm is much faster than LLL and is implemented in the FPLLL library [46].

It is well understood that lattice reduction algorithms generally provide much better output quality in practice than their theoretical bounds. Experimental analyses [33,19,42] have been performed on the average-case behaviour of LLL, and comparisons are drawn with the worst-case theoretical results. In particular, Gama and Nguyen [19] show that in practice, the estimate of the shortest vector output by LLL, DeepLLL and BKZ algorithms are indeed exponential in the dimension, but with smaller constants than their respective theoretical estimates. In other words, the quality of the reduced bases is better than the best theoretical bounds. The same paper also mentions that DeepLLL with deep insertions restricted to blocks of vectors, may be run using much larger block sizes than BKZ. At around block size 20–25, BKZ suffers from a rapid increase in runtime, which is not experienced by DeepLLL experimentally. They conjecture that DeepLLL may outperform BKZ for high dimensional lattices. We restrict our experiments with BKZ to block size $\beta = 21$.

There have been several other variants of LLL. The Segment LLL algorithm introduced in [27], yields a slightly weaker reduction than LLL but is more efficient by a factor n. This is achieved by partitioning a basis of dimension $n = km$ into m segments comprising k consecutive vectors, and LLL reducing these segments. Their algorithm was further improved in [32], where BKZ-like overlapping blocks were reduced, resulting in an asymptotically fast algorithm. Furthermore, the upper bound on the length of the first vector of the basis output from this algorithm is slightly stronger than LLL. In [31], the costly GSO computations were approximated by Householder transformations performed using *fpa*. In [9] a perturbation analysis was performed, analysing how a small change in the basis affects the $\mathbf{R}$ factor of the $\mathbf{QR}$ factorisation. These results may be applied to the floating-point implementations of LLL-type algorithms. In [25], the authors used parallelisation and recursion to improve the efficiency of LLL by decreasing the precision required for reduction. A further improvement on LLL was described in [40], where a notion of recursive reduction based on the drop of the profile (the vector comprising values of $\log \|\mathbf{b}_i^*\|$) of a lattice is introduced. A novel method for precision management was also introduced to experimentally show that their algorithm outperforms the state-of-the-art FPLLL [46] implementation of L^2 as well as the algorithm of [25]. Beyond algorithms improving LLL for general lattices, another important direction is the application of LLL to lattices with an underlying structure or form, for example, ideal lattices [39], module lattices [28] and parametric lattices [8].

We believe that many techniques from the above works may be used to improve output quality, precision handling and runtime of the X-GG algorithms. Our algorithms could also be used to substitute algorithms like LLL for better output quality at the cost of slower runtime.

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 $\mathtt{span}(\mathbf{B}) = \{\mathbf{Br} : \mathbf{r} \in \mathbb{R}^n\}$. A subset of vectors in $\mathbf{B}$ gives 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. For computational purposes, 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 (GSO) 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

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

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 $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}))$ 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

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

This can be written as $\left(\delta - \mu_{k,k-1}^2\right) \cdot \left\|\mathbf{b}_{k-1}^*\right\|^2 \leq \|\mathbf{b}_k^*\|^2$ 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

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

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 $\mu'_{i,j}$.

for $j = k-1, \ldots, 1$ /* The 'reverse order' as in Remark 2 */ **do**
  if $|\mu_{k,j}| > \frac{1}{2}$ **then**
    $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$
    $\mu_{k,j} \leftarrow \mu_{k,j} - \lfloor \mu_{k,j} \rceil$
    for $i = 1, \ldots, j-1$ **do**
      $\mu_{k,i} \leftarrow \mu_{k,i} - \lfloor \mu_{k,j} \rceil \mu_{j,i}$ /* As in Remark 1 part (3) */
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 $\mathbf{b}_i$ for $1 \leq i < k$ 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* $1 \leq j < i \leq n$, $|\mu_{i,j}| \leq 1/2$.

We also define the notion of size-reduction where the vector $\mathbf{b}_i$ is unchanged if $|\mu_{i,j}| < \eta$ for some $\eta \geq 1/2$. This is important for implementations of size reduction where floating-point approximations to real numbers are used.

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

Remark 1 (Changes in GSO Coefficients Upon Size Reduction). Based on the descriptions in [12, Chapter 2] and [18], 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 [12] and [18, Exercise 17.4.8 (3)]. The proof is as follows[6]. Let $\mathbf{b}_i$ be already size reduced with respect to the vectors $\mathbf{b}_{i-1}, \mathbf{b}_{i-2}, \ldots, \mathbf{b}_{j+1}$. 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

$$\mu'_{i,l} = \frac{\langle \mathbf{b}'_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|} - \lfloor \mu_{i,j} \rceil \frac{\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|}.$$

[6] Although quite straightforward, the proof is not detailed in the literature to the best of our knowledge.

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

$$\mu'_{i,l} = \frac{\langle \mathbf{b}_i, \mathbf{b}_l^* \rangle}{\|\mathbf{b}_l^*\|^2} - \lfloor \mu_{i,j} \rceil \frac{\langle \mathbf{b}_j, \mathbf{b}_l^* \rangle}{\|\mathbf{b}_l^*\|^2} = \frac{\langle \mathbf{b}_i, \mathbf{b}_l^* \rangle}{\|\mathbf{b}_l^*\|^2} = \mu_{i,l}.$$

3. For all $1 \leq l \leq j-1$, 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 $j < l \leq i-1$ *do not change*. However, the values $\mu_{i,l}$ for $1 \leq l \leq j$ *may change*.

Remark 2 (Reducing in Reverse). Note that in Algorithm 1, while size reducing the vector $\mathbf{b}_k$ with $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$, 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 $1 \leq i < k$ 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. Several measures 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 [19]. 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 [19] 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 [29] 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 $\mathbf{B}^*$:

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

This quantity was introduced in [17] in the context of sampling short lattice vectors, and has then been used in [48] for lattice reduction. Similarly to Pot, the squared sum varies with changes in the lengths of the GSO vectors. However, unlike Pot, 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 $1 \le i < k \le n$ defined as follows.

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

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 $\mathrm{X}(\mathbf{B})$ 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

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

In particular, we get $\mathit{\Delta}\mathrm{Pot}_{i,k} = \mathrm{Pot}(\mathbf{B}) - \mathrm{Pot}(\sigma_{i,k}(\mathbf{B}))$ and $\mathit{\Delta}\mathrm{SS}_{i,k} = \mathrm{SS}(\mathbf{B}) - \mathrm{SS}(\sigma_{i,k}(\mathbf{B}))$[7]. We note that $\texttt{argmax}_{1\le i<k\le n}(\mathit{\Delta}\mathrm{X}_{i,k})$ returns the pair of indices (i,k) for which the value of $\mathit{\Delta}\mathrm{X}_{i,k}$ is maximised.

[7] 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.

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.

Definition 3 (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 [29]

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 δ-LLL reduced

Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
$k \leftarrow 2$
while $k \leq n$ **do**
  Size reduce $\mathbf{b}_k$ /* As in Algorithm 1 */
  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**
    $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$ /* Swap vectors $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$ */
    Update $\mathbf{b}_{k-1}^*, \mathbf{b}_k^*$ /* As in [18, Lemma 17.4.3] */
    Update $\mu_{i,j}$'s /* As in [12, Algorithm 2.6.3] */
    $k \leftarrow \max(k-1, 2)$
  else
    $k \leftarrow k+1$
return $\mathbf{B}'$, a δ-LLL reduced basis

Definition 4 (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 [29] is described in Algorithm 2. The output basis $\mathbf{B}'$ is δ-LLL reduced as in Definition 3. 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 [43] and its variants [16,48] to a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ where $1 \le i < k \le n$. 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 $\delta \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \le \|\pi_{k-1}(\mathbf{b}_k)\|^2$. This can be written as

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

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 $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$ were 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 the vector $\mathbf{b}_k$ to the earlier position $k-1$ in the basis ordering. In general, for $1 \le i < k \le n$, the Lovász condition for a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ is given by $\delta \cdot \|\pi_i(\mathbf{b}_i)\|^2 \le \|\pi_i(\mathbf{b}_k)\|^2$ which can also be written similarly as

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

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* $1 \le i < n$, *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

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

which fails if $\Delta\mathrm{LC}_i > (1-\delta) \cdot \mathrm{LC}_i(\mathbf{B})$ and calls for 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 $\mathrm{X}(\mathbf{B})$ 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 $\mathrm{X}(\mathbf{B})$.

Variants of DeepLLL: transition from a local measure to a global measure of quality. As noted above, the Lovász condition in LLL [29] and its generalisation in DeepLLL [43] 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 [16] where instead of a localised measure of quality, the potential Pot was used in DeepLLL so that the effect of permuting vectors on the entire basis is considered. In SS-DeepLLL [48], Pot was replaced by another global measure SS. The basic operation of deep insertion for reordering the basis vectors is the same in all three algorithms.

Definition 5 (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 *if the following two conditions are satisfied.*

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

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

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. The 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 Pot and SS in place of the generic X, Definition 5 is instantiated to that of a Pot-DeepLLL [16] reduced basis and a SS-DeepLLL [48] reduced basis. We know from [16, Lemma 2] that a Pot-DeepLLL reduced basis is LLL reduced. Also, from [16, Lemma 3], for $1/4^{n-1} < \delta \leq 1$, a δ-DeepLLL reduced basis is δ^{n-1}-Pot-DeepLLL reduced. From [48, 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 [48, 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 $\mathrm{X}(\mathbf{B})$ of the quality of $\mathbf{B}$. The generalisation is instantiated for different local and global quality measures of a basis $\mathbf{B}$ that

Algorithm 3: The X-DeepLLL Algorithm

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_n)$, a threshold $0 < \delta \leq 1$
Output: $\mathbf{B}' = (\mathbf{b_1}', \dots, \mathbf{b}'_n)$ which is δ-X-DeepLLL reduced
1 Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
2 Initialise other bookkeeping data structures, if required for $\mathrm{X}(\mathbf{B})$
3 $k \leftarrow 2$
4 **while** $k \leq n$ **do**
5   Size reduce $\mathbf{b}_k$ /* As in Algorithm 1 */
6   **for** $i = 1$ **to** $k - 1$ **do**
7   **if** $\Delta\mathrm{X}_{i,k} > (1 - \delta) \cdot \mathrm{X}(\mathbf{B})$ **then**
8             $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ /* Deep insert $\mathbf{b}_k$ before $\mathbf{b}_i$ */
9   Update $\mathbf{B}^*$ and $\mu_{l,j}$ /* As in [47, Theorem 1 and Proposition 1] */
10             $k \leftarrow \max(i, 1)$
11   **break**
12     $k \leftarrow k + 1$
13 **return** $\mathbf{B}'$, a δ-X-DeepLLL reduced basis

are all based on the GSO vectors $\mathbf{B}^*$. The localised measure $\mathrm{LC}_i(\mathbf{B}) = \|\mathbf{b}_i^*\|^2$ (used in DeepLLL [43]) is only for a single GSO basis vector, while the measures Pot [16] and SS [48] are on the entire GSO basis $\mathbf{B}^*$. In Remark 4 we have 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 [43]. Pot-DeepLLL and SS-DeepLLL are both variants of DeepLLL for X = Pot and X = SS respectively. A key feature of these algorithms is the deep insertion of a basis vector $\mathbf{b}_k$ at the earliest position i satisfying the condition in Line 7 of Algorithm 3. If such a position is found, the deep insertion is executed, the GSO is updated and k is set to $i + 1$ because all vectors up to position i are already size reduced. Line 12 of Algorithm 3 ensures that the algorithm proceeds to consider the next vector for size reduction and possible deep insertion.

In the generalised X-DeepLLL algorithm 3, we note that the threshold value δ represents a fraction of the measure $\mathrm{X}(\mathbf{B})$. If a reordering of the basis $\mathbf{B}$ can change its quality by *more than* $(1 - \delta) \cdot \mathrm{X}(\mathbf{B})$, 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 $\delta \cdot \mathrm{X}(\mathbf{B})$, that is when the basis is considered to be δ-X-DeepLLL reduced as in Definition 4. As may be expected, the threshold δ depends on the measure X in the context. The notation δ is commonly used [29,43,16] to denote the fraction in the context of algorithms based on the localised measure LC (when using the Lovász condition) and the measure Pot for the whole basis. The notation η has been used in [48] to denote the threshold in the context of the measure SS. 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 and X_2, 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 $\mathrm{X}_1, \mathrm{X}_2$ as in [16,48].

Since X-DeepLLL (Algorithm 3) attempts to *reduce* the measure $\mathrm{X}(\mathbf{B})$ to at most $\delta \cdot \mathrm{X}(\mathbf{B})$ in each iteration, we have $\delta \leq 1$ and, consequently, $1 - \delta \geq 0$. In particular, for $\delta = 1$, a deep insertion is allowed for any decrease $\Delta\mathrm{X} > 0$ in the measure. Assuming the measure $\mathrm{X}(\mathbf{B}) > 0$ for any basis $\mathbf{B}$, since the decrease in the measure $\Delta\mathrm{X}$ can not be more than or equal to the measure $\mathrm{X}(\mathbf{B})$ itself, hence we necessarily have $\delta > 0$. For the algorithms using the measure LC (based on the Lovász condition), the threshold must further satisfy $\delta > 0.25$ [29].

The LLL-style algorithms mentioned above (LLL, DeepLLL, SS-DeepLLL and Pot-DeepLLL) 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 $(\mathbf{b}_1, \ldots, \mathbf{b}_i)$. Note from Algorithm 3 Step 5, at the start of an iteration of X-DeepLLL, the vector $\mathbf{b}_k$ is size reduced with all previous vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$. Therefore, 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 only size reduce one vector $\mathbf{b}_k$ in Line 5 of 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 are given by [47, Theorem 1]. The corresponding changes in the GSO coefficients are given by [47, Proposition 1].

Remark 6. From [47, 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 $i \leq j \leq k$, and $i + 1 \leq l \leq n$.

4 The X-GG 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-GG algorithm 4 provides a general description of algorithms realised by specifying a basis quality measure X. The algorithm starts by finding the GSO basis

Algorithm 4: The X-GG 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 a δ-X-GG 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 $(i', k') = \texttt{argmax}_{1 \leq i < k \leq n}(\Delta \mathrm{X}_{i,k})$ and set $\Delta \mathrm{X} = \Delta \mathrm{X}_{i',k'}$
4 **while** $\Delta \mathrm{X} > (1 - \delta) \cdot \mathrm{X}(\mathbf{B})$ **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 [47, 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 $(i', k') = \texttt{argmax}_{1 \leq i < k \leq n}(\Delta \mathrm{X}_{i,k})$ and $\Delta \mathrm{X} = \Delta \mathrm{X}_{i',k'}$
9 **end**
10 **return** $\mathbf{B}'$, a δ-X-DeepLLL reduced basis.

in step 1 followed by 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 $\Delta \mathrm{X} = \mathrm{X}(\mathbf{B}) - \mathrm{X}(\sigma_{i',k'}(\mathbf{B}))$ is greater than a fraction $(1 - \delta)$ of the current measure $\mathrm{X}(\mathbf{B})$. 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 $\mathrm{X}(\mathbf{B})$ by a fraction $(1-\delta)$ that may be considered as a substantial change to the quality of the basis. Thus, X-GG returns a δ-X-DeepLLL reduced basis as in Definition 5. We prove this in Lemma 2.

In a single iteration, LLL, DeepLLL, Pot-DeepLLL and SS-DeepLLL either increase the index $2 \leq k \leq n$ by 1, or decrease the index by some value i $(< k - 1)$. 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. 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 $\Delta\mathrm{X}(\mathbf{B})$ 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 $\mathrm{X}(\mathbf{B})$ 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[8]) 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 get reflected in the experimental results provided in Section 6 where our algorithms perform exceedingly well with respect to previous LLL-style algorithms in terms of their runtimes, the total number of deep insertions, and total number of size reductions.

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 a δ fraction) in that iteration, but creates the scope for a larger decrease in the measure in subsequent iterations. We do not consider such strategies in this work and leave them for future consideration. 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-GG and for the squared sum, we get SS-GG. 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[9]. However, these computations can be done on the fly, and are covered by step 8 of Algorithm 4.

Note that $\mathrm{Pot}(\mathbf{B})$ and $\mathrm{SS}(\mathbf{B})$ are global measures of quality of the basis. Given that the local measure $\mathrm{LC}_i(\mathbf{B}) = \|\pi_i(\mathbf{b}_i)\|^2 = \|\mathbf{b}_i^*\|^2$ is on a single GSO vector, and not on the entire basis, more careful consideration must be taken to establish a greedy approach on independent $\mathrm{LC}_i(\mathbf{B})$ measures for different indices i. So we only instantiate Algorithm 4 using global measures in this work.

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

[9] In Pot-DeepLLL, when computing the position i for deep inserting a vector $\mathbf{b}_k$, it is necessary to compute the projections $\pi_i(\mathbf{b}_k)$ 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 insertion can be computed directly using the GSO information that was updated in a previous step without computing a projection [48, Equation 5].

Remark 7 (Preprocessing Reduction). The description of Pot-DeepLLL in [16, Algorithm 1] includes a preprocessing of the basis $\mathbf{B}$ by LLL. In the case of the SS-DeepLLL algorithm in [48, Algorithm 2], the description itself does not include the preprocessing step. However, they have included the preprocessing step with 0.99-LLL (that is LLL with $\delta = 0.99$) while reporting the performance results [48, Section 4.3.3]. We note here from [48] 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. Furthermore, any lattice reduction algorithm can be used for preprocessing the basis before being fed into a second algorithm for further reduction. We have excluded the preprocessing step from our theoretical descriptions and asymptotic analysis and have focused on their independent performances. In our experiments, however, we have examined the standalone algorithms as well as the algorithms after the basis has been 0.99-LLL preprocessed.

The BKZ algorithm [43] runs LLL as a preprocess before applying the blockwise reduction. We use the FPLLL [46] library implementation of BKZ that inherits this feature in our experiments.

5 Theoretical Results

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

Proof. In every iteration of Algorithm 4, the measure decreases by $\Delta\mathrm{X}(\mathbf{B}) = \mathrm{X}(\mathbf{B}) - \mathrm{X}(\sigma_{i,k}(\mathbf{B}))$. Since it can keep decreasing only until $Z_{\mathrm{X}} > 0$, 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 [47, 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, however* $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ *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 $j < l$, $i \le j \le k$, and $i + 1 \le l \le n$.

- In particular, for vectors $\mathbf{b}_l$ for $i+1 \leq l \leq k$, the following things change.
  - The GSO of $\mathbf{b}_l$ 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 $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_i, \ldots, \mathbf{b}_{l-1}))$.
  - Also, $\mathbf{b}_l$ may not be size reduced with respect to this newly inserted vector $\mathbf{b}_k$.

  Hence, we start with $\mathbf{b}_i$ (now in position $i+1$), and size reduce it with the newly inserted vector $\mathbf{b}_k$. Upon this size reduction, for $1 \leq t < k$, the values of $\mu_{i+1,t}$ will be updated by part (3) of Remark 1. Hence, we must size reduce $\mathbf{b}_i$ with all vectors $\mathbf{b}_k, \mathbf{b}_{i-1}, \ldots, \mathbf{b}_1$ in reverse as explained in Remark 2. We similarly reduce all vectors $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$.
- Reordering the vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ does not change $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_k))$. 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 $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_k))$ remain the same. Thus, their GSO remains the same. However, 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 the vectors in positions $i'+1, \ldots, n$, 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 5.*

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

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

for all $1 \leq i < k \leq n$. Equivalently, $\delta \cdot \mathrm{X}(\mathbf{B}') \leq \mathrm{X}(\sigma_{i,k}(\mathbf{B}'))$ for all $1 \leq i < k \leq n$. Also by Lemma 1, the basis $\mathbf{B}'$ on output is size reduced. Hence, the output of X-GG is a δ-X-DeepLLL reduced basis as per Definition 5.

In Algorithm 4, the basis quality measure $\mathrm{X}(\mathbf{B})$ 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 $1 \leq i \leq k \leq n$. 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-GG algorithm.

Lemma 3. *Let $\|\mathbf{b}_i\|^2 \leq C$ for all $1 \leq i \leq n$ 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 $\Delta\mathrm{X}_{i,k}$ 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* $\mathrm{X}(\mathbf{B})$ *and let* $0 < \delta < 1$[10]*. Then the total number of bit operations performed by the X-GG algorithm 4 is given by*

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + f_{\mathrm{X}}(C, m, n)\right) \log_{1/\delta}\left(\frac{I_{\mathrm{X}}}{Z_{\mathrm{X}}}\right)\right) \tag{2}$$

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 the length of the vectors in the basis $\mathbf{B}$ do not increase throughout the algorithm [29]. All arithmetic operations are on integers of size $O(n \log C)$ bits by the same argument as in [29][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 $\mathrm{X}^{(i)} = \frac{1}{\delta^i}\mathrm{X}$ satisfies

$$Z_{\mathrm{X}} \leq \frac{1}{\delta^i}\mathrm{X} \leq I_{\mathrm{X}}.$$

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

$$\log_{1/\delta}\left(\frac{I_{\mathrm{X}}}{Z_{\mathrm{X}}}\right).$$

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 [47, 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 $\Delta\mathrm{X}_{i',k'}$ is the minimum among all possible pairs. We assume this step requires $\mathcal{O}(f_{\mathrm{X}}(C, m, n))$ bit operations in every iteration.

[10] 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.

So Algorithm 4 needs a total of $\mathcal{O}\left(n^4 \log^2 C + mn^4 \log^2 C + f_{\mathrm{X}}(C, m, n)\right)$ 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

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

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 much more efficient than what is denoted by its worst case asymptotic complexity.

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

5.1 Computational Complexity of Pot-GG

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

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

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-GG. From Lemma 3, the maximum number of iterations in Pot-GG 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 [16, 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 indices (i, k) satisfying $\mathtt{argmax}_{1 \leq i < k \leq n}(\Delta\mathrm{Pot}_{i,k})$ 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 $i \in \{k-1,\dots,1\}$ where $\Delta\mathrm{Pot}_{i-1,k}$ is computed using the value of $\Delta\mathrm{Pot}_{i,k}$. Note that this optimisation applies to Pot-DeepLLL [16] as well as Pot-GG. Hence, Pot-GG computes $\mathtt{argmax}_{1\le i<k\le n}(\Delta\mathrm{Pot}_{i,k})$ using $\mathcal{O}(n^5\log^2 C)$ bit operations in each iteration. In total, Pot-GG 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 [16, 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-GG. The number of bit operations for each part of the Pot-DeepLLL and Pot-GG algorithms have been listed in Table 1.

5.2 Computational Complexity of SS-GG

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

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

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

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

as was noted in [48, Proposition 2]. From [48, Equation 5], we know that $\Delta\mathrm{SS}_{i,k}$ can be computed in $\mathcal{O}(n^3\log^2 C)$ bit operations. By similar arguments as in Section 5.1 for Pot-GG, the computation of $\mathtt{argmax}_{1\le i<k\le n}(\Delta\mathrm{SS}_{i,k})$ requires $\mathcal{O}(n^4\log^2 C)$ bit operations. Hence, SS-GG 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-GG. The number of bit operations for each part of the SS-DeepLLL and SS-GG algorithms have been listed in Table 1.

Remark 8 (Comparison between X-DeepLLL and X-GG). A comparison between the number of bit operations required in different parts of the X-DeepLLL and

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-GG	$n^4 \log^2 C$	$mn^4 \log^2 C$	$n^5 \log^2 C$	$n^2 \log_{1/\delta} C$
SS-DeepLLL	$mn^3 \log^2 C$	$mn^3 \log^2 C$	$n^3 \log^2 C$	$n \log_{1/\delta} C$
SS-GG	$n^4 \log^2 C$	$mn^4 \log^2 C$	$n^4 \log^2 C$	$\log_{1/\delta} C$

Table 1. Complexity comparison of X-DeepLLL and X-GG.

X-GG 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-GG requires $\mathcal{O}(n^4 \log^2 C)$ bit operations using [47, Algorithm 4]. For size reductions, X-DeepLLL requires $\mathcal{O}(n)$ fewer bit operations than X-GG 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 the index k being considered in an iteration. X-DeepLLL also requires $\mathcal{O}(n)$ fewer bit operations for index search than X-GG. 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-GG, 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-GG 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 [29]. 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-GG, hence it requires $\mathcal{O}(n)$ fewer iterations.

6 Experimental Results

We conduct concrete comparative analysis of LLL [29], Pot-DeepLLL [16], SS-DeepLLL [48], BKZ [43,11] with various block sizes, and our two new algorithms Pot-GG and SS-GG.

The elements of the input bases to these algorithms are often very large integers. The floating-point arithmetic (or *fpa*) may involve a mantissa/significand requiring many bits. To ensure that the algorithms run correctly, implementations use multi-precision data types that can represent numbers using a larger number of bits than the standard data types. With increasing number of bits of precision, the time to execute an arithmetic operation on multi-precision data types increases. The number of bits of mantissa available for computations is called the *floating-point precision* (or *fpp*).

We use the NTL library [44], the GNU MPFR library [22] and the FPLLL library [46] for our multi-precision implementations. The MPFR and FPLLL libraries automatically adjust the number of bits used for integer computations, whereas the NTL library requires this to be set manually. However, the *fpp* has to be specified manually in all three libraries. Values that do not fit the specified *fpp* get rounded. We run our multi-precision implementations with (over)estimated *fpp* to avoid anomalies due to *fpa*. We note from [16, Section 3], [48, Section 4] and Lemma 2 that the X-DeepLLL and the X-GG algorithms terminate if and only if the basis is X-DeepLLL reduced. We utilise this fact to (over)estimate the precision through trial-and-error. A precision is chosen for a dimension when all algorithms successfully terminate for every input basis, producing X-DeepLLL reduced bases.

We first compare the standalone performances of LLL, X-DeepLLL and X-GG, with input bases of dimensions 40 to 150 that have not been preprocessed. Since the BKZ algorithm starts with LLL reduction as preprocessing before running blockwise reduction, it is not part of our standalone comparison. We next compare X-DeepLLL, X-GG and BKZ with various block sizes, on 0.99-LLL preprocessed bases. The state-of-the-art multi-precision implementation of BKZ is available in the FPLLL library [46]. At dimensions 40 to 210, all implementations use standard data types. At dimensions greater than 210, the FPLLL implementation of BKZ using the `long double` data type enters an infinite loop in Babai's algorithm (i.e. the size reduction step) for some input bases. Therefore, the `mpfr_t` data type from the GNU MPFR library [22] is used for *fpa* in the BKZ implementation of FPLLL [46] at higher dimensions. For a fair comparison with BKZ at these higher dimensions, we have implemented (Pot-GG and) SS-GG using the GNU MPFR library [22] and some wrapper functions from the FPLLL library [46] taking advantage of some FPLLL data structures. We extensively compare SS-GG (the best performing GG algorithm in terms of basis quality and runtime) with BKZ at these higher dimensions.

The reporting of our results is as follows. For every dimension, we run the algorithms on a certain number of bases and report the averages of a selection of parameters, in particular, runtime and root Hermite factor (RHF). Our first observation is that even though the output bases are *almost always* different between preprocessed (with 0.99-LLL) and standalone executions of an algorithm, they have very similar RHF. So we do not report them separately. In Section 6.1, we compare the output quality of LLL, X-DeepLLL, X-GG, and BKZ in terms of the RHF and the length of the first vector in the reduced basis. However, the runtime behaviour of the algorithms is significantly different between standalone and preprocessed executions. Hence,we report them separately in Sections 6.2 and 6.3 respectively. The comparison between SS-GG and BKZ at higher dimensions is presented in Section 6.4. The tables and figures for additional insights on the behaviour of the algorithms are in Appendix A.

For LLL, Pot-DeepLLL and Pot-GG we use the threshold value $\delta = 0.99$ for dimensions 40 to 210 as has been common in previous works as well as the default value of δ in FPLLL [46]. For SS-DeepLLL and SS-GG we first use the threshold

$\delta = 1 - 10^{-6}$ in Sections 6.1, 6.2 and 6.3, following the rationale provided in the discussion in [48, Section 4.3.1]. In Section 6.4, we further provide a detailed comparison between δ-SS-GG and BKZ-β by varying δ and β. All algorithms use the size reduction relaxation parameter $\eta = 0.51$ (in place of 1/2) as is the default in the FPLLL library implementations.

Our Implementations. To the best of our knowledge, there is no publicly available implementation of SS-DeepLLL [48, Algorithm 2]. Hence, for the sake of uniformity and fairness, we use our own implementations of all LLL-style algorithms in C++ using *fpa*. We use the `gcc 11.3.0` compiler to run each algorithm on a single Intel® Xeon® Platinum 8358 CPU at 2.60 GHz on a shared memory machine.

We implement LLL, Pot-DeepLLL, SS-DeepLLL, Pot-GG and SS-GG with the NTL data types [44] for fairness in the standalone comparison. Additionally, for preprocessed comparison with the state-of-the-art FPLLL [46] implementation of BKZ, we implement our Pot-GG and SS-GG algorithms using both standard data types and MPFR data types. Overall, we conduct experiments using four versions of Pot-GG and SS-GG:

- Using the NTL [44] data types `ZZ` for integers and `RR` for rational numbers for standalone comparison with classic LLL (not L^2 [34,46]), DeepLLL, Pot-DeepLLL and SS-DeepLLL.
- Using standard data types for both integer and floating-point numbers, we compare the performance of these algorithms with LLL-preprocessed bases for dimensions between 40 and 210.
- Using standard data type for integers, but `mpfr_t` data type for floating-point computations, we compare SS-GG and BKZ with LLL-preprocessed bases for dimensions between 250 and 450.
- Using `mpz_t` data type for integers and `mpfr_t` data type for floating-point numbers, we compare SS-GG with BKZ using LLL-preprocessed bases for dimensions between 500 and 600.

Our implementations, the input lattice bases we use in our experiments and the outputs of our experiments are available at [7].

Input Bases. We generate 300 bases each for dimensions 40 to 210 (in steps of 10), and 10 bases for dimensions 250 to 600 (in steps of 50), unless otherwise stated. The bases are random in the sense of Goldstein and Mayer [20] and are akin to those provided by the SVP Challenge [15]. A basis $\mathbf{B}$ has the form

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

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 at random and $\mathbf{I}$ is the $(n-1) \times (n-1)$ identity matrix.

In the standalone comparison of Section 6.2, we test all 300 bases in dimensions 40 to 90. In dimensions 100 to 150, due to the slow runtime caused by the overestimated floating-point precision, we only test the first 50 bases. Similarly, for the tests at dimension 250 and above, we use only 10 bases for each dimension for the majority of our tests. In the preprocessed comparison provided in Sections 6.1, 6.3 and 6.4, these bases are 0.99-LLL reduced using the FPLLL implementation [46] before being passed as input to the algorithms being compared.

GSO recomputations. We note here that errors due to the rounding of values in *fpa* can accumulate during the execution of a lattice reduction algorithm. This issue is particularly noticeable at higher dimensions. When the *fpp* used is insufficient, the output basis may not be appropriately reduced. For example, the output $\mathbf{B}'$ of our X-GG implementation may not be X-DeepLLL-reduced. However, we have observed that often towards the end of the algorithm execution, the GSO coefficients $\mu_{i,j}$ of $\mathbf{B}'$ are reasonably close to η (= 0.51, the size reduction relaxation parameter). A similar issue may arise when computing the possible change in measure ΔX.

If the values of $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$ are recomputed from the integer basis during the execution of the algorithm, then the accumulated floating-point error is essentially "reset". For simplicity, we call this a "GSO recomputation" step. We utilise this observation to enable our MPFR implementations of X-GG to further reduce an output basis $\mathbf{B}'$ that is perhaps close to completion in being X-DeepLLL reduced. Along with the usual inputs to our implementation, we also pass the maximum number of GSO recomputations allowed. When the algorithm has terminated as it would normally, a check is performed which determines, using a fresh recomputation of the GSO, whether the output is indeed X-GG reduced. If so, then the execution terminates as normal. If not, then, using the recomputed GSO, another execution of the algorithm is performed. This will result in additional size reductions and deep insertions. This procedure continues until either the basis is successfully X-DeepLLL reduced, or the number of GSO recomputations exceeds the value passed as input, and an error is reported stating that the basis has not been sufficiently reduced.

The default number of GSO recomputations in our tests is 1 which is equivalent to at most two runs of X-GG with a recomputation of the GSO coefficients in between.

$$\mathbf{B} \to \mathrm{GG} \to \mathbf{B}' \to \mathrm{GG} \to \mathbf{B}''$$

We tested our implementations to experimentally find a precision for each dimension which successfully reduced all bases with a single GSO recomputation. This allows our implementations to run with significantly smaller precision than would be required for a single run to achieve X-DeepLLL-reducedness, leading to a better runtime.

Experimental Data. Our comparisons of the algorithms are based on the average values of three efficiency parameters, namely, (1) running time (Table 5 and Figure 5 for preprocessed execution; Table 4 and Figure 4 for standalone execution), (2) number of reorderings/deep insertions (Table 10 and Figure 8 for preprocessed execution; Table 8 for standalone execution), and (3) number of size reductions of the basis vectors as in step 3 of Algorithm 1 (Table 10 and Figure 9 for preprocessed execution; Table 9 for standalone execution). We measure the output quality using averages of (1) the root Hermite factor (RHF) (Table 2 and Figure 1), as well as (2) the lengths of the first vector in the reduced bases (Table 3 and Figure 3). For a detailed comparison of the behaviour of δ-SS-GG with BKZ-β at dimensions 250 to 600, we vary δ from $1 - 10^{-4}$ to 1 while we also vary β, to compare their runtimes and RHFs (Table 6 and Figures 7 and 6). For more insights on their behaviour, we compare the average number of deep insertions of SS-GG with the average number of tours in BKZ (Table 11 and Figure 10), and the average depth of deep insertions of SS-GG with the block size of BKZ (Table 11 and Figures 10 and 11). Finally, we compare the average lengths of the nth GSO vectors for comparison of their output qualities (Table 12).

Improving Runtime. In [47, Algorithm 4], the authors described an efficient way to update the GSO coefficients $\mu_{i,j}$ and the values of $\|\mathbf{b}_i^*\|^2$ in an algorithm using deep insertion, to provide better runtime than its naive implementation. We provide the first public implementations of Pot-DeepLLL, SS-DeepLLL along with our own Pot-GG and SS-GG algorithms using the techniques from [47, Algorithm 4].

6.1 Output Quality in Small Dimensions

We first compare the output quality of X-DeepLLL, X-GG and BKZ with $\beta = 8, 10, 12, 20$ in dimensions 40 to 210. The average RHF achieved by each of these algorithms for these dimensions are shown in Table 2 and Figure 1. In Table 2, we also provide the RHF of the LLL-reduced bases for comparison. Furthermore, the average length of the first vector in the reduced bases that give the average RHFs are shown in Table 3 and Figure 3.

Throughout these tests, *the average RHF achieved by X-GG is smaller than the average RHF achieved by X-DeepLLL.* Therefore, X-GG returns a better quality basis on average than the corresponding X-DeepLLL algorithm, whilst achieving the same theoretical notion of reduction (Lemma 2) and the same asymptotic complexity (Sections 5.1, 5.2). For example, at dimension 210, SS-GG outputs shortest vectors that are 11.6% shorter than SS-DeepLLL on average.

When comparing with BKZ, SS-GG (with $\delta = 1 - 10^{-6}$) has a smaller RHF than BKZ-8 in dimensions 40 to 210. Although the SS-GG RHF is larger than BKZ-10 and 12 at dimension 40, it eventually outperforms at higher dimensions in the current range. In Figure 2, the ratios of the average RHFs of SS-GG and BKZ-$8, 10, 12$ and 20 are provided. A value above $y = 1$ implies that the RHF of SS-GG is smaller, whereas a value below it implies that the RHF of BKZ is smaller. Figures 1, 2 show that SS-GG starts outperforming BKZ-10 at around dimension 60 and BKZ-12 around dimension 100. Furthermore, Figure 2 shows that the ratios of RHFs keeps getting better as the dimension increases up to

	Algorithm								
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	BKZ-8	BKZ-10	BKZ-12	BKZ-20	LLL
40	1.01372	1.01341	1.01366	1.01333	1.01352	1.01323	1.01300	1.01243	1.01687
50	1.01427	1.01390	1.01413	1.01355	1.01388	1.01344	1.01316	1.01250	1.01817
60	1.01425	1.01404	1.01410	1.01373	1.01411	1.01375	1.01334	1.01241	1.01860
70	1.01443	1.01418	1.01418	1.01378	1.01427	1.01382	1.01351	1.01248	1.01923
80	1.01457	1.01432	1.01410	1.01378	1.01438	1.01402	1.01359	1.01250	1.01966
90	1.01466	1.01453	1.01407	1.01378	1.01453	1.01404	1.01368	1.01252	1.01985
100	1.01479	1.01460	1.01412	1.01379	1.01462	1.01413	1.01380	1.01260	1.02025
110	1.01484	1.01471	1.01410	1.01372	1.01470	1.01416	1.01383	1.01259	1.02032
120	1.01496	1.01481	1.01414	1.01375	1.01472	1.01428	1.01381	1.01267	1.02056
130	1.01496	1.01484	1.01416	1.01380	1.01481	1.01425	1.01389	1.01268	1.02063
140	1.01504	1.01493	1.01413	1.01377	1.01488	1.01430	1.01395	1.01267	1.02097
150	1.01507	1.01501	1.01414	1.01375	1.01491	1.01434	1.01398	1.01269	1.02092
160	1.01512	1.01501	1.01414	1.01375	1.01495	1.01439	1.01400	1.01269	1.02098
170	1.01521	1.01508	1.01416	1.01373	1.01494	1.01442	1.01400	1.01274	1.02116
180	1.01524	1.01509	1.01414	1.01375	1.01500	1.01448	1.01403	1.01273	1.02127
190	1.01522	1.01511	1.01425	1.01377	1.01504	1.01447	1.01407	1.01276	1.02124
200	1.01526	1.01516	1.01430	1.01383	1.01505	1.01451	1.01408	1.01276	1.02139
210	1.01530	1.01519	1.01449	1.01389	1.01507	1.01451	1.01411	1.01280	1.02149

Table 2. The average RHF using the preprocessed input bases. Plotted in Figure 1.

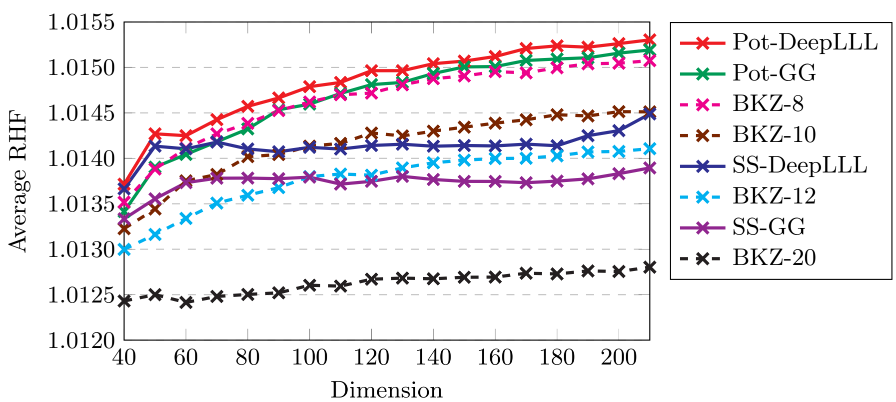

Fig. 1. The average RHF using the preprocessed input bases. Values taken from Table 2.

around dimension 180. The higher dimension tests of Section 6.4 provide deeper insights on the comparison of SS-GG varying δ with BKZ varying β.

Whilst Pot-GG is an improvement on Pot-DeepLLL in terms of quality, it does not compete as well with BKZ. Pot-GG has an RHF less than BKZ-8 at dimension 40 and comparable up to approximately dimension 100.

	Algorithm								
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	BKZ-8	BKZ-10	BKZ-12	BKZ-20	LLL
40	1755.2	1734.2	1751.4	1728.6	1740.6	1720.5	1705.1	1666.8	1992.7
50	2070.2	2033.0	2056.3	1998.0	2030.3	1986.4	1959.0	1895.5	2513.6
60	2385.1	2355.4	2363.6	2311.8	2363.7	2314.0	2257.9	2137.1	3090.8
70	2783.0	2736.5	2734.7	2661.5	2751.6	2667.7	2610.1	2430.8	3883.8
80	3250.9	3188.1	3131.8	3053.7	3201.7	3109.6	3007.5	2758.0	4870.0
90	3790.1	3746.5	3595.2	3502.3	3740.9	3583.5	3469.7	3129.7	6017.0
100	4439.5	4355.0	4154.2	4023.4	4363.5	4159.5	4023.4	3574.4	7617.1
110	5168.4	5100.7	4771.7	4576.4	5089.3	4802.3	4629.6	4048.9	9395.8
120	6081.1	5971.7	5516.6	5265.0	5903.2	5603.3	5302.8	4630.1	11804.2
130	7058.0	6939.7	6358.7	6076.8	6915.3	6432.8	6147.6	5261.1	14601.8
140	8276.4	8153.9	7299.2	6939.5	8087.2	7466.2	7114.3	5962.4	18796.6
150	9650.2	9560.1	8408.9	7932.5	9415.7	8663.6	8208.7	6781.1	22956.8
160	11300.2	11104.3	9671.0	9094.8	11000.7	10059.4	9457.3	7694.8	28527.8
170	13330.0	13036.3	11167.5	10400.5	12739.8	11680.6	10878.8	8794.8	36243.5
180	15576.5	15185.6	12813.4	11958.1	14918.2	13608.4	12559.8	9968.0	45461.1
190	18076.2	17683.8	15055.7	13773.0	17458.3	15678.3	14552.4	11381.2	55843.7
200	21194.1	20751.2	17538.2	15970.7	20298.2	18271.4	16756.4	12907.8	71041.1
210	24861.7	24296.8	20999.1	18564.3	23707.4	21091.0	19390.3	14793.8	89502.9

Table 3. The average lengths of the first vector in reduced bases using the preprocessed input bases. Plotted in Figure 3.

6.2 Standalone Runtime

We now compare the standalone runtimes of the LLL-style algorithms in dimensions 40 to 210, without 0.99-LLL preprocessing.

Whilst the asymptotic runtime complexities (comparisons in Table 1) of our greedy global algorithms Pot-GG and SS-GG are the same as the corresponding X-DeepLLL algorithms, we observe in Table 4 that *our algorithms run in much less time than the corresponding DeepLLL algorithm* on average in every dimen-

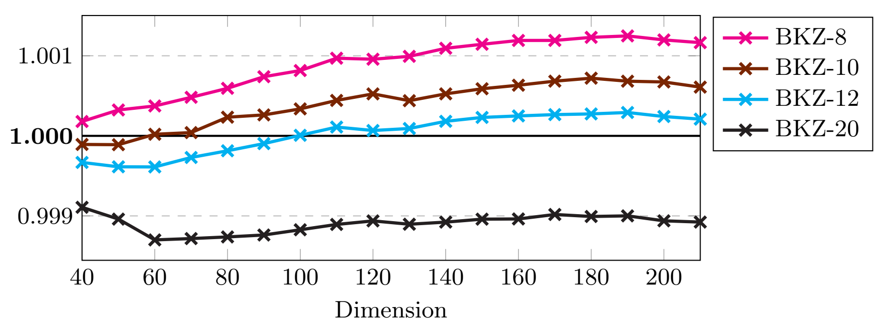

Fig. 2. The gap between the RHF of SS-GG and BKZ with block size $\beta = 8$, 10, 12 and 20, expressed as $\frac{\mathrm{RHF}_{\text{BKZ-}\beta}}{\mathrm{RHF}_{\text{SS-GG}}}$.

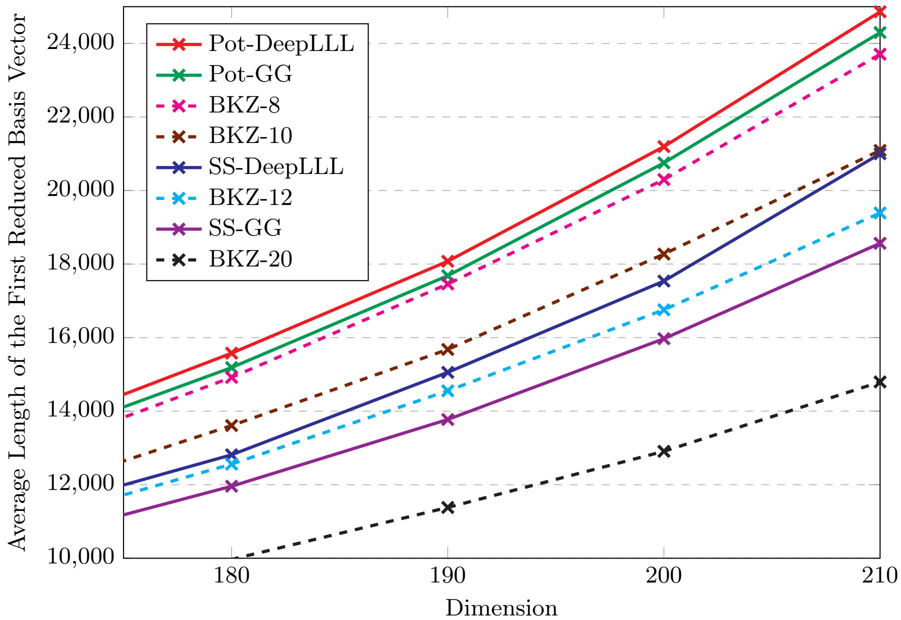

Fig. 3. The average length of the first basis vector after being reduced in dimensions 180 to 210 using the preprocessed input bases. Values taken from Table 3.

sion. Furthermore, as the dimension grows, *the greedy global algorithms become even better in comparison.* At dimension 150, SS-GG is around 2.3 times faster than SS-DeepLLL and Pot-GG is about 1.4 times faster than Pot-DeepLLL. In fact, *SS-GG is only second to LLL in standalone runtime*, as is quite clear in Figure 4, being around 3.41 times slower at dimension 150.

A more granular investigation of the standalone runtime is conducted using the numbers of reorderings (deep insertions) $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ and size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ performed by each algorithm. At dimension 150, the number of deep insertions of Pot-GG is only 0.87% of Pot-DeepLLL, and the number of deep insertions of SS-GG is only 1.87% of SS-DeepLLL, as reported in Table 8.

Remark 9. The comparisons of the number of reorderings in LLL-style algorithms provide strong intuitive justification for our greedy global approach in terms of improving 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-GG algorithm, there is more that needs to be done compared to 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 analysis of efficiency for fairer comparison between the LLL-style algorithms.

In Table 9, we see that at dimension 150 the average number of size reductions (Algorithm 1 [Step 3]) of Pot-GG is 61.68% of Pot-DeepLLL, and that of SS-GG is 68.13% of SS-DeepLLL. These percentages are higher than the percentages of deep insertions mentioned above, because of the reasons explained in Remarks 8 and 9. However, the decrease in the number of deep insertions and iterations is so prominent, that the additional operations after every deep insertion is well compensated. In summary, *X-GG requires significantly fewer deep insertions and overall fewer size reductions in standalone comparison than X-DeepLLL.* Hence,the overall standalone runtime of the X-GG algorithms is much better than the X-DeepLLL algorithms.

	Algorithm				
Dim	LLL	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	2.02	14.2	6.37	8.54	2.19
50	5.74	48.1	20.7	25.3	6.06
60	12.6	136	59.1	63.2	15.1
70	22.7	295	134	124	31.1
80	41.9	645	301	244	63.2
90	85.0	1323	649	490	120
100	148	3022	1555	954	275
110	248	5644	3043	1777	497
120	348	8918	5130	2479	816
130	580	15929	9865	4103	1695
140	831	25617	16886	6268	2554
150	1134	37328	26738	8752	3867

Table 4. Average runtime in seconds (rounded to most significant 3 digits for smaller values) for the standalone algorithms. Plotted in Figure 4.

6.3 Runtime with LLL Preprocessing

Here we compare the runtime of the algorithms in dimensions 40 to 210 using bases that have been 0.99-LLL reduced using FPLLL [46].

Our preprocessed runtime results are in contrast with the standalone runtime comparison from Section 6.2. We see from Table 5 (Figure 5) that *our algorithms have a slower runtime than the corresponding DeepLLL algorithm* on the 0.99-LLL reduced bases used as input. Furthermore, as the dimension grows, the gap between X-DeepLLL and X-GG slowly increases. However, the greedy global algorithms are still quite efficient in practice; reducing the input bases at dimension 210 in less than 17 seconds on average.

As before, we consider the subroutines within each iteration to conduct a granular analysis of the runtime differences between X-DeepLLL and X-GG. We consider the numbers of basis reorderings (deep insertions) and size reductions performed by each of the algorithms. The average number of reorderings in the

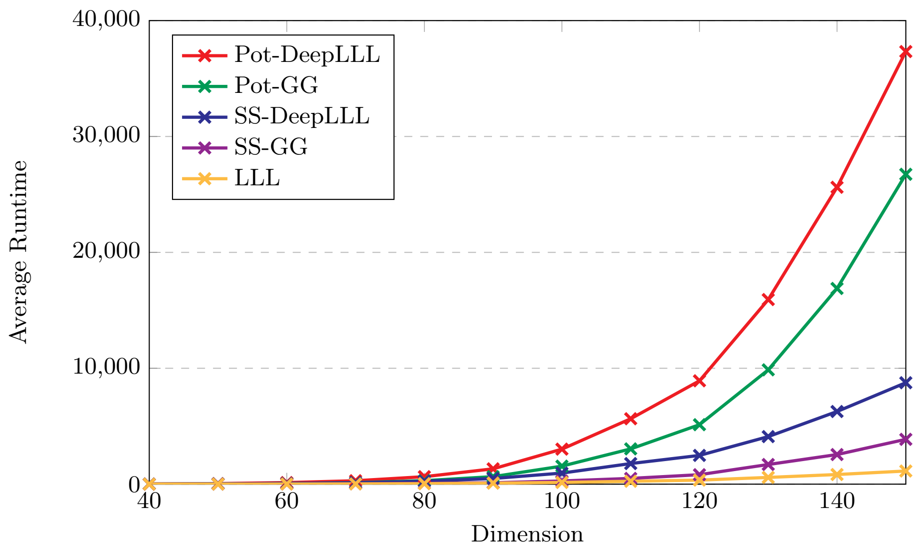

Fig. 4. The average runtime of the standalone algorithms in dimensions 40 to 150. Values taken from Table 4.

LLL-style algorithms is provided in Table 10 (Figure 8). As in the standalone comparison, across all tested dimensions, *the greedy global algorithms perform fewer deep insertions than their DeepLLL counterparts*. It is also interesting to note here that Pot-GG performed *fewer reorderings* than SS-GG on average across all tested dimensions.

As pointed out in Remark 9, the average number of size reductions (Algorithm 1 [Step 3]) is a more granular indicator of the runtime. Table 10 also provides the average number of size reductions. Across all tested dimensions *the greedy global variants always perform more size reductions than their DeepLLL counterparts*. Figure 9 shows the diverging curves of X-GG and the respective X-DeepLLL counterparts showing that *the difference in the number keeps growing as the dimension increases*. Unlike the standalone performance of the algorithms, the reduction in the number of reorderings in the preprocessed performance of the greedy global framework has not been able to compensate for the increase in its number of size reductions. This, along with the global index search step, appears to be the reason X-GG is slower than X-DeepLLL for preprocessed bases using standard data type implementations.

When comparing with BKZ, we see that *both SS-GG and Pot-GG are faster than BKZ using the* `long double` *data type for GSO information for all* 4 *block sizes* in these dimensions. Since X-DeepLLL is faster than X-GG on the preprocessed bases, we also have that Pot-DeepLLL and SS-DeepLLL are also faster than BKZ. At dimension 150, BKZ-8, 10, 12 and 20 took around 6.9, 7.9, 8.7 and 28.1 times longer than SS-GG respectively. At dimension 210, BKZ-8, 10, 12 and 20 took around 4.5, 5.2, 5.9 and 22.1 times longer than SS-GG respectively.

	Algorithm							
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	BKZ-8	BKZ-10	BKZ-12	BKZ-20
40	0.00146	0.00287	0.00134	0.00165	0.0276	0.0301	0.0336	0.0582
50	0.00753	0.00377	0.00316	0.00386	0.0743	0.0835	0.0931	0.166
60	0.00953	0.0185	0.00783	0.00915	0.165	0.184	0.214	0.426
70	0.0220	0.0409	0.0172	0.0209	0.347	0.381	0.427	0.927
80	0.0439	0.0861	0.0352	0.0438	0.684	0.733	0.828	1.96
90	0.0800	0.155	0.0632	0.0824	1.23	1.33	1.48	3.62
100	0.141	0.286	0.111	0.155	2.01	2.21	2.48	6.80
110	0.232	0.463	0.184	0.276	3.02	3.42	3.90	11.6
120	0.364	0.742	0.287	0.468	4.48	5.33	5.83	17.7
130	0.545	1.18	0.435	0.775	6.72	7.36	8.32	26.5
140	0.814	1.75	0.651	1.23	9.82	10.6	11.5	39.2
150	1.17	2.55	0.94	1.94	13.4	15.3	16.9	54.6
160	1.65	3.68	1.34	3.01	20.7	24.3	27.1	88.2
170	2.26	5.23	1.85	4.44	27.3	30.0	34.3	125
180	3.04	7.10	2.46	6.37	34.8	38.5	42.6	172
190	4.00	9.79	3.18	8.99	46.1	49.7	55.6	226
200	5.36	13.2	4.16	12.6	58.8	64.1	72.3	285
210	6.73	16.8	4.84	16.5	74.7	85.2	97.8	364

Table 5. Average runtime in seconds (rounded to most significant 3 digits) on the preprocessed bases. Plotted in Figure 5.

Summary. We summarise the comparison between SS-GG with $\delta = 1-10^{-6}$ and BKZ with $\beta = 8, 10, 12$, using preprocessed bases in dimensions 40 to 210. Figure 5 shows that the SS-GG is significantly faster than BKZ with $\beta = 8, 10, 12$, and keeps getting better with increasing dimensions. Figures 1, 2 show that SS-GG provides better RHF than BKZ-8 throughout, BKZ-10 from around dimension 60 and BKZ-12 from around dimension 100. So SS-GG is significantly faster than BKZ with $\beta = 8, 10, 12$ while simultaneously also providing better RHF in most cases.

6.4 Higher Dimension Comparisons with BKZ

Finally, we present the comparison of SS-GG and BKZ with block sizes $5, 8, 10, 12, 14, 18, 20$ and 21 at dimensions $250, 300, 350, 400, 450, 500, 550$ and 600 using their respective MPFR implementations. The runtimes and RHFs for SS-GG and BKZ are provided in Table 6. If a table reads '-', the runtime was too long to feasibly complete the reduction, and so no data has been generated. If an entry is italicised, only 5 bases were used to generate the average instead of the usual 10 due to the increased runtime.

The runtime of BKZ-β is known to increase dramatically at around $\beta = 20$ to 25 and beyond [19], which is confirmed by our experimental results.

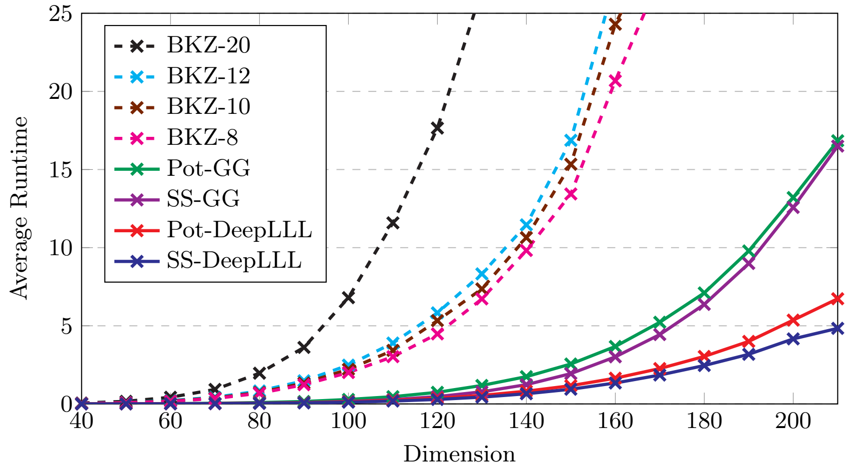

Fig. 5. The average runtime of lattice basis reduction algorithms in dimensions 40 to 210 using the preprocessed input bases. Values taken from Table 5.

Varying δ. The threshold δ determines the minimal change in the squared sum (SS) in each iteration of SS-DeepLLL [48] and our SS-GG algorithm. For the smaller dimension tests, our choice of $\delta = 1 - 10^{-6}$ for SS is based on the following arguments from [48] surrounding SS-DeepLLL. They ran experiments on SS-DeepLLL for $n \leq 150$ with $\delta = 1 - 10^{-\varepsilon}$ for $\varepsilon \in \{4, 6, 8\}$ as well as $\delta = 1$. In [48, Section 4.3.1], it was noted that the output quality of SS-DeepLLL with $\delta = 1 - 10^{-6}$ is almost equal to that with $\delta = 1$. Using $\delta = 1$ would substantially increase the runtime of SS-DeepLLL (and SS-GG) with no appreciable improvement of the RHF. So we ran our experiments with $\delta = 1 - 10^{-6}$ for SS at smaller dimensions. However, this observation does not hold for larger dimensions.

At higher dimensions, our observation with $\delta = 1 - 10^{-6}$ is as follows. A deep insertion $\sigma_{i,k}(\mathbf{B})$ would not happen if it does not satisfy $\mathrm{SS}(\sigma_{i,k}(\mathbf{B})) < \delta \cdot \mathrm{SS}(\mathbf{B})$, even though the *absolute value* of the potential decrease $\Delta\mathrm{SS}(\mathbf{B}) = \mathrm{SS}(\mathbf{B}) - \mathrm{SS}(\sigma_{i,k}(\mathbf{B}))$ is large and increases with the dimension. In other words, *large improvements of SS are restricted by small values of δ at higher dimensions.* For example, a 0.99-LLL preprocessed basis in our experiments has SS in the order of 10^{11} for $n = 250$, and in the order of 10^{17} for $n = 600$. At $n = 250$ and 600, an insertion must reduce SS by at least 10^5 and 10^{11} respectively, if $\delta = 1 - 10^{-6}$ is used. This warrants the use of larger values of δ.

We run our higher dimension tests with $\delta = 1 - 10^{-\varepsilon}$ for $\varepsilon \in \{6, 7, 8, 9\}$ and $\delta = 1$ to progressively improve the RHF. Table 6 shows that using $\delta = 1 - 10^{-6}$ in SS-GG yields significantly weaker reduction than higher values of $\delta \leq 1$. On the other hand, in order to see if the time advantage seen in the smaller dimensions can be achieved, albeit with a weaker reduction, we have included experiments for smaller values of $\delta = 1 - 10^{-\varepsilon}$ for $\varepsilon \in \{4, 5\}$.

		DIMENSION							
δ for SS-GG		**250**	**300**	**350**	**400**	**450**	**500**	**550**	**600**
$\mathbf{1-10^{-4}}$	Runtime	107	225	400	629	984	2583	3256	4319
	RHF	1.01711	1.01774	1.01810	1.01837	1.01839	1.01831	1.01822	1.01804
$\mathbf{1-10^{-5}}$	Runtime	368	850	1489	2243	3358	8764	10666	13230
	RHF	1.01534	1.01594	1.01656	1.01701	1.01728	1.01730	1.01735	1.01731
$\mathbf{1-10^{-6}}$	Runtime	1075	2786	5639	10117	14541	35411	37188	39706
	RHF	1.01409	1.01468	1.01526	1.01570	1.01615	1.01650	1.01675	1.01687
$\mathbf{1-10^{-7}}$	Runtime	1685	4955	11402	22667	38816	114861	144164	177719
	RHF	1.01345	1.01384	1.01424	1.01461	1.01509	1.01539	1.01582	1.01615
$\mathbf{1-10^{-8}}$	Runtime	2432	9475	27056	59988	121723	380481	544791	780430
	RHF	1.01323	1.01334	1.01351	1.01376	1.01414	1.01453	1.01490	1.01521
$\mathbf{1-10^{-9}}$	Runtime	2615	13551	55517	151765	351711	*1220147*	*1787847*	-
	RHF	1.01318	1.01312	1.01300	1.01310	1.01340	*1.01375*	*1.01409*	-
1.0	Runtime	2935	17718	108887	625399	-	-	-	-
	RHF	1.01306	1.01293	1.01260	1.01237	-	-	-	-
BKZ		**DIMENSION**							
block size, β		**250**	**300**	**350**	**400**	**450**	**500**	**550**	**600**
$\beta=\mathbf{5}$	Runtime	629	1472	3141	5448	7902	11582	16346	23274
	RHF	1.01643	1.01655	1.01653	1.01668	1.01663	1.01658	1.01644	1.01632
$\beta=\mathbf{8}$	Runtime	998	2406	4997	10118	16442	23648	31771	46201
	RHF	1.01527	1.01515	1.01533	1.01530	1.01525	1.01533	1.01527	1.01512
$\beta=\mathbf{10}$	Runtime	1232	2756	6196	10571	17475	24409	33173	48057
	RHF	1.01465	1.01465	1.01468	1.01482	1.01475	1.01474	1.01474	1.01461
$\beta=\mathbf{12}$	Runtime	1264	3008	6528	13657	23422	33204	43655	63323
	RHF	1.01426	1.01425	1.01422	1.01428	1.01427	1.01429	1.01430	1.01416
$\beta=\mathbf{14}$	Runtime	1572	3943	8551	16076	28066	41677	55669	76927
	RHF	1.01382	1.01386	1.01375	1.01386	1.01392	1.01395	1.01388	1.01380
$\beta=\mathbf{18}$	Runtime	3470	8378	17652	33558	55691	100085	133628	175322
	RHF	1.01307	1.01319	1.01317	1.01323	1.01333	1.01324	1.01328	1.01326
$\beta=\mathbf{20}$	Runtime	7944	19257	41966	102322	205247	*316521*	*627868*	*804470*
	RHF	1.01283	1.01285	1.01297	1.01290	1.01295	*1.01296*	*1.01299*	*1.01294*
$\beta=\mathbf{21}$	Runtime	33057	169574	546368	1988942	-	-	-	-
	RHF	1.01260	1.01260	1.01266	1.01265	-	-	-	-

Table 6. The average runtime (to the nearest second) and the RHF of SS-GG with various values of δ and BKZ with various block sizes β in dimensions 250-600. Plotted in Figures 6 and 7.

RHF and Runtime. The RHF and runtime are presented in Table 6, and Figures 6 and 7. Our first observation is that while the RHF of BKZ remains almost constant as the dimension increases (previously noted by Gama and Nguyen [19] for $n \leq 200$ in all lattice reduction algorithms they tested including DeepLLL), the RHF of SS-GG is not constant. It varies with the dimension n and the reduction parameter δ. For $\delta = 1 - 10^{-\varepsilon}, \varepsilon = 4, 5, 6, 7, 8$, Figure 7 and Table 6 show that the RHF keeps increasing with the dimension. For $\delta = 1 - 10^{-9}$, the RHF first decreases (output quality improves), and then increases. For $\delta = 1$, the RHF is always decreasing (tested until $n = 400$ due to the huge runtime). This is in sharp contrast to BKZ, where no matter the block size β, the output

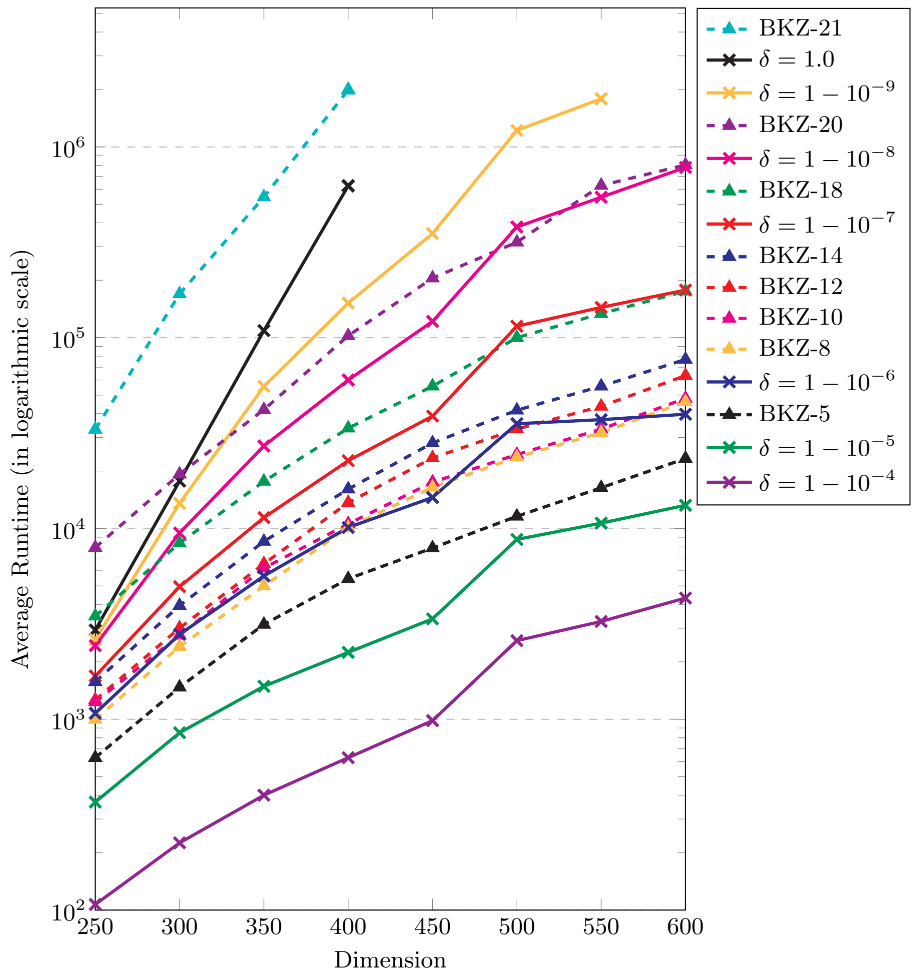

Fig. 6. The average runtime (in logarithmic scale) of SS-GG with various values of δ and BKZ with various block sizes β in dimensions 250-600. Values taken from Table 6.

quality remains reasonably consistent as the dimension increases. The runtimes of SS-GG and BKZ are both increasing with the dimension.

For the smaller values of δ (namely $1 - 10^{-4}$ and $1 - 10^{-5}$), the runtime of SS-GG is consistently faster than small block size BKZ. In particular, SS-GG with these values of δ is *faster than BKZ-5 in all dimensions 250-600*. The corresponding RHF increases initially until around dimension 400–450, before stabilising, and in the case of $\delta = 1 - 10^{-4}$, starts decreasing again. The RHF for $\delta = 1 - 10^{-4}$ is always worse than BKZ-5, but due to its superior runtime, indicates that *SS-GG with $\delta = 1 - 10^{-4}$ achieves a middle ground (in runtime and RHF) between LLL and BKZ-*5. However, for $\delta = 1 - 10^{-5}$, SS-GG has

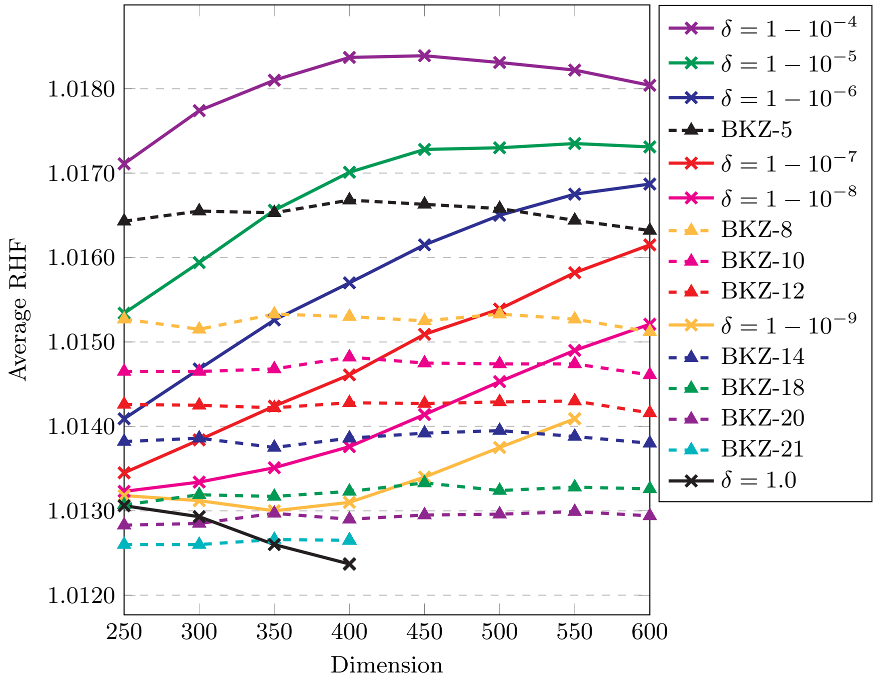

Fig. 7. The average RHF of SS-GG with various values of δ and BKZ with various block sizes β in dimensions 250-600. Values taken from Table 6.

smaller RHF than BKZ-5 in dimensions 250 and 300, and is roughly the same in dimension 350. So *SS-GG outperforms BKZ-5 in dimensions 250 – 300 in both RHF and runtime.* Also, in dimension 350, *SS-GG can achieve the same RHF as BKZ-5, whilst taking less than half the time on average.*

When $\delta = 1-10^{-6}, 1-10^{-7}, 1-10^{-8}$, the runtime of δ-SS-GG ranges between BKZ-$\beta_1, \ldots, \beta_2$ as follows:

– $\beta = 8, \ldots, 12$ for $\delta = 1 - 10^{-6}$,
– $\beta = 14, \ldots, 18$ (except for dimensions 500-550) for $\delta = 1 - 10^{-7}$, and
– $\beta = 18, \ldots, 20$ for $\delta = 1 - 10^{-8}$.

SS-GG with $\delta = 1 - 10^{-6}$ provides significantly better RHF than BKZ-8 up to $n = 350$ (with similar runtime), and BKZ-5 up to $n = 500$ (while being slightly slower). In general, since it is faster (except for dimension 500) but has worse RHF than BKZ-12 in dimensions 250-600, we posit that *SS-GG with* $\delta = 1 - 10^{-6}$ *achieves a middle ground between LLL and BKZ-12 in terms of runtime and RHF on output.*

Finally, for $\delta = 1$, we see that the RHF of SS-GG is consistently decreasing while the runtime is sharply increasing, crossing BKZ-20 between $n = 300, 350$ for both. So *SS-GG outperforms BKZ-20 in RHF soon after* $n = 300$*, at the cost of an increased runtime.* In comparison to BKZ-21, we see that *the runtime of*

SS-GG with $\delta = 1$ *is significantly faster than that of BKZ-21 in all dimensions 250 – 400 we have tested.* As well as being an improvement in runtime, *SS-GG with* $\delta = 1$ *also achieves a similar RHF to BKZ-21 at dimension 350, and a significantly better RHF at dimension 400.* However, this simultaneous improvement of runtime and RHF over BKZ-β may not continue for larger dimensions or block sizes $\beta > 21$. From Figure 6 it seems that at some higher dimension beyond 400, SS-GG (with $\delta = 1$) may become slower than BKZ-21. From Figure 7, it also seems that the RHF of SS-GG (with $\delta = 1$) may continue to improve over BKZ-21, perhaps at the cost of a slower runtime at very large dimensions. It is likely that there is some large block size β for which SS-GG (with $\delta = 1$) will not be able to achieve a better RHF at any dimension, but may still be faster. We leave such detailed investigations as future work.

	DIMENSION							
	250	**300**	**350**	**400**	**450**	**500**	**550**	**600**
SS-GG	84	93	104	112	123	132	141	151
BKZ	74	87	99	106	113	123	128	133

Table 7. Precisions used for the `FP_NR` multi-precision floating-point datatype for SS-GG and BKZ for dimensions 250-600.

The values of precision used at the respective dimensions are provided in Table 7, and one may note that the precision required by SS-GG is more than BKZ in these tests. We note here that there is a large jump in the runtime of SS-GG between dimension 450 and 500. One reason for this is the need to use non-standard large integer datatypes to correctly reduce the bases. This provides context not only for this sharp runtime increase, but also for the smaller runtime differences between dimensions 500 and 600 for the smaller values of δ.

Inner Workings of δ-SS-GG and BKZ-β. It is clear from Figure 7 that SS-GG and BKZ are very different algorithms in the way they reduce a basis. Nevertheless, if one has to compare them, a deep insertion of SS-GG would be analogous to an SVP call in BKZ. Similarly, the average depth of deep insertions in SS-GG would be analogous to the block size in BKZ. In Table 11 and Figures 10 and 11, we provide data on the average number of insertions and depth of insertion performed by SS-GG in dimensions 250-600.

In order to compute the number of SVP calls in dimension β, we ran the FPLLL [46] implementation of BKZ with the `BKZ_VERBOSE` flag which, along with other information, provides the number of tours performed. The number of SVP calls (with dimension β) performed during the algorithm is given by

$$\#\text{ SVP Calls at Dim } \beta \text{ for BKZ-}\beta = (n - \beta + 1) \cdot (\#\text{ of Tours})$$

as there are $(n - \beta + 1)$ different blocks of β consecutive basis vectors. These vectors are then projected, and an SVP algorithm is called on them, which finds a short vector to be inserted into the basis. We note that at the end of a tour, BKZ then reduces the size of the window so that it comprises one fewer vector with each call to the SVP oracle. The final SVP call is performed only on the projected block generated by the pair of vectors $(\mathbf{b}_{n-1}, \mathbf{b}_n)$. We do not include the SVP calls in dimension $< \beta$ in our analysis. Our observations from the above comparison are as follows.

– While the number of insertions vary significantly with δ, the number of SVP calls hardly change between $\beta = 10$ and $\beta = 12$.
– With increasing dimension, the number of insertions does not increase as much as one would expect, and even decreases for smaller δ.
– As δ increases, SS-GG allows for much smaller insertion depths. This results in significant improvement in quality compared with the smaller values of δ. On the other hand, the block size β is fixed for an SVP call in BKZ. *In SS-GG, the shallower insertions lead to better RHF. In BKZ, the larger block size leads to better RHF.*

These observations are further evidence that SS-GG and BKZ are very different algorithms.

We also note that since the volume of a lattice is an invariant, for an output basis, the lengths of its first vector $\|\mathbf{b}_1\|$ and the final GSO vector $\|\mathbf{b}_n^*\|$ is indicative of the balance in the lengths of its GSO vectors. This notion of balance is similar to the *spread of the profile* of the basis, which is the difference between the maximum and minimum values of $\|\mathbf{b}_i^*\|$ [32,25,40]. Table 12 shows how the (squared) length of the final GSO vector evolves with the dimension for SS-GG with $\delta = 1 - 10^{-6}$ and $1 - 10^{-8}$, as well as for BKZ-12 and BKZ-20. In all cases, we observe that $\|\mathbf{b}_n^*\|$ *for an SS-GG-reduced basis is shorter than that of a BKZ-reduced basis.* Furthermore, note that the RHF of SS-GG with $\delta = 1 - 10^{-8}$ is smaller than that of BKZ-12 until dimension 500, and is larger than that of BKZ-20 in every dimension. Therefore, *this observation holds regardless of whether the first vector in the SS-GG reduced basis is shorter.* This implies that in general, as we proceed through the basis, the GSO vectors decrease in length more in SS-GG than in BKZ. This further highlights the differences between SS-GG and BKZ in the way that basis reduction is performed.

7 Conclusion

In this work, we first interpreted the (generalised) Lovász condition [43] as a reordering constraint used to improve the length $\|\mathbf{b}_i^*\|$ of the i^{th} GSO vector, which is a localised measure of the quality of the basis. We thus arrived at a coherent generalised representation of DeepLLL [43], Pot-DeepLLL [16] and SS-DeepLLL [48] algorithms where they iteratively improve a local or global measure of basis quality. This generalisation leads us to the new greedy global framework in the form of a generic algorithm X-GG for lattice basis reduction. The algorithm works by iteratively decreasing a general measure X of basis quality. The key novelty in the framework is the dynamic greedy choice of a pair of indices for deep insertion globally from the whole basis such that the basis quality measure X is minimised. Our framework is instantiated by substituting the general measure X with the concrete quality measures potential (Pot) and squared sum (SS). We reckon that the framework would be able to easily utilise other global measures (like the `Eval` function of [45]). A local measure like $\mathrm{LC}_i(\mathbf{B})$ or the one from G6K [4] provides independent values for different indices i. A vector measure like the profile [32,25,40] is constituted of such local measures of the (whole) basis. More careful consideration must be taken to construct a GG algorithm using such local or vector measures. As long as a measure X can be computed from the values of $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$, the index search of X-GG is the only step that may need to be customised based on the choice of X. We have proved results on the efficiency of the generic algorithm X-GG and on the two new concrete algorithms Pot-GG and SS-GG. Furthermore, we have shown that the bases produced by our algorithms are of provable quality i.e. X-DeepLLL reduced. Our implementations are public.

Using multi-precision arithmetic implementations for standalone comparison between the algorithms (without preprocessing the bases), our GG variants have a faster runtime than their DeepLLL counterparts, whilst also outperforming them in terms of basis quality. In fact, SS-GG is only second to LLL in standalone runtime, while of course providing much shorter vectors. We provide justifications for the runtime comparisons based on more granular runtime parameters like the number of reorderings and the number of size reductions.

All deep insertion based algorithms (DeepLLL, X-DeepLLL and X-GG) can be improved using the Cholesky Factorisation Algorithm (CFA) [34] and the lazy size reduction algorithms [5,6,34]. These two techniques have been used in L^2 [34] to improve LLL and are key to its correctness. These techniques have not yet been incorporated into deep insertion based algorithms. Instead, we have introduced the GSO recomputation step to ensure correctness of the outputs of our X-GG implementations. Using the techniques from [34] would be a significant contribution to the theory and practice of lattice reduction algorithms using deep insertions.

The design principle of X-GG 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 improve the output quality.

The theoretical runtime analysis of the greedy global framework significantly overestimates the number of iterations of the algorithm. A different proof mechanism can take into account the greedy choice of indices, to improve the theoretical bound on runtime which is closer to its practical performance. Utilising reduction on sublattices like [40], restricting insertions to fixed blocks akin to the technique in [43], or using a sliding window of sublattices like BKZ [43] may lead to more efficient algorithms. Since SS-GG with $\delta = 1 - 10^{-6}$ is superior (in both RHF and runtime) to BKZ-12 in dimensions 100 to 250, it may be particularly useful on sublattices of these dimensions; perhaps as a subroutine within larger dimension BKZ much like the use of DeepLLL in [19]. Running SS-GG with a smaller value of δ in high dimension reduces the basis to a better quality than LLL very quickly. This may allow the subsequent series of BKZ with increasing block size to run quicker, resulting in a technique which performs stronger reduction faster overall. We leave such explorations for future work with the belief that our framework has opened up avenues for designing interesting new lattice reduction algorithms and their analyses.

8 Declarations

Data Availability. The datasets generated during and/or analysed during the current study are available in the GG-LLL GitHub repository [7].

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. Albrecht, M.R., Bai, S., Fouque, P.A., Kirchner, P., Stehlé, D., Wen, W.: Faster enumeration-based lattice reduction: Root hermite factor $k^{1/(2k)}$ time $k^{k/8+o(k)}$. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020. pp. 186–212. Springer International Publishing, Cham (2020). https://doi.org/https://doi.org/10.1007/978-3-030-56880-1_7
3. Albrecht, M.R., Curtis, B.R., Deo, A., Davidson, A., Player, R., Postlethwaite, E.W., Virdia, F., Wunderer, T.: Estimate all the {LWE, NTRU} schemes! In: Catalano, D., De Prisco, R. (eds.) Security and Cryptography for Networks. pp. 351–367. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-98113-0_19
4. Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E.W., Stevens, M.: The general sieve kernel and new records in lattice reduction. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019. pp. 717–746. Springer International Publishing, Cham (2019). https://doi.org/https://doi.org/10.1007/978-3-030-17656-3_25
5. Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem (shortened version). In: Proceedings of the 2nd Symposium of Theoretical Aspects of Computer Science. p. 13–20. STACS '85, Springer-Verlag, Berlin, Heidelberg (1985). https://doi.org/https://doi.org/10.1007/BFb0023990
6. Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica **6**, 1–13 (mar 1986). https://doi.org/https://doi.org/10.1007/bf02579403
7. Bhattacherjee, S., Moyler, J.: Our implementations of some LLL-style algorithms (2024), https://github.com/GG-LLL/Greedy-Global-LLL
8. 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
9. 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). https://doi.org/https://doi.org/10.1090/s0025-5718-2012-02545-2

10. Chen, J., Stehlé, D., Villard, G.: Computing an LLL-reduced basis of the orthogonal lattice. In: Arreche, C. (ed.) 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
11. Chen, Y., Nguyen, P.Q.: BKZ2.0: Better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Seoul, South Korea (Dec 2011). https://doi.org/10.1007/978-3-642-25385-0_1
12. Cohen, H.: A course in computational algebraic number theory. Springer (1993). https://doi.org/https://doi.org/10.1007/978-3-662-02945-9
13. Coppersmith, D.: Finding a small root of a bivariate integer equation; factoring with high bits known. In: Maurer, U. (ed.) Advances in Cryptology — EUROCRYPT '96. pp. 178–189. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/3-540-68339-9_16
14. Coppersmith, D.: Finding a small root of a univariate modular equation. In: Maurer, U. (ed.) Advances in Cryptology — EUROCRYPT '96. pp. 155–165. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/3-540-68339-9_14
15. Darmstadt T.U.: SVP challenge. https://www.latticechallenge.org/svp-challenge/
16. 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
17. Fukase, M., Kashiwabara, K.: An accelerated algorithm for solving SVP based on statistical analysis. J. Inf. Process. **23**, 67–80 (2015). https://doi.org/10.2197/ipsjjip.23.67
18. Galbraith, S.D.: Mathematics of public key cryptography. Cambridge University Press, USA, 1st edn. (2012). https://doi.org/https://doi.org/10.1017/CBO9781139012843
19. 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
20. 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
21. Hanrot, G.: LLL: A tool for effective Diophantine approximation, pp. 215–263. Vol. 1 of Nguyen and Vallée [37] (2009). https://doi.org/10.1007/978-3-642-02295-1_6
22. Hanrot, G., Lefèvre, V., Pélissier, P., Théveny, P., Zimmermann, P.: The GNU MPFR library (2023), available at https://www.mpfr.org/index.html
23. Howgrave-Graham, N.: Finding small roots of univariate modular equations revisited. In: Darnell, M. (ed.) Crytography and Coding. pp. 131–142. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0024458
24. Howgrave-Graham, N.A.: Isodual reduction of lattices. Cryptology ePrint Archive, Paper 2007/105 (2007), https://eprint.iacr.org/2007/105
25. 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
26. Klüners, J.: The van Hoeij algorithm for factoring polynomials, pp. 283–291. Vol. 1 of Nguyen and Vallée [37] (2009). https://doi.org/10.1007/978-3-642-02295-1_8

27. 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
28. 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 2019). https://doi.org/10.1007/978-3-030-34621-8_3
29. Lenstra, A.K., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**, 515–534 (1982). https://doi.org/10.1007/BF01457454
30. 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). https://doi.org/https://doi.org/10.1007/978-3-0348-8268-2_3
31. Morel, I., Stehlé, D., Villard, G.: H-LLL: using householder inside LLL. In: May, J.P. (ed.) 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
32. Neumaier, A., Stehlé, D.: Faster LLL-type reduction of lattice bases. In: Rosenkranz, M. (ed.) 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
33. Nguyen, P.Q., Stehlé, D.: LLL on the average. In: Hess, F., Pauli, S., Pohst, M. (eds.) 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
34. Nguyen, P.Q., Stehlé, D.: An LLL algorithm with quadratic complexity. SIAM Journal on Computing **39**(3), 874–903 (2009). https://doi.org/10.1137/070705702
35. Nguyen, P.Q., Stehlé, D.: Low-dimensional lattice basis reduction revisited. ACM Trans. Algorithms **5**(4) (nov 2009). https://doi.org/10.1145/1597036.1597050, https://doi.org/10.1145/1597036.1597050
36. 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
37. Nguyen, P.Q., Vallée, B.: The LLL algorithm. Springer (2010)
38. Odlyzko, A.: Cryptanalytic attacks on the multiplicative knapsack cryptosystem and on Shamir's fast signature scheme. IEEE Transactions on Information Theory **30**(4), 594–601 (1984). https://doi.org/10.1109/TIT.1984.1056942
39. 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
40. Ryan, K., Heninger, N.: Fast practical lattice reduction through iterated compression. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023. pp. 3–36. Springer Nature Switzerland, Cham (2023). https://doi.org/https://doi.org/10.1007/978-3-031-38548-3_1
41. Schneider, M., Buchmann, J.: Extended lattice reduction experiments using the BKZ algorithm. In: Sicherheit 2010. Sicherheit, Schutz und Zuverlässigkeit, pp. 241–251. Gesellschaft für Informatik e.V., Bonn (2010)
42. Schneider, M., Buchmann, J., Lindner, R.: Probabilistic analysis of LLL reduced bases. In: Buchmann, J.A., Cremona, J., Pohst, M.E. (eds.) Algorithms and Number Theory. Dagstuhl Seminar Proceedings (DagSemProc), vol. 9221, pp. 1–6.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2009). https://doi.org/10.4230/DagSemProc.09221.4
43. 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
44. Shoup, V.: NTL: A library for doing number theory (2021), available at https://github.com/libntl/ntl
45. Teruya, T., Kashiwabara, K., Hanaoka, G.: Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem. In: Abdalla, M., Dahab, R. (eds.) Public-Key Cryptography – PKC 2018. pp. 437–460. Springer International Publishing, Cham (2018). https://doi.org/https://doi.org/10.1007/978-3-319-76578-5_15
46. The FPLLL development team: FPLLL, a lattice reduction library, Version: 5.4.4 (2023), available at https://github.com/fplll/fplll
47. 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 International Publishing, Cham (2018). https://doi.org/https://doi.org/10.1007/978-3-319-76620-1_9
48. 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
49. 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

A Additional Data and Plots

	Algorithm				
Dim	LLL	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	27265	11891	301	12512	362
50	47667	22251	429	23996	517
60	73652	36332	571	39682	705
70	105302	54153	737	59628	927
80	142102	75783	914	83783	1206
90	184177	101189	1128	112048	1525
100	231595	130547	1364	144138	1938
110	282967	163159	1578	179535	2400
120	339909	199762	1887	217762	2930
130	400127	239663	2186	258387	3626
140	465034	283244	2548	301361	4429
150	535176	329958	2902	346082	5442

Table 8. Average number of basis reorderings (rounded to nearest whole number) for the standalone algorithms.

	Algorithm				
Dim	LLL	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	80917	127230	67410	143755	70706
50	185909	320840	155502	371094	164548
60	359996	669874	312213	790779	337483
70	623946	1228812	570384	1480132	630993
80	992473	2052506	963137	2522662	1106309
90	1485987	3199109	1541376	4010701	1827259
100	2123061	4732812	2353368	6044032	2903142
110	2901939	6677443	3417334	8694534	4428745
120	3859389	9122406	4894355	12063243	6553179
130	4982003	12083865	6759215	16242772	9523874
140	6281915	15604490	9173358	21403549	13473035
150	7815673	19710300	12158151	27623812	18822813

Table 9. Average number of size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ (rounded to the nearest whole number) for the standalone algorithms.

	Algorithm							
	Reorderings				**Size Reductions**			
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	211	71	247	83	2620	4927	3224	5987
50	502	122	584	143	8470	15524	10357	18917
60	983	180	1170	216	21509	37996	27524	47166
70	1716	253	2045	315	46684	81529	60196	104540
80	2740	358	3359	450	90205	161833	122865	213469
90	4058	457	4966	604	158490	284433	220329	392239
100	5881	604	7151	826	264844	490594	377831	705232
110	8042	733	9739	1110	413640	770525	610176	1200356
120	10748	907	12811	1464	624169	1190832	938518	1960126
130	13820	1138	16178	1918	893756	1792218	1384967	3104523
140	17938	1338	20411	2440	1282404	2584909	2026213	4742813
150	22354	1594	24644	3162	1754879	3619918	2845232	7133488
160	27284	1840	29193	3986	2331875	4931985	3882114	10412952
170	33425	2195	34349	4925	3086531	6751534	5235544	14816911
180	40278	2516	39459	5988	4005652	8927973	6840290	20568885
190	47565	2935	44186	7163	5058505	11673474	8599039	27791502
200	56683	3363	49229	8413	6438373	15158279	10614901	36908973
210	66134	3766	53005	9715	7998874	19132055	12410508	47824142

Table 10. Average number of reorderings and size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ for the preprocessed bases. Plotted in Figures 8 and 9.

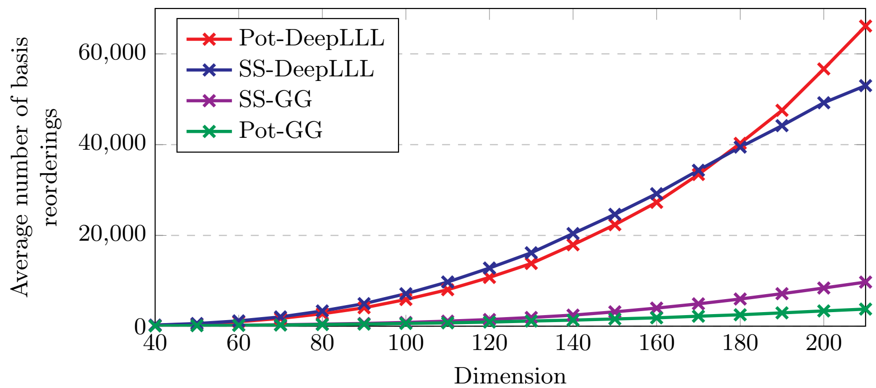

Fig. 8. The average number of basis reorderings required to reduce bases in dimensions 40-210. Values taken from Table 10.

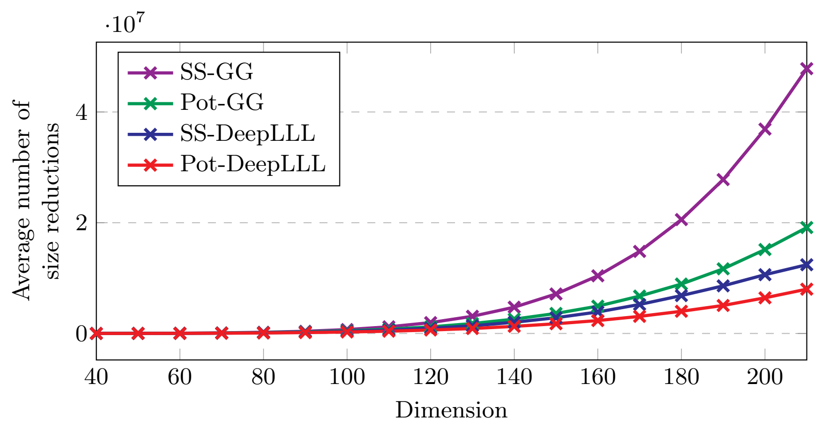

Fig. 9. The average number of individual size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ to reduce bases in dimensions 40-210 using the preprocessed input bases. Values taken from Table 10.

		DIMENSION							
δ **for SS-GG**		**250**	**300**	**350**	**400**	**450**	**500**	**550**	**600**
$\mathbf{1-10^{-4}}$	Avg. Insertions	1195	1438	1591	1682	1765	1730	1741	1783
	Avg. Depth	94.9	113.3	133.2	152.6	172.8	193.8	214.5	233.4
$\mathbf{1-10^{-5}}$	Avg. Insertions	5403	6879	7493	7514	7431	7068	6940	6454
	Avg. Depth	62.3	76.8	92.3	107.9	124.3	141.3	158.2	177.0
$\mathbf{1-10^{-6}}$	Avg. Insertions	14806	21377	26112	31413	30202	28546	23713	18741
	Avg. Depth	43.9	54.1	65.3	75.3	88.0	100.4	115.8	133.6
$\mathbf{1-10^{-7}}$	Avg. Insertions	30977	52005	75229	96636	111820	122629	124411	114532
	Avg. Depth	32.6	39.8	48.0	56.9	66.1	75.2	84.7	94.8
$\mathbf{1-10^{-8}}$	Avg. Insertions	44789	105857	188303	272046	367018	427016	494208	527263
	Avg. Depth	27.4	30.4	36.2	43.9	51.6	60.5	69.1	78.5
$\mathbf{1-10^{-9}}$	Avg. Insertions	53625	165242	413594	725029	1103952	*1445371*	*1671977*	-
	Avg. Depth	24.9	25.2	28.1	34.1	41.1	*49.2*	*58.3*	-
1.0	Avg. Insertions	62156	226633	901847	3333802	-	-	-	-
	Avg. Depth	22.8	21.7	20.6	21.1	-	-	-	-
BKZ-10	# Tours	280.3	400.7	563.5	643.5	846.1	951.0	-	-
	# SVP Calls	67552.3	116603.7	192153.5	251608.5	373130.1	466941.0	-	-
BKZ-12	# Tours	317.2	425.6	578.8	747.8	963.8	1071.9	-	-
	# SVP Calls	75810.8	122998.4	196213.2	290894.2	423108.2	524159.1	-	-

Table 11. Average number of insertions (nearest whole number) and average depth of insertions for SS-GG in dimensions 250-600 and varying δ. Number of BKZ-β tours and calls to the SVP subroutine performed in dimensions 250-500 for $\beta = 10, 12$. Plotted in Figures 10 and 11.

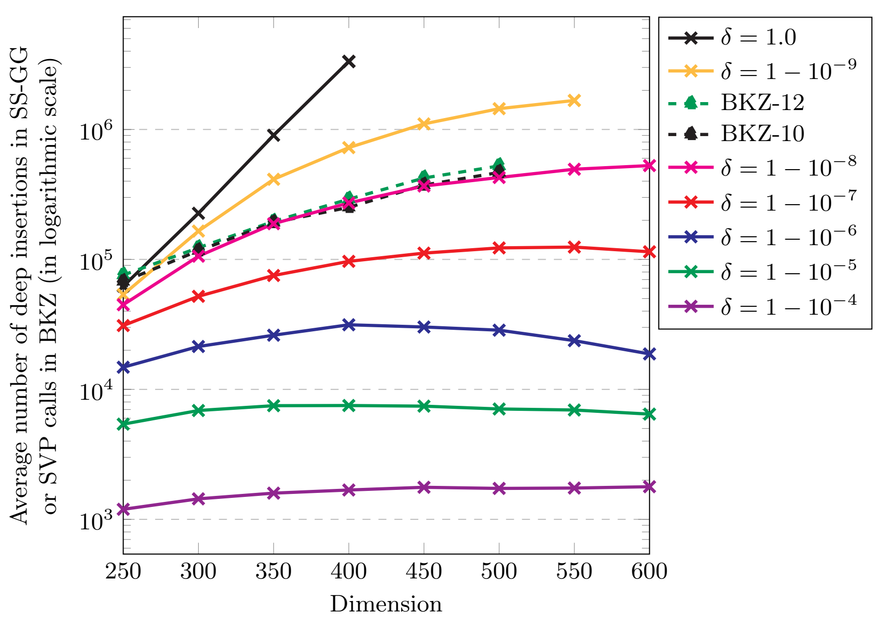

Fig. 10. The average number of deep insertions (in logarithmic scale) performed by SS-GG and SVP calls performed by BKZ-10 and BKZ-12 in dimensions 250-600 with various values of δ. Values taken from Table 11.

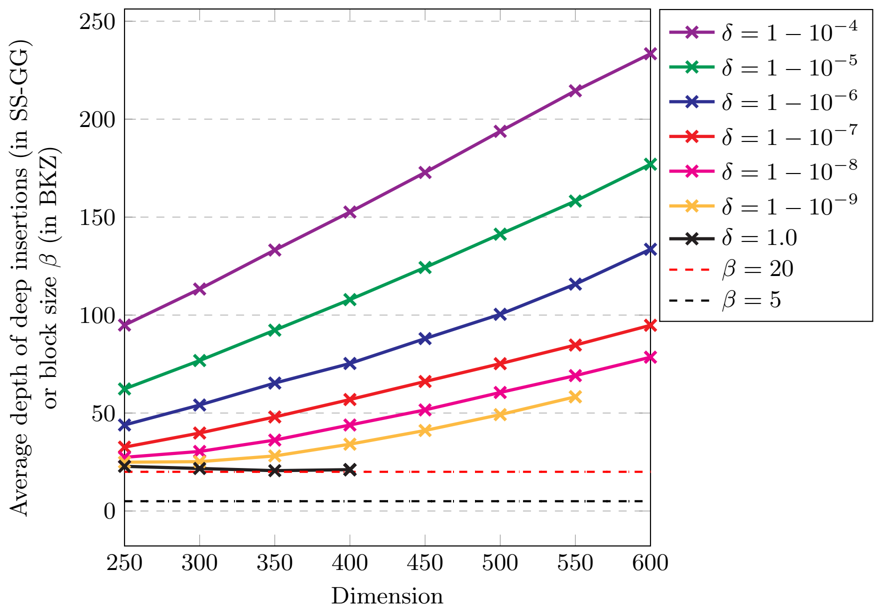

Fig. 11. The average depth of deep insertions performed by SS-GG in dimensions 250-600 with various values of δ. Values taken from Table 11. The two straight dashed lines at the bottom correspond to the BKZ block sizes $\beta = 20, 5$. Values taken from Table 11.

	δ-SS-GG		**BKZ-β**	
DIM	$\delta = 1 - 10^{-6}$	$\delta = 1 - 10^{-8}$	$\beta = 12$	$\beta = 20$
250	121.9	501.4	884.2	1453.7
300	11.39	68.91	204.8	384.1
350	0.9458	6.188	50.04	112.6
400	0.1014	0.6671	11.40	28.63
450	0.0091	0.0555	2.918	7.377
500	0.0011	0.0052	1.000	2.163

Table 12. Evolution of the squared length of the final GSO vector $\|\mathbf{b}_n^*\|^2$ for $n = 250, \ldots, 500$ for δ-SS-GG ($\delta = 1 - 10^{-6}, 1 - 10^{-8}$) and BKZ-$\beta$ ($\beta = 12, 20$).