Shortest Vector from Lattice Sieving: a Few Dimensions for Free

Léo Ducas*

Cryptology Group, CWI, Amsterdam, The Netherlands

Abstract. Asymptotically, the best known algorithms for solving the Shortest Vector Problem (SVP) in a lattice of dimension n are sieve algorithms, which have heuristic complexity estimates ranging from $(4/3)^{n+o(n)}$ down to $(3/2)^{n/2+o(n)}$ when Locality Sensitive Hashing techniques are used. Sieve algorithms are however outperformed by pruned enumeration algorithms in practice by several orders of magnitude, despite the larger super-exponential asymptotical complexity $2^{\Theta(n \log n)}$ of the latter.
In this work, we show a concrete improvement of sieve-type algorithms. Precisely, we show that a few calls to the sieve algorithm in lattices of dimension less than $n - d$ solves SVP in dimension n, where $d = \Theta(n/\log n)$.
Although our improvement is only sub-exponential, its practical effect in relevant dimensions is quite significant. We implemented it over a simple sieve algorithm with $(4/3)^{n+o(n)}$ complexity, and it outperforms the best sieve algorithms from the literature by a factor of 10 in dimensions 70-80. It performs less than an order of magnitude slower than pruned enumeration in the same range.
By design, this improvement can also be applied to most other variants of sieve algorithms, including LSH sieve algorithms and tuple-sieve algorithms. In this light, we may expect sieve-techniques to outperform pruned enumeration in practice in the near future.

Keywords: Cryptanalysis, Lattice, Sieving, Nearest-Plane.

1 Introduction

The concrete hardness of the Shortest Vector Problem (SVP) is at the core of the cost estimates of attacks against lattice-based cryptosystems. While those schemes may use various underlying problems (NTRU [?], SIS [?], LWE [?]) their cryptanalysis boils down to solving large instances of the Shortest Vector Problem inside BKZ-type algorithms. There are two classes of algorithms for SVP: enumeration algorithms and sieve algorithms.

The first class of algorithms (enumeration) was initiated by Pohst [?]. Kannan [?, ?, ?] proved that with appropriate pre-processing, the shortest vector could be found in time $2^{\Theta(n \log n)}$. This algorithm only requires a polynomial

* Supported by a Veni Innovational Research Grant from NWO under project number 639.021.645.

amount of memory. These algorithms can be made much faster in practice using some heuristic techniques, in particular the pruning technique [?, ?, ?, ?].

The second class of algorithms (sieving) started with Ajtai et al. [?], and requires single exponential time and memory. Variants were heuristically analyzed [?, ?], giving a $(4/3)^{n+o(n)}$ time complexity and a $(4/3)^{n/2+o(n)}$ memory complexity. A long line of work, including [?, ?, ?, ?] decrease this time complexity down to $(3/2)^{n/2+o(n)}$ at the cost of more memory. Other variants (tuple-sieving) are designed to lower the memory complexity [?, ?].

The situation is rather paradoxical: asymptotically, sieving algorithms should outperform enumeration algorithms, yet in practice, sieving remains several orders of magnitude slower. This situation makes security estimates delicate, requiring both algorithms to be considered. In that respect, one would much prefer enumeration to become irrelevant, as the heuristics used in this algorithm makes prediction of its practical cost tedious and maybe inaccurate.

To this end, an important goal is to improve not only the asymptotic complexity of sieving, but also its practical complexity. Indeed, much can been gained from asymptotically negligible tricks, fine-tuning of the parameters, and optimized implementation effort [?, ?, ?].

This work. We propose a new practical improvement for sieve algorithms. In theory, we can heuristically show that it contributes a sub-exponential gain in the running time and the memory consumption. In practice, our implementation outperforms all sieving implementations of the literature by a factor of 10 in dimensions 70-80, despite the fact that we did not implement some known improvements [?, ?]. Our improved sieving algorithm performs reasonably close to pruned enumeration; more precisely, within less than an order of magnitude of the optimized pruned enumeration implementation in `fplll`'s library [?, ?, ?].[1]

In brief, the main idea behind our improvement is exploiting the fact that sieving produces many short vectors, rather than only one. We use this fact to our advantage by solving SVP in lattices of dimension n while running a sieve algorithm in projected sub-lattices of dimension smaller than $n - d$. Using an appropriate pre-processing, we show that one may choose d as large as $\Theta(n/\log n)$. Heuristic arguments lead to a concrete prediction of $d \approx \frac{n \ln(4/3)}{\ln(n/2\pi e)}$. This prediction is corroborated by our experiments.

At last, we argue that, when combined with the LSH techniques [?, ?], our new technique should lead to a sieve algorithm that outperforms enumeration in practice, for dimensions maybe as low as $n = 90$. We also suggest four approaches to further improve sieving, including amortization inside BKZ.

Outline. We shall start with preliminaries in Section ??, including a generic presentation of sieve algorithms in Section ??. Our main contribution is presented in Section ??. In Section ??, we present details of our implementation, including

[1] Please note that this library was not so fast for SVP and BKZ a few years ago and it recently caught up with the state of the art with the addition of a `pruner` module [?], and of an external `Strategizer` [?].

other algorithmic tricks. In Section ?? we report on the experimental behavior of our algorithm, and compare its performances to the literature. We conclude with a discussion in Section ??, on combining our improvement with the LSH techniques [?, ?, ?], and suggest further improvements.

Acknowledgments

The author wishes to thank Koen de Boer, Gottfried Herold, Pierre Karman, Elena Kirshanova, Thijs Laarhoven, Marc Stevens and Eamonn Postlethwaite for enlightening conversations on this topic. The author is also extremely grateful to Martin Albrecht and the FPLLL development team for their thorough work on the `fplll` and `fpylll` libraries. This work was supported by a Veni Innovational Research Grant from NWO under project number 639.021.645.

2 Preliminaries

2.1 Notations and Basic Definitions

All vectors are denoted by bold lower case letters and are to be read as column-vectors. Matrices are denoted by bold capital letters. We write a matrix $\mathbf{B}$ as $\mathbf{B} = (\mathbf{b}_0, \cdots, \mathbf{b}_{n-1})$ where $\mathbf{b}_i$ is the i-th column vector of $\mathbf{B}$. If $\mathbf{B} \in \mathbb{R}^{m \times n}$ has full-column rank n, the lattice $\mathcal{L}$ generated by the basis $\mathbf{B}$ is denoted by $\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} \mid \mathbf{x} \in \mathbb{Z}^n\}$. We denote by $(\mathbf{b}_0^*, \cdots, \mathbf{b}_{n-1}^*)$ the Gram-Schmidt orthogonalization of the matrix $(\mathbf{b}_0, \cdots, \mathbf{b}_{n-1})$. For $i \in \{0, \cdots, n-1\}$, we denote the orthogonal projection to the span of $(\mathbf{b}_0, \cdots, \mathbf{b}_{i-1})$ by π_i. For $0 \leq i < j \leq n$, we denote by $\mathbf{B}_{[i,j]}$ the local projected block $(\pi_i(\mathbf{b}_i), \cdots, \pi_i(\mathbf{b}_{j-1}))$, and when the basis is clear from context, by $\mathcal{L}_{[i,j]}$ the lattice generated by $\mathbf{B}_{[i,j]}$. We use $\mathbf{B}_i$ and $\mathcal{L}_i$ as shorthands for $\mathbf{B}_{[i,n]}$ and $\mathcal{L}_{[i,n]}$.

The Euclidean norm of a vector $\mathbf{v}$ is denoted by $\|\mathbf{v}\|$. The volume of a lattice $\mathcal{L}(\mathbf{B})$ is $\mathrm{Vol}(\mathcal{L}(\mathbf{B})) = \prod_i \|\mathbf{b}_i^*\|$, that is an invariant of the lattice. The first minimum of a lattice $\mathcal{L}$ is the length of a shortest non-zero vector, denoted by $\lambda_1(\mathcal{L})$. We use the abbreviations $\mathrm{Vol}(\mathbf{B}) = \mathrm{Vol}(\mathcal{L}(\mathbf{B}))$ and $\lambda_1(\mathbf{B}) = \lambda_1(\mathcal{L}(\mathbf{B}))$.

2.2 Lattice Reduction

The Gaussian Heuristic predicts that the number $|\mathcal{L} \cap \mathcal{B}|$ lattice of points inside a measurable body $\mathcal{B} \subset \mathbb{R}^n$ is approximately equal to $\mathrm{Vol}(\mathcal{B})/\mathrm{Vol}(\mathcal{L})$. Applied to Euclidean n-balls, it leads to the following prediction of the length of a shortest non-zero vector in a lattice.

Definition 1 (Gaussian Heuristic). *We denote by* $\mathrm{gh}(\mathcal{L})$ *the expected first minimum of a lattice* $\mathcal{L}$ *according to the Gaussian Heuristic. For a full rank lattice* $\mathcal{L} \subset \mathbb{R}^n$*, it is given by:*

$$\mathrm{gh}(\mathcal{L}) = \sqrt{n/2\pi e} \cdot \mathrm{Vol}(\mathcal{L})^{1/n}.$$

We also denote $\mathrm{gh}(n)$ *for* $\mathrm{gh}(\mathcal{L})$ *of any* n*-dimensional lattice* $\mathcal{L}$ *of volume* 1*:* $\mathrm{gh}(n) = \sqrt{n/2\pi e}$.

Definition 2 (Hermite-Korkine-Zolotarev and Block-Korkine-Zolotarev reductions [?]).
The basis $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$ *of a lattice* $\mathcal{L}$ *is said to be HKZ reduced if* $\|\mathbf{b}_i^*\| = \lambda_1(\mathcal{L}(\mathbf{B}_i))$ *for all* $i < n$. *It is said BKZ reduced with block-size* b *(for short BKZ-*b *reduced)* $\|\mathbf{b}_i^*\| = \lambda_1(\mathcal{L}(\mathbf{B}_{[i:\max(i+b,n)]}))$ *for all* $i < n$.[2]

Under the Gaussian Heuristic, we can predict the shape $\ell_0 \ldots \ell_{n-1}$ of an HKZ reduced basis, i.e., the sequence of expected norms for the vectors $\mathbf{b}_i^*$. The sequence is inductively defined as follows:

Definition 3. *The HKZ-shape of dimension* n *is defined by the following sequence:*

$$\ell_0 = \mathrm{gh}(n) \quad \textit{and} \quad \ell_i = \mathrm{gh}(n-i) \cdot \Big(\prod_{j<i} \ell_j\Big)^{-\frac{1}{n-i}} .$$

Note that the Gaussian Heuristic is known to be violated in small dimensions [?], fortunately we only rely on the above prediction for $i \ll n$.

Definition 4 (Geometric Series Assumption). *Let* $\mathbf{B}$ *be a BKZ-*b *reduced basis of a lattice of volume* 1. *The Geometric Series Assumption states that:*

$$\|\mathbf{b}_i^*\| = \alpha_b^{\frac{n-1}{2}-i}$$

where $\alpha_b = \mathrm{gh}(b)^{2/b}$.

This model is reasonably accurate in practice for $b > 50$ and $b \ll n$. For further discussion on this model and its accuracy, the reader may refer to [?, ?, ?].

2.3 Sieve Algorithms

There are several variants of sieving algorithms, even among the restricted class of Sieving algorithms having asymptotic complexity $(4/3)^{n+o(n)}$ [?, ?]. Its generic form is given below.

Algorithm 1 Sieve($\mathcal{L}$)

Require: The basis $\mathbf{B}$ of a lattice $\mathcal{L}$ of dimension n
Ensure: A list L of vectors
  $L \leftarrow$ a set of N random vectors (of length at most $2^n \cdot \mathrm{Vol}(\mathcal{L})^{1/n}$) from $\mathcal{L}$ where $N = (4/3)^{n/2+o(n)}$.
  while $\exists (\mathbf{v}, \mathbf{w}) \in L^2$ such that $\|\mathbf{v} - \mathbf{w}\| < \|\mathbf{v}\|$ **do**
    $\mathbf{v} \leftarrow \mathbf{v} - \mathbf{w}$
  end while
  return L

[2] The notion of BKZ-reduction is typically slightly relaxed for algorithmic purposes, see [?].

The initialization of the list L can be performed by first computing an LLL-reduced basis of the lattice [?], and taking small random linear combinations of that basis.

Using heuristic arguments, one can show [?] that this algorithm will terminate in time $N^2 \cdot \mathrm{poly}(n)$, and that the output list contains a shortest vector of the lattice. The used heuristic reasoning might fail in some special lattices, such as $\mathbb{Z}^n$. However, nearly all lattices occurring in a cryptographic context are random-looking lattices, for which these heuristics have been confirmed extensively.

Many tricks can be implemented to improve the hidden polynomial factors. The most obvious one consists of working modulo negation of vectors (halving the list size), and to exploit the identity $\|\mathbf{v} \pm \mathbf{w}\|^2 = \|\mathbf{v}\|^2 + \|\mathbf{w}\|^2 \pm 2\langle \mathbf{v}, \mathbf{w} \rangle$: two reductions can be tested for the price of one inner product.

More substantial algorithmic improvements have been proposed in [?]: sorting the list by Euclidean length to make early reduction more likely, having the list size be adaptive, and having a queue of updated vectors to avoid considering the same pair several times. Another natural idea used in [?] consists of strengthening the LLL-reduction to a BKZ-reduction with medium block-size, so as to decrease the length of the initial random vectors.

One particularly cute low-level trick proposed by Fitzpatrick et al. [?] consists of quickly rejecting pairs of vectors depending on the Hamming weight of the XOR of their bit signs. We shall re-use (a variant of) this trick in our implementation. This technique is in fact well known in the Nearest-Neighbor-Search (NNS) literature [?], and sometimes referred to as SimHash.

The N^2 factor may also be improved to a sub-quadratic factor N^c, $1 < c < 2$ using advanced NNS data-structures [?, ?, ?]. While improving the exponential term, those techniques introduce extra hidden sub-exponential factors, and typically require more memory.[3] In practice these improvements remain substantial [?]. Yet, as the new improvements presented in this paper are orthogonal, we leave it to the interested reader to consult this literature.

3 The SubSieve Algorithm and its Analysis

3.1 Approach

Our improvements rely on the remark that the output of the sieve contains much more information than a shortest vector of $\mathcal{L}$. Indeed, the analysis of [?, ?] suggests that the outputted list contains the N shortest vector of the lattice, namely, all the vectors of the lattice of length less than $\sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L})$.

We proceed to exploit this extra information by solving SVP in a lattice of larger dimension. Let us choose an index d, and run the sieve in the projected sub-lattice $\mathcal{L}_d$, of dimension $n - d$. We obtain the list:

$$L := \mathsf{Sieve}(\mathcal{L}_d) = \{\mathbf{x} \in \mathcal{L}_d \setminus \{\mathbf{0}\} | \ \|\mathbf{x}\| \leq \sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L}_d)\}. \quad (1)$$

[3] Becker *et al.* [?] proposed a way to not require extra memory, yet it may hide an extra polynomial factor on time.

Our hope is that the desired shortest non-zero vector $\mathbf{s}$ (of expected length $\mathrm{gh}(\mathcal{L})$) of the full lattice $\mathcal{L}$ projects to a vector contained in L, i.e. $\pi_d(\mathbf{s}) \in L$ or equivalently by equation (**??**), that $\|\pi_d(\mathbf{s})\| \leq \sqrt{4/3}\,\mathrm{gh}(\mathcal{L}_d)$. Because $\|\pi_d(\mathbf{s})\| \leq \|\mathbf{s}\| = \mathrm{gh}(\mathcal{L})$, it is sufficient that:

$$\mathrm{gh}(\mathcal{L}) \leq \sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L}_d). \tag{2}$$

In fact, we may relax this condition, as we rather expect the projection to be shorter: $\|\pi_d(\mathbf{s})\| \approx \sqrt{(n-d)/n}\|\mathbf{s}\|$ assuming the direction of $\mathbf{s}$ is uniform and independent of the basis $\mathbf{B}$. More precisely, it will happen with constant probability that $\|\pi_d(\mathbf{s})\| \leq \sqrt{(n-d)/n}\|\mathbf{s}\|$. Instead we may therefore optimistically require:

$$\sqrt{\frac{n-d}{n}} \cdot \mathrm{gh}(\mathcal{L}) \leq \sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L}_d). \tag{3}$$

We are now searching for a vector $\mathbf{s} \in L$ such that $\|\mathbf{s}\| \approx \mathrm{gh}(\mathcal{L})$, and such that $\mathbf{s}_d := \pi_d(\mathbf{s}) \in L$. By exhaustive search over the list L, let us assume we know $\mathbf{s}_d$; we now need to recover the full vector $\mathbf{s}$. We write $\mathbf{s} = \mathbf{B}\mathbf{x}$ and split $\mathbf{x} = (\mathbf{x}', \mathbf{x}'')$ where $\mathbf{x}' \in \mathbb{Z}^d$ and $\mathbf{x}'' \in \mathbb{Z}^{n-d}$. Note that $\mathbf{s}_d = \pi_d(\mathbf{B}\mathbf{x}) = \mathbf{B}_d\mathbf{x}''$, so we may recover $\mathbf{x}''$ from $\mathbf{s}_d$.

We are left with the problem of recovering $\mathbf{x}' \in \mathbb{Z}^d$ such that $\mathbf{B}'\mathbf{x}' + \mathbf{B}''\mathbf{x}''$ is small where $[\mathbf{B}'|\mathbf{B}''] = \mathbf{B}$, i.e., finding the short vector $\mathbf{s}$ in the lattice coset $\mathcal{L}(\mathbf{B}') - \mathbf{B}''\mathbf{x}$.

For appropriate parameters, this is an easy BDD instance over the d-dimensional lattice spanned by $\mathbf{B}'$. More precisely, a sufficient condition to solve this problem using Babai's Nearest-Plane algorithm [**?**] is that $|\langle \mathbf{b}_i^*, \mathbf{s}\rangle| \leq \frac{1}{2}\|\mathbf{b}_i^*\|^2$ for all $i < d$. A sufficient condition is that:

$$\mathrm{gh}(\mathcal{L}) \leq \frac{1}{2} \min_{i<d} \|\mathbf{b}_i^*\|. \tag{4}$$

This conditions is far from tight, and in practice should not be a serious issue. Indeed, even for a strongly reduced basis, the d first Gram-Schmidt lengths won't be much smaller than $\mathrm{gh}(\mathcal{L})$, say by more than a factor 2. On the other hand assuming $\mathbf{s}$ has a random direction we expect $|\langle \mathbf{b}_i^*, \mathbf{s}\rangle| \leq \omega(\ln n)/\sqrt{n} \cdot \|\mathbf{b}_i^*\| \cdot \|\mathbf{s}\|$ except with super-polynomially small probability. We will check this condition in the complexity analysis below (Section **??**), and will simply ignore it in the rest of this paper.

Algorithm 2 $\mathsf{SubSieve}(\mathcal{L}, d)$

Require: The basis $\mathbf{B} = [\mathbf{B}'|\mathbf{B}'']$ of a lattice $\mathcal{L}$ of dimension n
Ensure: A short vector of $\mathcal{L}$
  $L \leftarrow \mathsf{Sieve}(\mathcal{L}_d)$
  for each $\mathbf{w}_i \in L$ **do**
    Compute $\mathbf{x}_i''$ such that $\mathbf{B}_d \cdot \mathbf{x}_i'' = \mathbf{w}_i$
    $\mathbf{t}_i = \mathbf{B}'' \cdot \mathbf{x}''$
    $\mathbf{s}_i \leftarrow \mathsf{Babai}(\mathbf{B}', \mathbf{t}_i) + \mathbf{t}_i$
  end for
  return the shortest $\mathbf{s}_i$

Heuristic Claim 1 *For a random lattice, and under conditions* (??) *and* (??), $\mathsf{SubSieve}(\mathcal{L}, d)$ *outputs the shortest vector of* $\mathcal{L}$*, and its complexity is dominated by the cost* $N^2 \cdot \mathrm{poly}(n)$ *of* $\mathsf{Sieve}(\mathcal{L}_d)$*, with an additive overhead of* $n^2 \cdot N$ *real arithmetic operations.*

We note that the success of our approach depends crucially on the length of the Gram-Schmidt norms $\|\mathbf{b}_i^*\|$ (indeed for a fixed d, $\mathrm{gh}(\mathcal{L}_d)$ depends only of $\prod_{i \geq d} \|\mathbf{b}_i^*\|$). In the following Section ??, we will argue that our approach can be successfully instantiated with $d = \Theta(n/\ln n)$ using an appropriate pre-processing of negligible cost.

3.2 Complexity analysis

Assume that our lattice $\mathcal{L}$ has volume 1 (without loss of generality by scaling), and that its given basis $\mathbf{B}$ is BKZ-b reduced. Using the Geometric Series Assumption (Definition ??) we calculate the volume of $\mathcal{L}_d$:

$$\mathrm{Vol}(\mathcal{L}_d) = \prod_{i=d}^{n-1} \|\mathbf{b}_i^*\| = \prod_{i=d}^{n-1} \alpha_b^{\frac{n-1}{2}-i} = \alpha_b^{d(d-n)/2}.$$

Recalling that for a k-dimensional lattice we have $\mathrm{gh}(\mathcal{L}) \approx \mathrm{Vol}(\mathcal{L})^{1/k}\sqrt{k/(2\pi e)}$, condition (??) is rewritten to

$$\sqrt{\frac{n}{2\pi e}} \leq \sqrt{\frac{4}{3}} \cdot \sqrt{\frac{n-d}{2\pi e}} \cdot \alpha_b^{-d/2}.$$

Taking logarithms, we rewrite the above condition as

$$d \ln \alpha_b \leq \ln(4/3) + \ln(1 - d/n).$$

We (arbitrarily) choose $b = n/2$ which ensures that the cost of the BKZ-preprocessing is negligible compared to the cost of sieving in dimension $n - o(n)$. Unrolling the definitions, we notice that $\ln \alpha_b = \Theta((\ln b)/b) = \Theta((\ln n)/n)$. We conclude that condition (??) is satisfied for some $d = \Theta(n/\ln n)$.

The second condition (??) for the correctness of Babai lifting is easily satisfied: for $i < d = o(n)$ we have $\|\mathbf{b}_i^*\| = \mathrm{gh}(b)^{(n-o(n))/b} = \mathrm{gh}(b)^{2-o(1)} = n^{1-o(1)}$, while $\mathrm{gh}(n) = \Theta(n^{1/2})$. This concludes our argument of the following claim.

Heuristic Claim 2 *Having preprocessed the basis* $\mathbf{B}$ *of* $\mathcal{L}$ *with the BKZ algorithm with blocksize* $b = n/2$ *—for a cost of at most* $\mathrm{poly}(n)$ *time the cost of* Sieve *in dimension* $n/2$*— our* $\mathsf{SubSieve}(\mathcal{L}, d)$ *algorithm will find the shortest vector of* $\mathcal{L}$ *for some* $d = \Theta(n/\ln n)$.

In particular, $\mathsf{SubSieve}(\mathcal{L}, d)$ *is faster than* $\mathsf{Sieve}(\mathcal{L})$ *by a sub-exponential factor* $2^{\Theta(n/\ln n)}$.

The fact that BKZ-b requires only $\mathrm{poly}(n)$ calls to an SVP oracle in dimension b is justified in [**?**].

3.3 (Progressive) Iteration as pre-processing

We now propose an alternative approach to provide pre-processing in our context. It consists of applying an extension of the SubSieve algorithm iteratively from a weakly reduced basis to a strongly reduced one. To proceed, we first need to slightly extend our algorithm, to not only provide one short vector, but a partial basis $\mathbf{V} = [\mathbf{v}_0|\dots|\mathbf{v}_m]$ of rank m, such that their Gram-Schmidt lengths are as short as possible. In other words, the algorithm now attempts to provide the first vectors of an HKZ-reduced basis. For all practical purpose, $m = n/2$ is sufficiently large. This extension comes at a negligible additional cost of $O(n^3)\cdot N$ compared to the sieve of complexity $\mathrm{poly}(n)\cdot N^2$.

Algorithm 3 $\mathsf{SubSieve}^+(\mathcal{L}, d)$

Require: The basis $\mathbf{B} = [\mathbf{B}'|\mathbf{B}'']$ of a lattice $\mathcal{L}$ of dimension n
Ensure: A short vector of $\mathcal{L}$
  $L \leftarrow \mathsf{Sieve}(\mathcal{L}_d)$
  for each $\mathbf{w}_i \in L$ **do**
    Compute $\mathbf{x}_i''$ such that $\mathbf{B}_d \cdot \mathbf{x}_i'' = \mathbf{w}_i$
    $\mathbf{t}_i = \mathbf{B}'' \cdot \mathbf{x}''$
    $\mathbf{s}_i \leftarrow \mathsf{Babai}(\mathbf{B}', \mathbf{t}_i) + \mathbf{t}_i$
  end for
  for $j = 0 \dots n/2 - 1$ **do**
    Set $\mathbf{v}_j$ to be the $\mathbf{s}_i$ vector minimizing $\|\pi_{(\mathbf{v}_0 \dots \mathbf{v}_{j-1})^\perp}(\mathbf{s}_i)\|$ such that $\mathbf{s} \notin \mathrm{Span}(\mathbf{v}_0 \dots \mathbf{v}_{j-1})$
  end for
  return $(\mathbf{v}_0 \dots \mathbf{v}_{n/2-1})$

Then, the iteration consists of completing $\mathbf{V}$ into a basis of $\mathcal{L}$, and to use it as our new input basis $\mathbf{B}$.[4]

[4] This can be done by applying LLL [**?**] on the matrix $[\mathbf{V}|\mathbf{B}]$, which eliminates linear dependencies. As LLL can only decrease partial determinants, the volume of the first d-vectors after this process can only be smaller than the volume of $\mathbf{V}$: this does not affect condition (**??**) and (**??**).

Additionally, as conditions (??) or even its optimistic variant (??) are not necessary conditions, we may hope that a larger value of d may probabilistically lead faster to the shortest vector. In fact, hoping to obtain the shortest vector with d larger than required by the pessimistic condition (??) can be interpreted in the pruning framework of [?, ?]; this will be discussed in Section ??.

For this work, we proceed with a simple strategy, namely we iterate starting with a large value of d (say $n/4$) and decrease d by 1 until the shortest vector (or a vector of the desired length) is found. This way, the failed attempts with too small d nevertheless contribute to the approximate HKZ-reduction, improving the basis for the next attempt.

The author admit to have no theoretical arguments (or even heuristic) to justify that this iterating approach should be more efficient than the preprocessing approach presented in Section ??. Yet, as we shall see, this method works quite well in practice, and has the advantage of being much simpler to implement.

Remark. One natural tweak is to also consider the vectors in $\mathbf{B}'$ when constructing the new partial basis $\mathbf{V}$ so as to ensure that the iteration never introduces a regression. Yet, as the optimistic condition is probabilistic, we may get stuck with an unlucky partial basis, and prefer to change it at each iteration. This is reminiscent of the rerandomization of the basis in the extreme pruning technique of Gama et al. [?]. It is therefore not entirely clear if this tweak should be applied. In practice, we noted that applying this trick made the running time of the algorithm much more erratic, making it hard to determine if it should be better on average. For the sake of this initial study, we prefer to stick with the more stable version of the algorithm.

3.4 Tentative prediction of d on quasi-HKZ reduced basis

We now attempt to estimate the concrete maximal value d allowing our algorithm to succeed. We nevertheless warn the reader against strong conclusions on the concrete hardness of SVP from the analysis below. Indeed, it does not capture some practical phenomena, such as the fact that (??) is not strictly true in practice,[5] or more subtly that the directions of the vectors of $\mathbf{B}$ are not independent of the direction of the shortest vector $\mathbf{s}$ when $\mathbf{B}$ is so strongly reduced. Additionally, we identify in Section ?? avenues for improvements that could make this analysis obsolete.

We work under the heuristic assumption that the iterations up to $d_{\text{last}} - 1$ have almost produced an HKZ-reduced basis: $\|\mathbf{b}_i^*\| \approx \ell_i$ where ℓ_i follows the HKZ-shape of dimension n (Definition ??). From there, we determine whether the last iteration with $d = d_{\text{last}}$ should produce the shortest vector according to both the pessimistic and optimistic condition. For $i \ll n$ we use the first order approximation $\ln \ell_i \approx \ln \ell_0 - i \cdot \ln \ell_0/\ell_1$ and obtain

$$\ln \ell_i \approx \ln \ell_0 - i \cdot \frac{\ln(n/2\pi)}{2n}.$$

[5] some vectors below the $\sqrt{4/3} \cdot \text{gh}(\mathcal{L}_d)$ bound may be missing, while other vectors above this bound may be included.

The pessimistic condition (??) and the optimistic condition (??) respectively rewrite as:

$$\ln \ell_0 \leq \ln \sqrt{4/3} + \ln \ell_d \qquad \text{and} \qquad \ln \sqrt{\frac{n-d}{n}} + \ln \ell_0 \leq \ln \sqrt{4/3} + \ln \ell_d.$$

With a bit of rewriting, we arrive at the following maximal value of d respectively under the following pessimistic and optimistic conditions:

$$d \approx \frac{n \ln 4/3}{\ln(n/2\pi)} \qquad \text{and} \qquad d \approx \frac{n \ln 4/3}{\ln(n/2\pi e)}.$$

We can also numerically simulate more precisely the maximal value of d using the exact values of the ℓ_i. All four predictions are depicted on Figure ??. Our plots start at dimension 50, the conventional cut-off for the validity of the Gaussian Heuristic [?, ?]. We note that the approximated predictions are accurate, up to an additive term 2 over the value of d for relevant dimensions $n \leq 250$. We also note that in this range the dimension gain d looks very much linear: for all practical concerns, our improvement should appear essentially exponential.

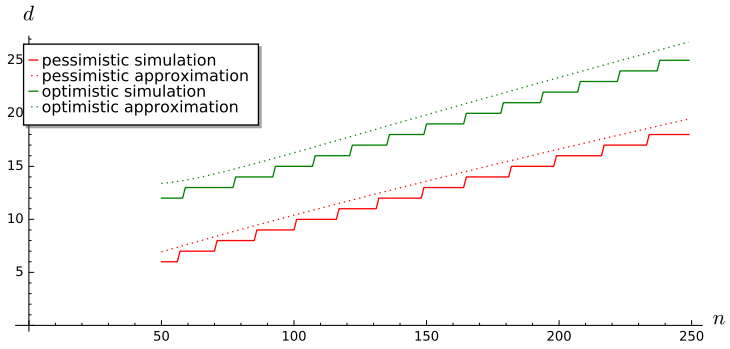

Fig. 1. Predictions of the maximal successful choice of d, under various methods and conditions.

4 Other Optimizations and Implementation Details

In this section, we describe a baseline sieve algorithm and two additional tricks to improve its practical efficiency. So as to later report the improvement brought by each trick and by our main contribution, we shall refer to 4 versions of our algorithm, activating one feature at the time:

- V0: GaussSieve baseline implementation
- V1: GaussSieve with `XOR-POPCNT` trick
- V2: GaussSieve with `XOR-POPCNT` trick and progressive sieving
- V3: Iterated SubSieve$^+$ with `XOR-POPCNT` trick and progressive sieving.

4.1 Baseline Implementation

As a baseline algorithm, we essentially use the Gauss-Sieve algorithm of [?], with the following tweaks.

First, we do not resort to Gaussian Sampling [?] for the construction of the list L as the sphericity of the initial list does not seem so crucial in practice, and leads to starting the sieve with vectors longer than necessary. Instead, we choose vectors by sampling their $n/4$ last coordinates in base $\mathbf{B}$ uniformly in $\{0, \pm 1, \pm 2\}$, and choose the remaining coordinates deterministically using the Babai Nearst-Plane algorithm [?].

Secondly, we do not maintain the list perfectly sorted, but only re-sort it periodically. This makes the implementation somewhat easier[6] and does not affect performances noticeably. Similarly, fresh random vectors are not inserted in L one by one, but in batches.

Thirdly, we use a hash table to prevent collisions: if $\mathbf{v} \pm \mathbf{w}$ is already in the list, then we cancel the reduction $\mathbf{v} \leftarrow \mathbf{v} \pm \mathbf{w}$. Our hash function is defined as random linear function $h : \mathbb{Z}^n \to \mathbb{Z}/2^{64}\mathbb{Z}$ tweaked so that $h(\mathbf{x}) = h(-\mathbf{x})$; hashing is fast, and false collisions should be very rare. This function is applied to the integer coordinates of the vector in base $\mathbf{B}$.

At last, the termination condition is as follows: the algorithm terminates when no pairs can be reduced, and when the ball of radius $\sqrt{4/3}\,\mathrm{gh}(\mathcal{L})$ is half-saturated according to the Gaussian Heuristic, *i.e.* when the list L contains at least $\frac{1}{2}\sqrt{4/3}^n$ vectors of length less than $\sqrt{4/3}\,\mathrm{gh}(\mathcal{L})$.

At the implementation level, and contrary to most implementations of the literature, our implementation works by representing vectors in bases $\mathbf{B}$ and $\mathbf{B}^*$ rather than in the canonical basis of $\mathbb{R}^n$. It makes application of Babai's algorithm [?] more idiomatic, and should be a crucial feature to use it as an SVP solver inside BKZ.

4.2 The `XOR-POPCNT` Trick (a.k.a. SimHash)

This trick —which can be traced back to [?]— was developed for sieving in [?]. It consists of compressing vectors to a short binary representation that still carries some geometrical information: it allows for a quick approximation of inner-products. In more detail, they choose to represent a real vector $\mathbf{v} \in \mathbb{R}^n$ by the binary vector $\tilde{\mathbf{v}} \in \mathbb{Z}_2^n$ of it signs, and compute the Hamming weight $H = |\tilde{\mathbf{w}} \oplus \tilde{\mathbf{v}}|$ to determine whether $\langle \mathbf{v}, \mathbf{w} \rangle$ is expected to be small or large (which in turn informs us about the length $\|\mathbf{v} - \mathbf{w}\|^2 = \|\mathbf{v}\|^2 + \|\mathbf{w}\|^2 - 2\langle \mathbf{v}, \mathbf{w} \rangle$). If H is small enough then the exact length is computed, otherwise the pair is directly rejected.

This trick greatly decreases the practical computational cost and the memory bandwidth of the algorithm, in particular by exploiting the native `POPCNT` instruction available on most modern CPUs.

[6] It avoids resorting to non-contiguous containers, following the nomenclature of `c++` standard library.

Following the original idea [?], we use a generalized version of this trick, allowing the length of the compressed representation to differ from the lattice dimension. Indeed, we can for example choose $c \neq n$ vectors $\mathbf{r}_1, \ldots, \mathbf{r}_c$, and compress $\mathbf{v}$ as $\tilde{\mathbf{v}} \in \mathbb{Z}_2^c$ where $\tilde{v}_i = \mathrm{sign}(\langle \mathbf{v}, \mathbf{r}_i \rangle)$. This allows not only to align c to machine-word size, but also to tune the cost and the fidelity of this compressed representation.

In practice we choose $c = 128$ (2 machine words), and set the $\mathbf{r}_i$'s to be sparse random ternary vectors. We set the acceptance threshold to $|\tilde{\mathbf{w}} \oplus \tilde{\mathbf{v}}| < 47$,[7] having optimized this threshold by trial and error. Experimentally, the overall positive rate of this test is of about 2%, with a false negative rate of less than 30%. The sieve algorithm automatically compensates for false-negatives by increasing the list size.

4.3 Progressive Sieving

The trick described in this section was independently invented by Laarhoven and Mariano in [?]; and their work provides a much more thorough investigation of it. It consists of progressively increasing the dimension, first running the sieve in sublattices $\mathcal{L}_{[0,i]}$ for i increasing from (say) $n/2$ to n.[8]

It allows us to obtain an initial small pool of rather short vectors for a much cheaper cost. In turn, when we increase the dimension and insert new fresh vectors, the long fresh vectors get shorter noticeably faster thanks to this initial pool. We use the same terminating condition over $\mathcal{L}_{[0,i]}$ to decide when to increase i than the one described over the full lattice in section **??**.

4.4 Implementation details

The core of the our Sieving implementation is written in `c++` and the high level algorithm in `python`. It relies mostly on the `fpylll` [?] python wrapper for the `fplll` [?] library, used for calls to floating-point LLL [?] and providing the Gram-Schmidt orthogonalization. Our code is not templated by the dimensions, doing so could improve the performance substantially by allowing the compiler to unroll and vectorize the inner-product loop.

Our implementation is open source, available at https://github.com/lducas/SubSieve.

5 Experiments and Performances

In this section, we report on the behavior in practice of our algorithm and the performances of our implementation. All experiments were ran on a single core (Intel Core i7-4790 @3.60GHz).

[7] Of course, we also test wether $|\tilde{\mathbf{w}} \oplus \tilde{\mathbf{v}}| > 128 - 47$ in which case we attempt the reduction $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{w}$ instead of $\mathbf{v} \leftarrow \mathbf{v} - \mathbf{w}$.

[8] Note that unlike in our main algorithm SubSieve, the sublattices considered here are not projected sublattices, but simply the lattice spanned by the first basis vectors.

For these experiments, we use the Darmstadt lattice challenges [?]. We make a first run of `fplll`'s pruned enumeration (repeating it until 99% success probability) to determine the exact shortest vector.[9] Then, for our experiments, we stop our iteration of the SubSieve$^+$ algorithm when it returns a vector of the same length.

5.1 The dimension gain d in practice

In Figure ??, we compare the experimental value of d to the predictions of Section ??. The area of each disc at position (n, d) is proportional the number of experiments that succeeded with $d_{\text{last}} = d$. We repeated the experiment 20 times for each dimension n.

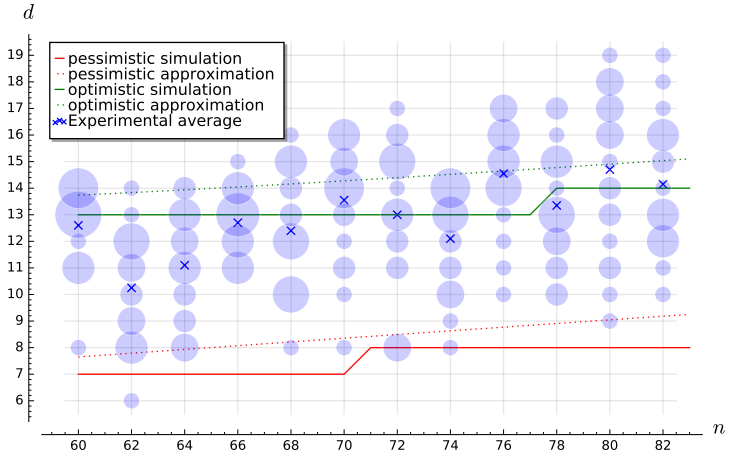

Fig. 2. Comparsion between experimental value of d with the prediction of Section ??.

We note that the average d_{last} fits reasonably well with the simulated optimistic prediction. Also, in the worst case, it is never lower than the simulated pessimistic prediction, except for one outlier in dimension 62.

Remark. The apparent erratic behavior of the average for varying n is most likely due to the fact that our experiments are only randomized over the input basis, and not over the lattice itself. Indeed the actual length of the shortest vectors vary a bit around the Gaussian Heuristic, and it seems that the shorter it actually is, the easier it is to find with our algorithm.

5.2 Perfomances

We present in Figure ?? the perfomances of the 4 versions of our implementation and of `fplll`'s pruned enumeration with precomputed strategies [?].

[9] Which is significantly harder than finding the approximation required by [?] to enter in the hall of fame.

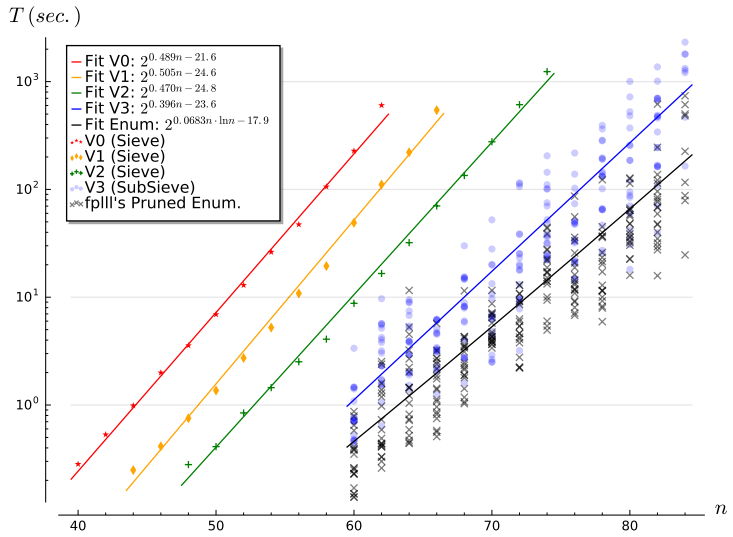

Fig. 3. Runing time T of all the 4 versions of sieving from `??` and `fplll`'s pruned enumeration with precomputed strategies.

Remark. In `fplll`, a strategy consists of the choice of a pre-processing blocksize b and of pruning parameters for the enumeration, as an attempt to reconstruct the BKZ 2.0 algorithm of Chen and Nguyen [?].

The external program `Strategizer` [?] first applies various descent techniques to optimize the pruning parameters, following the analysis of [?, ?, ?], and iterates over all (reasonable) choices of b, to return the best strategy for each dimension n. It may be considered near the state of the art, at least for the dimensions at hand. Unfortunately, we are unaware of timing reports for exact-SVP in this range of dimensions for other implementations.

It would also be adequate to compare ourselves to the recent discrete-pruning techniques of Fukase and Kashiwabara [?, ?], but again, we lack matching data. We note that neither the analysis of [?] nor the experiments of [?] provide evidences that this new method is significantly more efficient than the method of [?].

For a fair comparison with SubSieve, we stop repeating the pruned enumeration as soon as it finds the shortest vector, without imposing a minimal success probability (unlike the first run used to determine the of length shortest vectors). We also inform the enumerator of the exact length of that shortest vector, making its task somehow easier: without this information, it would enumerate at a larger radius.

As Algorithms V0, V1 and V2 have a rather deterministic running time depending only on the dimension, we only provide one sample. For V3 and enumeration, we provide 20 samples. To compute the fits, we first averaged the

running times for each dimension n, and then computed the least-square linear fit of their logarithms (computing directly an exponential least-square fit leads to a fit only capturing the two last dimensions).

The given fits are only indicative and we warn against extrapolations. In particular, we note thatthe linear fit of V3 is below the heuristic asymptotic estimate of $(4/3)^{n+o(n)}$.

We conclude that our main contribution alone contributes a speed-up of more than an order of magnitude in the dimensions ≥ 70 (V3 versus V2), and that all the tricks taken together provide a speed-up of more than two orders of magnitudes (V3 versus V0). It performs within less than an order of magnitude of enumeration (V3 versus Pruned Enum).

5.3 Performance comparison to the literature

The literature on lattice sieving algorithms is vast [?, ?, ?, ?, ?, ?, ?, ?, ?], and many papers do report implementation timings. We compare ourselves to four of them, namely a baseline implementation [?], and three advanced sieve implementations [?, ?, ?], which represent (to the best of our knowledge) the state of the art in three different directions. This is given in Table ??.

	Algorithms							
	V0	V1	V2	V3	[?][a]	[?]	[?]	[?]
features								
`XOR-POPCNT` trick		x	x	x		x		
pogressive sieving			x	x				
SubSieve				x				
LSH (more mem.)							x	
tuple (less mem.)								x
Dimension	Running times							
$n = 60$	227s	49s	8s	.9s	464s	79s	13s	1080s
$n = 70$	-	-	276s	10s	23933s	4500s	$\approx$ 250s [b]	33000s
$n = 80$	-	-	-	234s	-	-	4320s	94700s
CPU frequency (GHz)	3.6	3.6	3.6	3.6	4.0	4.0	2.3	2.3

[a] As reported by [?].
[b] This value is not given in [?] as their implementation only handles dimensions that are multiples of 4. We estimated it from the given values for $n = 68$ (169s) and $n = 72$ (418s).

Table 1. Comparison with other Sieve implementations.

Accounting for the CPU frequencies, we conclude that the implementation of our algorithm is more than 10 times faster than the current fastest sieve, namely the implementation of the Becker et al. algorithm [?] from Mariano et al. [?].[10]

Remark. While we can hardly compare to this computation considering the lack of documentation, we note that T. Kleinjung holds the record for the shortest vector found in Darmstadt Lattice challenge [?] of dimension 116 (seed 0), since May 2014, and reported having used *a* sieve algorithm. According to Herold and Kirshanova [?, Acknowledgments], the algorithm used by Kleinjung is similar to theirs.

Another Sieving record was achieved by Bos et al. [?], for an ideal lattice of dimension 128, exploiting symmetries of ideal lattices to improve time and memory substantially. The computation ran over 1024 cores for 9 days. Similar computation have been run on GPU's [?], using 8 GPU's for about 35 days.

6 Conclusion

6.1 Sieve will outperform enumeration

While this statement is asymptotically true, it was a bit unclear where the cross-over should be, and therefore whether sieving algorithms have any practical relevance for concrete security levels. For example, it is argued in [?] that the cross-over would happen somewhere between $n = 745$ and $n = 1895$.

Our new results suggest otherwise. We do refrain from computing a cross-over dimension from the fits of Figure ?? which are far from reliable enough for such an extrapolation; our prediction is of a different nature.

Our prediction is that —unless new enumerations techniques are discovered— further improvements of sieving techniques and implementations will outperform enumeration for exact-SVP in practice, for reachable dimensions, maybe even as low as $n = 90$. This, we believe, would constitute a landmark result. This prediction is backed by the following guesstimates, but also by the belief that fine-tuning, low-level optimizations and new ideas should further improve the state of the art. Some avenues for further improvements are discussed in Section ??.

Guesstimates. We can try to guesstimate how our improvements would combine with other techniques, in particular with List-Decoding Sieve [?]. The exact conclusion could be affected by many technical details, and is mostly meant to motivate further research and implementation effort.

Mariano et al. [?] report a running time of $1850s$ for LDSieve [?] in dimension $n = 76$. First, the `XOR-POPCNT` trick is not orthogonal to LSH techniques, so we shall omit it.[11] The progressive sieving trick provides a speed up of about 4

[10] The CPU frequency may not be the only property of the machines to take account of for a perfect comparison: memory access delay, memory bandwidth and cache sizes may have noticeable impacts.

[11] It could still be that, with proper tuning, combining them gives an extra speed-up.

in the relevant dimensions (V1 vs V2). Then, our main contribution offers 14 dimensions "for free", ($n = 90$, $d_{\text{last}} = 14$). More accurately, the iteration for increasing d would come at cost a factor $\sum_{i\geq 0}(\frac{3}{2})^{-i/2} \approx 5.5$. Overall we may expect to solve exact-SVP 90 in time $\approx 5.5 \cdot 1850/4 \approx 2500$s. In comparison, `fpylll`'s implementation of BKZ 2.0 [?] solved exact-SVP in average time 2612s over Darmstadt lattice challenge 90 (seed 0) over 20 samples on our machine. For a fairer comparison across different machines, this Enumeration timing could be scaled up by 3.6GHz/2.3GHz ≈ 1.5.

6.2 Avenues for Further Improvements

Pruning in SubSieve. As we mentioned in Section ??, our optimistic condition (??) can be viewed as a form of pruning: this condition corresponds in the framework of [?, ?] to a pruning vector of the form $(1, 1, \ldots, 1, \gamma, \ldots \gamma) \in \mathbb{R}^n$ with d many 1's, and $\gamma = (n-d)/n$. A natural idea is to attempt running SubSieve using $\gamma < (n-d)/n$, *i.e.* being even more optimistic than condition (??). Indeed, rather than cluelessly increasing d at each iteration, we could compute for each d the success probability, and choose the value of d giving the optimal cost over success probability ratio.

Walking beyond $\sqrt{4/3}\cdot\mathbf{gh}(\mathcal{L}_d)$. Noting $m = n-d$, another idea could consist of trying to get more vectors than the $\sqrt{4/3}^m$ shortest for a similar or slightly higher cost than the initial sieve, as this would allow d to increase a little bit. For example, we can extract the sublist A of all the vectors of length less than $\alpha \cdot \text{gh}(\mathcal{L}_d)$ where $\alpha \leq \sqrt{4/3}$ from the initial sieve, and use them to walk inside the ball of radius $\beta \cdot \text{gh}(\mathcal{L}_d) \geq \sqrt{4/3}$ where $\frac{\alpha}{\beta}\sqrt{\beta^2 - \alpha^2/4} = 1$. Indeed, one can show that the volume of $(\mathbf{v} + \alpha\mathcal{B}) \cap (\beta\mathcal{B}) = \Omega(n^c)$ for some constant c, where $\|\mathbf{v}\| = \beta$. According to the Gaussian Heuristic, this means that from any lattice point in the ball of radius $\beta + \epsilon$, there exists a step in the list A that leads to another lattice point in the ball of radius $\beta + \epsilon$, for some $\epsilon = o(1)$. This kind of variation have already been considered in the Sieving literature [?, ?].

Each step of this walk would cost α^m and there are $\beta^{m+o(m)}$ many points to visit. Note that in our context, this walk can be done without extra memory, by instantly applying Babai lifting and keeping only interesting lifted vectors. We suspect that this approach could be beneficial in practice for $\beta = \sqrt{4/3} + o(1)$, if not for the running time, at least for the memory complexity.

Amortization inside BKZ. We now consider two potential amortizations inside BKZ. Both ideas are not orthogonal to each others (yet may not be incompatible). If our SubSieve algorithm is to be used inside BKZ, we suggest fixing d_{last} (say, using the optimistic simulation), and to accept that we may not always solve SVP exactly; this is already the case when using pruned enumeration.

Already pre-processed. One notes that SubSieve$^+$ does more than ensure the shorteness of the first vector, and in fact attempts a partial HKZ reduction. This means that the second block inside the BKZ loop is already quite reduced when we are over with the first one. One could therefore hope that directly starting the iteration of Section **??** at $d = d_{\text{last}}$ could be sufficient for the second block, and so forth.

Optimistically, this would lead to an amortization factor f of $f = \sum_{i \geq 0} (\frac{4}{3})^{-i} = 4$, or even $f = \sum_{i \geq 0} (\frac{3}{2})^{-i/2} \approx 5.5$ depending on which sieve is used. In practice, it may be preferable to start at $d = d_{\text{last}} - 1$ for example.

5 *blocks for the price of* 9/4. A second type of amortization consists of overshooting the blocksize by an additive term k, so as to SVP-reduce $k+1$ consecutive blocks of dimension b for the price of one sieving in dimension $b+k$. Indeed, an HKZ-reduction of size $b+k$ as attempted by SubSieve$^+$ directly guarentees the BKZ-b reduction of the first $k+1$ blocks: we may jump directly by $k+1$ blocks. This overshoot costs a factor $(3/2)^{k/2}$ using the List-Decoding-Sieve [**?**]. We therefore expect to gain a factor $f = (k+1)/(3/2)^{k/2}$, which is maximal at $k = 4$, with $f = 20/9 \approx 2.2$.

Further, we note that the obtained basis could be better than a usual BKZ-b reduced basis, maybe even as good as a BKZ-$(b + \frac{k-1}{2})$ reduced basis. If so, the gain may be as large as $f' = (k+1)/(3/2)^{(k+1)/4}$, which is maximal at $k = 9$, with $f' \approx 3.6$.