# Finding closest lattice vectors using approximate Voronoi cells

Thijs Laarhoven

IBM Research
Rüschlikon, Switzerland
mail@thijs.com

**Abstract.** The two classical hard problems underlying the security of lattice-based cryptography are the shortest vector problem (SVP) and the closest vector problem (CVP). For SVP, lattice sieving currently has the best (heuristic) asymptotic time complexity: in high dimensions $d$, sieving can solve SVP in time $2^{0.292d+o(d)}$, using $2^{0.208d+o(d)}$ memory [Becker–Ducas–Gama–Laarhoven, SODA'16]. The best heuristic time complexity to date for CVP is $2^{0.377d+o(d)}$, using $2^{0.292d+o(d)}$ memory [Becker–Gama–Joux, ANTS'14].

In practice, the memory requirements of exponential-space algorithms makes it difficult to run these directly on high-dimensional lattices, and perhaps the most promising application of such methods is as part of a hybrid with lattice enumeration. A faster algorithm for solving the closest vector problem with preprocessing (CVPP) in low dimensions could be used to speed up enumeration for solving SVP or CVP in high dimensions, but so far it is not even clear whether the fastest heuristic SVP algorithms can solve CVP at all.

Our contributions are two-fold. First, we show that with sieving, we can heuristically solve CVP with equivalent asymptotic costs as SVP, improving upon the best complexities of Becker–Gama–Joux. Our second and main contribution is that by constructing approximate Voronoi cells of the lattice as preprocessing, we obtain significantly better complexities for CVPP. We can heuristically solve CVPP in $2^{d/4+o(d)}$ time and space, and the time complexity can be further reduced to as little as $2^{\varepsilon d+o(d)}$ for arbitrary $\varepsilon > 0$, using $(1/\varepsilon)^{O(d)}$ space.

Preliminary experiments for CVPP support these claims, and in dimension 50 we roughly obtain a factor 2000 speedup compared to the fastest sieving algorithms for solving SVP/CVP (without preprocessing). This may be a first step towards a practical hybrid between enumeration and sieving-based methods.

**Keywords:** lattices, sieving algorithms, Voronoi cells, shortest vector problem (SVP), closest vector problem (CVP), bounded distance decoding (BDD)

## 1 Introduction

**Hard lattice problems.** Lattices are discrete subgroups of $\mathbb{R}^d$: given a basis $B = \{\boldsymbol{b}_1, \dots, \boldsymbol{b}_d\} \subset \mathbb{R}^d$, the lattice generated by $B$ is defined as $\mathcal{L} = \mathcal{L}(B) := \{\sum_{i=1}^d \lambda_i \boldsymbol{b}_i : \lambda_i \in \mathbb{Z}\}$. Given a basis of a lattice $\mathcal{L}$, the Shortest Vector Problem (SVP) asks to find a shortest non-zero vector in $\mathcal{L}$ under the Euclidean norm, i.e., a non-zero lattice vector $\boldsymbol{s}$ of norm $\|\boldsymbol{s}\| = \lambda_1(\mathcal{L}) := \min_{\boldsymbol{v} \in \mathcal{L}\setminus\{\boldsymbol{0}\}} \|\boldsymbol{v}\|$. Given a basis of a lattice and a target vector $\boldsymbol{t} \in \mathbb{R}^d$, the Closest Vector Problem (CVP) asks to find a lattice vector $\boldsymbol{s} \in \mathcal{L}$ closest to $\boldsymbol{t}$. The preprocessing variant of CVP (CVPP) asks to preprocess the input data such that, when later given a target vector $\boldsymbol{t}$, one can quickly find a closest lattice vector to $\boldsymbol{t}$.

SVP and CVP are fundamental in the study of lattice-based cryptography, as the security of these schemes is directly related to their hardness in high dimensions. Various other hard lattice problems, such as the Learning With Errors (LWE) and Shortest Integer Solution (SIS) problems are closely related to SVP and CVP; see e.g. [Mic08,vdP11,Ste16] for reductions between different lattice problems. These reductions show that understanding the hardness of SVP and CVP is crucial for accurately estimating the security of lattice-based cryptographic schemes.

A preliminary version of this paper previously appeared at SAC 2016 [Laa16].

**Provable SVP/CVP algorithms.** Although SVP and CVP are both central in the study of lattice-based cryptography, algorithms for SVP have received somewhat more attention, including a benchmarking website to compare different methods [SG15]. Various SVP algorithms have been studied which can solve CVP as well, such as the polynomial-space, superexponential-time enumeration studied in e.g. [Kan83, FP85, GNR10, MW15]. More recently, several methods were proposed to solve SVP and/or CVP using only single exponential time (but also single exponential space). By constructing the Voronoi cell of the lattice [AEVZ02, SFS09, MV10a, BD15], Micciancio and Voulgaris showed that SVP and CVP(P) can provably be solved in time $2^{2d+o(d)}$, where Bonifas and Dadush improved the complexity for CVPP to $2^{d+o(d)}$. In high dimensions this method outperforms provable algorithms based on lattice sieving [AKS01b, AKS01a, NV08, PS09], running in $2^{2.465d+o(d)}$ time or more [PS09], but currently the best provable complexities for SVP and CVP are due to the discrete Gaussian sampling approach of [ADRS15, ADS15], provably solving both problems in $2^{d+o(d)}$ time and space on arbitrary lattices.

**Heuristic SVP/CVP algorithms.** When considering and comparing these methods in practice, we get a completely different picture. For dimensions of practical interest (say $d \approx 100$), both the Voronoi cell algorithm and the discrete Gaussian sampling approach seem completely impractical, while the asymptotically-inferior enumeration and lattice sieving turn out to perform much better than their provable asymptotics suggest.

The fastest heuristic methods in high dimensions are based on lattice sieving, and after a long series of theoretical works on constructing efficient heuristic sieving algorithms [NV08, MV10b, WLTB11, ZPH13, Laa15a, LdW15, BGJ15, BL16, BDGL16] as well as applied papers studying how to further speed up these algorithms in practice [MS11, Sch11, Sch13, FBB$^+$14, IKMT14, MTB14, MODB14, MLB15, BNvdP16, MB16, MLB17], the best heuristic time complexity for solving SVP currently stands at $2^{0.292d+o(d)}$ [BDGL16, MLB17], using $2^{0.208d+o(d)}$ memory. The current best heuristic time complexity for CVP is $2^{0.377d+o(d)}$ using $2^{0.292d+o(d)}$ memory, using sieving on a tower of overlattices [BGJ14].

In moderate dimensions, enumeration-based methods still dominate, as the cross-over point with single-exponential time algorithms like sieving seems to lie in higher dimensions. Moreover, the exponential memory of e.g. lattice sieving makes it not only hard to *run* this method at all in high dimensions, but it also significantly *slows down* the algorithm due to the large number of random memory accesses. Furthermore, parallelizing sieving efficiently is less trivial than parallelizing enumeration [MS11, IKMT14, BNvdP16, MLB17]. Some previous work focused on obtaining a tradeoff between enumeration and sieving, using less memory for sieving [BLS16, HK16] or using more memory for enumeration [KF16].

**CVPP instances inside lattice enumeration.** A well-known potential application of fast CVP(P) algorithms is as a subroutine within enumeration methods. As described in e.g. [GNR10, MW15], at any given level in the enumeration tree, one is attempting to solve a CVP instance in a lower-dimensional sublattice of $\mathcal{L}$, where the target vector is determined by the path from the root to the current node in the tree. Each node at this level in the tree corresponds to a CVP instance in the same sublattice, but with a different target. If we can preprocess this sublattice such that the amortized time complexity of solving many CVP instances in this sublattice is small, then this may speed up processing the bottom part of the enumeration tree. This in turn might help speed up the lattice basis reduction algorithm BKZ [Sch87, SE94, CN11], which commonly uses enumeration as its SVP subroutine in practice, and is key in assessing the security of lattice-based schemes.

## 1.1 Contributions

In this paper we revisit heuristic lattice sieving methods, as well as the recent trend to speed up these algorithms using nearest neighbor searching, and we investigate how these algorithms can be modified to solve CVP(P) and its variants efficiently. The resulting CVPP algorithm is perhaps best understood in terms of Voronoi cells, as explained below.

**Solving CVP with sieving.** To solve CVP efficiently, we show how to adapt lattice sieving algorithms based on the Nguyen–Vidick sieve [NV08] to a target vector $\boldsymbol{t}$, so that we can also solve CVP. Since the resulting algorithm is tailored specifically to the given CVP instance, this leads to the best asymptotic complexity for solving a single CVP instance: we can heuristically solve CVP in time $2^{0.292d+o(d)}$ and space $2^{0.208d+o(d)}$, matching the best complexities for SVP of [BDGL16]. This improvement is depicted in Figure 1. An open problem is to apply similar ideas to sieving methods based on the GaussSieve [MV10b], which would likely lead to improved complexities in practice.

**Solving CVPP using approximate Voronoi cells.** For solving CVPP efficiently, our approach is to first generate a long list of the shortest lattice vectors (using e.g. enumeration or sieving) as preprocessing, and then using this list of short lattice vectors as the preprocessed data for finding closest vectors to arbitrary targets faster. The preprocessed data can best be understood as an *approximate* Voronoi cell of the lattice, where the size of the preprocessed list determines how good this approximation is. Our algorithm for finding closest vectors is then essentially a nearest neighbor optimized form of the iterative slicing method of Sommer–Feder–Shalvi [SFS09].

To (heuristically) guarantee correctness of the resulting CVP method, we take two different approaches, leading to two different sets of time/space complexities for CVPP. Denoting $\mathcal{V}$ and $\mathcal{V}_L$ the exact and approximate Voronoi cells of the lattice (generated by the preprocessed list $L$), the two methods differ in how well $\mathcal{V}_L$ approximates $\mathcal{V}$:

- **$\mathbf{Vol}(\mathcal{V}_L) \approx \mathbf{Vol}(\mathcal{V})$**: If the approximate Voronoi cell is a good approximation of the exact Voronoi cell, then with high probability over the randomness over the target vectors, the iterative slicer returns the closest lattice vector to the target. To guarantee $\mathrm{Vol}(\mathcal{V}_L) \approx \mathrm{Vol}(\mathcal{V})$ we heuristically need a preprocessed list $L$ of size $2^{d/2+o(d)}$, where more memory can be used to speed up the nearest neighbor part of the iterative slicer. The resulting time/space complexities (ignoring preprocessing costs) are sketched in Figure 1, starting from $(\mathrm{Space}, \mathrm{Time}) = (2^{0.500d}, 2^{0.292d})$.
- **$\mathbf{Vol}(\mathcal{V}_L) \gg \mathbf{Vol}(\mathcal{V})$**: If the preprocessed list is much shorter, then with overwhelming probability the iterative slicer will not return the closest lattice point to a target. However, similar to *pruning* in lattice enumeration [GNR10], the running time of this method is decreased by a much more significant factor compared to the above method than the success probability: e.g. the success probability may be 0.001, but the time complexity decreases by a factor more than 1000. By appropriately *rerandomizing* the iterative slicing procedure, we obtain the improved complexities depicted in Figure 1.

Note that the costs for the second method are better, but are based on the additional assumption that running the randomized slicer $N$ times for the *same target* increases the success probability of finding the closest vector by a factor approximately $N$. Also note that the curves for both methods keep decreasing as the available preprocessed space increases – for arbitrary $\varepsilon > 0$, one can achieve an amortized time complexity for CVP of $2^{\varepsilon d+o(d)}$ at the cost of $(1/\varepsilon)^{O(d)}$ preprocessed space.

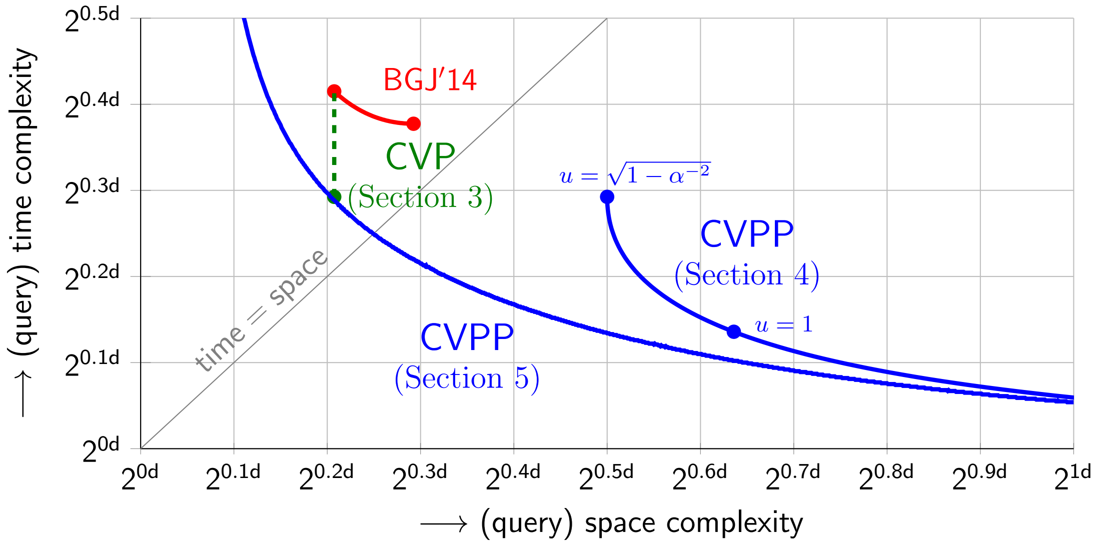


**Fig. 1.** Heuristic query complexities for solving CVP and CVPP. The red curve shows the previous best asymptotic CVP complexities of Becker–Gama–Joux [BGJ14], the dashed green line denotes our CVP complexities. The rightmost blue curve shows the complexities for CVPP without rerandomizations, while the lower blue curve denotes the improved complexities using randomized slicing. Note that this curve passes just *below* the CVP curve, i.e. solving a batch of $2^{\varepsilon d}$ CVP instances for small $\varepsilon > 0$ can be done with the same asymptotic time and space complexities as solving one CVP instance.

**Experiments for solving CVPP.** Preliminary experiments for sieving as a CVPP-solver, based on the fast but asymptotically suboptimal HashSieve [Laa15a], support the conjectured heuristic speedups/complexities, and the additional assumption behind rerandomizing the slicing procedure. In dimension 50, we obtain a speedup of a factor approximately 2000 for solving CVPP compared to solving SVP with the HashSieve on the same lattice. These first results suggest that the crossover point between sieving and enumeration may be in much lower dimensions for CVPP than for SVP or CVP.

**Variants of CVP(P).** For easier variants of CVP, such as when the target lies closer to the lattice than expected or an approximate solution to CVP suffices as a solution, we obtain considerable gains in both the time and space complexities when using preprocessing. For instance, approximate CVPP with approximation factor 2 can heuristically be solved in time and space $2^{0.05d+o(d)}$, with preprocessing costs similar to the costs of solving SVP with sieving. We further obtain polynomial time and space complexities for approximate CVPP iff the desired approximation factor scales as $\Omega(\sqrt{d/\log d})$. This heuristically closes the gap between the decision-CVPP approximation factor of $\Omega(\sqrt{d/\log d})$ of [AR04] and the search-CVPP approximation factor of $\Omega(d/\sqrt{\log d})$ of [DRS14] required to obtain polynomial time and space complexities for solving approximate CVPP.

**Outline.** In Section 2 we describe preliminaries on lattices, lattice sieving, Voronoi cells, and nearest neighbor searching. Section 3 describes how to solve CVP with equivalent asymptotic costs as SVP. Section 4 describes how to solve CVPP without randomizing the slicing algorithm, while Section 5 continues with a more general analysis based on arbitrary success probabilities for the slicer, and a way to randomize the latter procedure. Section 6 describes experiments to verify the heuristic assumptions and the claimed speedups. Sections 7 and 8 discuss variants of CVPP, and how the asymptotic complexities change for the preprocessing approach. Finally, Section 9 concludes with open problems.

## 2 Preliminaries

### 2.1 Problem statements

**Problems without preprocessing.** We first recall the definitions of some common hard lattice problems, and the problems often described in the (approximate) nearest neighbor searching literature. The problems below are without preprocessing.

**Definition 1 (Shortest Vector Problem (SVP)).** *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$, find a non-zero vector $\boldsymbol{s} \in \mathcal{L}$ such that $\|\boldsymbol{s}\| = \min_{\boldsymbol{v} \in \mathcal{L} \setminus \{\boldsymbol{0}\}} \|\boldsymbol{v}\|$.*

**Definition 2 (Closest Vector Problem (CVP)).** *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$ and a target vector $\boldsymbol{t} \in \mathbb{R}^d$, find a vector $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s} - \boldsymbol{t}\| = \min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

**Definition 3 (Bounded Distance Decoding (BDD$_\delta$)).** *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$, a target vector $\boldsymbol{t} \in \mathbb{R}^d$, and a distance guarantee $\delta \in (0, 1)$ such that $\min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\| \leq \delta \lambda_1(\mathcal{L})$, find a vector $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s} - \boldsymbol{t}\| = \min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

**Definition 4 (Approximate Closest Vector Problem (CVP$_\kappa$)).** *Given a description of a lattice $\mathcal{L} \subset \mathbb{R}^d$, a target vector $\boldsymbol{t} \in \mathbb{R}^d$, and an approximation factor $\kappa \geq 1$, find a vector $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s} - \boldsymbol{t}\| \leq \kappa \cdot \min_{\boldsymbol{v} \in \mathcal{L}} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

**Problems with preprocessing.** We denote the preprocessing versions of CVP, CVP$_\kappa$, and BDD$_\delta$, by CVPP, CVPP$_\kappa$, and BDDP$_\delta$ respectively. In the preprocessing variants, the lattice is given first and may be preprocessed so that, when given a target vector later, a solution to the problem can be provided faster than without preprocessing the lattice basis. For SVP clearly a preprocessing variant does not make sense, as the shortest vector could be precomputed.

Related to nearest neighbor searching, we recall the following problems. These are all problems where preprocessing is essential, and the most general statements of these problems are given below.

**Definition 5 (Nearest Neighbor Searching (NNS)).** *Given a data set $L \subset \mathbb{R}^d$, preprocess this data in such a way that, when given a target vector $\boldsymbol{t} \in \mathbb{R}^d$ later, one can quickly find a vector $\boldsymbol{s} \in L$ such that $\|\boldsymbol{s} - \boldsymbol{t}\| = \min_{\boldsymbol{v} \in L} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

**Definition 6 (Approximate Nearest Neighbors (ANN$_c$)).** *Given a data set $L \subset \mathbb{R}^d$ and an approximation factor $c \geq 1$, preprocess the data in such a way that when given a target vector $\boldsymbol{t} \in \mathbb{R}^d$ later, one can quickly find a vector $\boldsymbol{s} \in L$ such that $\|\boldsymbol{s} - \boldsymbol{t}\| \leq c \cdot \min_{\boldsymbol{v} \in L} \|\boldsymbol{v} - \boldsymbol{t}\|$.*

NNS is essentially equivalent to CVPP, except that (1) the data set in nearest neighbor searching is not assumed to be structured, and (2) the data set is assumed to be of finite cardinality $n < \infty$. Naive algorithms for nearest neighbor searching take $O(n)$ time and $O(n)$ space without any preprocessing costs, and the literature commonly focuses on sublinear time algorithms, running in time $O(n^\rho)$ for $\rho < 1$, commonly with $O(n^{1+\rho})$ space and preprocessing costs. Note that in the context of lattice sieving one commonly has $n = 2^{\Theta(d)}$, whereas the literature commonly focuses on the case $n = 2^{o(d)}$; therefore it is not clear whether lower bounds on the query complexity for (approximate) nearest neighbor searching (e.g. [ALRW17, Chr17]) also apply in the context of lattice sieving.

### 2.2 Nearest neighbor algorithms

A celebrated technique for finding near neighbors in high dimensions is Locality-Sensitive Hashing (LSH) [IM98, Cha02, AI06, AR15, AIL$^+$15, WSSJ14], where the idea is to construct many random partitions of the space, and store the list $L$ in hash tables with buckets corresponding to regions. Preprocessing then consists of constructing these hash tables, and a query $\boldsymbol{t}$ is answered by doing a lookup in each of the hash tables, and searching for a(n approximate) nearest neighbor in the buckets containing $\boldsymbol{t}$. For a data set of size $|L| = n$, this commonly leads to a sublinear time complexity $O(n^\rho)$ ($\rho < 1$). LSH has also been used to speed up lattice sieving, and more details on this can be found in e.g. [Laa15a, LdW15, BGJ15, BL16], as well as implicitly in [WLTB11, ZPH13].

Similar to locality-sensitive hash functions, Locality-Sensitive Filters (LSF) [BDGL16, Laa15b, ALRW17, Chr17] divide the space into regions, with the added relaxation that these regions do not have to form a partition; regions may overlap, and part of the space may not even be covered by any region. This turns out to lead to improved results compared to known LSH techniques when $n$ is exponential in $d$ [BDGL16, Laa15b], and this allows for more natural time-space tradeoffs compared to LSH for arbitrary $n$ [Laa15b, ALRW17, Chr17]. For the case of $n = 2^{o(d)}$, this leads to optimal tradeoffs for nearest neighbor searching within certain frameworks [ALRW17, Chr17].

Below we restate the main result of [Laa15b] for our applications, where $n$ is assumed to be exponential in $d$. The specific problem considered here is: given a data set $L$ sampled uniformly at random from the unit sphere $\mathcal{S}^{d-1}$, and a random query $\boldsymbol{t} \in \mathcal{S}^{d-1}$, return a vector $\boldsymbol{w} \in L$ such that the angle between $\boldsymbol{w}$ and $\boldsymbol{t}$ is at most $\theta \in (0, \frac{\pi}{2})$. The following result further assumes that the list $L$ contains exactly $n = (1/\sin\theta)^{d+o(d)}$ vectors, denoted the *critical density* in [Laa15b]. The following is a restatement of [Laa15b, Corollary 1].

**Lemma 1 (Nearest neighbor costs for spherical data sets).** *Let $\theta \in (0, \frac{1}{2}\pi)$, and let $u \in [\cos\theta, 1/\cos\theta]$. Let $L \subset \mathcal{S}^{d-1}$ be a list of $n = (1/\sin\theta)^{d+o(d)}$ vectors sampled uniformly at random from $\mathcal{S}^{d-1}$. Then, using spherical LSF with parameters $\alpha_{\mathrm{q}} = u\cos\theta$ and $\alpha_{\mathrm{u}} = \cos\theta$, one can preprocess $L$ in time $n^{1+\rho_{\mathrm{u}}+o(1)}$, using $n^{1+\rho_{\mathrm{u}}+o(1)}$ space, and with high probability answer a random query $\boldsymbol{t} \in \mathcal{S}^{d-1}$ correctly in time $n^{\rho_{\mathrm{q}}+o(1)}$, where:*

$$n^{\rho_{\mathrm{q}}} = \left(\frac{\sin^2\theta\,(u\cos\theta + 1)}{u\cos\theta - \cos 2\theta}\right)^{d/2}, \qquad n^{\rho_{\mathrm{u}}} = \left(\frac{\sin^2\theta}{1 - \cot^2\theta\,(u^2 - 2u\cos\theta + 1)}\right)^{d/2}. \tag{1}$$

### 2.3 Lattice sieving algorithms

Heuristic lattice sieving algorithms for solving SVP are based on the following two observations: (1) if $\boldsymbol{v}, \boldsymbol{w} \in \mathcal{L}$, then their sum/difference $\boldsymbol{v} \pm \boldsymbol{w}$ is also a lattice vector; and (2) if we have a sufficiently long list $L$ of lattice vectors, then we expect there to be pairs $\boldsymbol{v}, \boldsymbol{w} \in L$ with $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \|\boldsymbol{v}\|, \|\boldsymbol{w}\|$. This intuitively describes the approach: we generate a long list of lattice vectors, and keep combining vectors in our list to form shorter and shorter lattice vectors until we (hopefully) find a shortest lattice vector in our list.

To make sure the algorithm makes progress in finding shorter lattice vectors, $L$ needs to contain many lattice vectors; for vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathcal{L}$ of similar norm, the vector $\boldsymbol{v} - \boldsymbol{w}$ is shorter than $\boldsymbol{v}, \boldsymbol{w}$ iff the angle between $\boldsymbol{v}, \boldsymbol{w}$ is smaller than $\pi/3$, which for random vectors $\boldsymbol{v}, \boldsymbol{w}$ of similar norm would occur with probability $(3/4)^{d/2+o(d)}$. This is exactly the assumption used in analyzing these heuristic sieving algorithms: when normalized, vectors in $L$ follow the same distribution as vectors sampled uniformly at random from the unit sphere.

**Heuristic assumption 1.** *When normalized, the list vectors $\boldsymbol{w} \in L$ behave as i.i.d. uniformly distributed random vectors from the unit sphere $\mathcal{S}^{d-1} := \{\boldsymbol{x} \in \mathbb{R}^d : \|\boldsymbol{x}\| = 1\}$.*

Note that this is only a simplifying assumption to facilitate a (hopefully) more accurate analysis of the costs of sieving. This assumption may well be false if the lattice $\mathcal{L}$ contains additional structure, and may become less precise as the list vectors become shorter and the discrete nature of the lattice presents itself more.

The expected space complexity of heuristic sieving algorithms follows from the previous observation: if we sample $(4/3)^{d/2+o(d)}$ vectors uniformly at random from the unit sphere, then we expect a significant number of pairs of vectors to have angle less than $\pi/3$, leading to many short difference vectors. Therefore, if we start by sampling a list $L$ of $(4/3)^{d/2+o(d)}$ long lattice vectors, and iteratively consider combinations of vectors in $L$ to find shorter vectors, we expect to keep making progress. Naively, combining all pairs of vectors in a list of size $(4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}$ takes time at least $(4/3)^{d+o(d)} \approx 2^{0.415d+o(d)}$.

**The Nguyen-Vidick sieve.** The heuristic sieve of Nguyen and Vidick [NV08] starts by sampling a list $L$ of $(4/3)^{d/2+o(d)}$ reasonably long lattice vectors, sampled from a discrete Gaussian with large standard deviation, and uses a *sieve* to map $L$, with maximum norm $R := \max_{\boldsymbol{v}\in L} \|\boldsymbol{v}\|$, to a new list $L'$, with maximum norm at most $R' := \gamma R$ for $0 \ll \gamma < 1$ close to 1. By repeatedly applying this sieve, after $\operatorname{poly}(d)$ iterations we expect to find a long list of lattice vectors of norm at most $\gamma^{\operatorname{poly}(d)} R = O(\lambda_1(\mathcal{L}))$, which then (with high probability) contains a shortest vector in the lattice. Algorithm 1 describes a sieve equivalent to Nguyen-Vidick's original sieve, to map $L$ to $L'$ in $|L|^2$ time (ignoring costs polynomial in $d$). The presented algorithm is a more intuitive but equivalent version of the original sieve; see [Laa15a, Appendix B] for details on this equivalence.

**Algorithm 1** The quadratic Nguyen-Vidick sieve for finding shortest vectors

```
Require: An LLL-reduced basis $B$ of a lattice $\mathcal{L}(B)$
Ensure: The algorithm finds a shortest lattice vector
1: Initialize empty lists $L, L'$ and set $\gamma \leftarrow 1 - 1/d$
2: Sample $(4/3)^{d/2+o(d)}$ lattice vectors and add them to $L$
3: Set $R \leftarrow \max_{\boldsymbol{w}\in L} \|\boldsymbol{w}\|$
4: repeat
5:     for each $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ do          ▷ NNS techniques can be used to speed this up
6:         if $\|\boldsymbol{w}_1 - \boldsymbol{w}_2\| < \gamma R$ then
7:             Add $\boldsymbol{w}_1 - \boldsymbol{w}_2$ to the list $L'$
8:     Replace $L \leftarrow L'$, set $L' \leftarrow \emptyset$, and recompute $R \leftarrow \max_{\boldsymbol{w}\in L} \|\boldsymbol{w}\|$
9: until $L$ contains a shortest lattice vector
```

Without any further modifications to this algorithm, the heuristic complexities for solving the shortest vector problem with this method are as follows, as described in [NV08, Section 4].

**Lemma 2 (Complexities of the Nguyen–Vidick sieve for SVP).** *Assuming heuristic assumption 1. holds, the Nguyen-Vidick sieve solves SVP in space* S *and time* T*, with*

$$\mathrm{S} = (4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}, \qquad \mathrm{T} = (4/3)^{d+o(d)} \approx 2^{0.415d+o(d)}. \tag{2}$$

By applying more sophisticated techniques for indexing the list $L$ and searching for pairs of vectors that can be combined to form shorter vectors, the time complexity can be further reduced to $(3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}$ [BDGL16]. Using a trick first described in [BGJ15], this can be done without increasing the space complexity, i.e. maintaining an asymptotic space complexity of only $2^{0.208d+o(d)}$, as the following result (a restatement of [BDGL16, Corollary 8]) shows.

**Lemma 3 (Complexities of the optimized Nguyen–Vidick sieve for SVP).** *Assuming that heuristic assumption 1. holds, the Nguyen-Vidick sieve with spherical LSF solves SVP in space* $\mathrm{S}$ *and time* $\mathrm{T}$*, with*

$$\mathrm{S} = (4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}, \qquad \mathrm{T} = (3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}. \tag{3}$$

**Micciancio and Voulgaris' GaussSieve.** Micciancio and Voulgaris used a slightly different approach in their GaussSieve algorithm [MV10b]. This algorithm reduces the memory usage by immediately *reducing* all pairs of lattice vectors that can be combined to form shorter lattice vectors. The algorithm uses a single list $L$, which is continuously kept in a state where for all $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$, $\|\boldsymbol{w}_1 \pm \boldsymbol{w}_2\| \geq \|\boldsymbol{w}_1\|, \|\boldsymbol{w}_2\|$. Each time a new vector $\boldsymbol{v} \in \mathcal{L}$ is sampled, its norm is reduced with vectors in $L$ by adding/subtracting vectors $\boldsymbol{w} \in L$ which would lead to a shorter vector. After $\boldsymbol{v}$ can no longer be reduced with $L$, the vectors in $L$ are reduced with $\boldsymbol{v}$, so that if $\boldsymbol{v}$ is finally added to the list, the pairwise reduction property is maintained. Modified list vectors are added to a stack to be reconsidered later. Algorithm 2 describes this procedure using pseudocode.

By immediately reducing all pairs of vectors, the GaussSieve achieves significantly better practical time and space complexities than the Nguyen-Vidick sieve. At the same time however, Nguyen and Vidick's (heuristic) proof technique does not apply to the GaussSieve, and there is no proven theoretical bound on the time complexity of the GaussSieve, even using heuristic assumptions. However, it is commonly believed that the Nguyen-Vidick sieve and the GaussSieve have the same heuristic asymptotic space and time complexities, i.e. using $2^{0.208d+o(d)}$ space and $2^{0.415d+o(d)}$ time without any further modifications, and using only $2^{0.292d+o(d)}$ time using nearest neighbor searching [BDGL16]. However, to apply nearest neighbor techniques the space complexity would increase to $2^{0.292d+o(d)}$ as well, as the same trick from e.g. [BGJ15, Laa15a] cannot be applied here.

**Algorithm 2** The GaussSieve algorithm for finding shortest vectors

```
Require: A basis $B$ of a lattice $\mathcal{L}(B)$
Ensure: The algorithm finds a shortest lattice vector
 1: Initialize an empty list $L$ and an empty stack $S$
 2: repeat
 3:     Get a vector $\boldsymbol{v}$ from the stack (or sample a new one if $S = \emptyset$)
 4:     for each $\boldsymbol{w} \in L$ do                 ▷ NNS techniques can be used to speed this up
 5:         if $\|\boldsymbol{v} - \boldsymbol{w}\| < \|\boldsymbol{v}\|$ then
 6:             Replace $\boldsymbol{v} \leftarrow \boldsymbol{v} - \boldsymbol{w}$
 7:         if $\|\boldsymbol{w} - \boldsymbol{v}\| < \|\boldsymbol{w}\|$ then
 8:             Replace $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{v}$
 9:             Move $\boldsymbol{w}$ from the list $L$ to the stack $S$ (unless $\boldsymbol{w} = \boldsymbol{0}$)
10:     if $\boldsymbol{v}$ has changed then
11:         Add $\boldsymbol{v}$ to the stack $S$ (unless $\boldsymbol{v} = \boldsymbol{0}$)
12:     else
13:         Add $\boldsymbol{v}$ to the list $L$ (unless $\boldsymbol{v} = \boldsymbol{0}$)
14: until $\boldsymbol{v}$ is a shortest lattice vector
```

### 2.4 Voronoi cells

We recall some definitions and results regarding the Voronoi cell of a lattice from [VB96, AEVZ02, SFS09, MV10a]. First, the Voronoi cell of a lattice is essentially the region of points closer to the origin than to any other lattice point.

**Definition 7 (Voronoi cell of a lattice).** *The Voronoi cell of a lattice $\mathcal{L}$ is defined as the region $\mathcal{V} \subset \mathbb{R}^d$ such that $\boldsymbol{v} \in \mathcal{V}$ iff $\|\boldsymbol{v}\| \leq \|\boldsymbol{v} - \boldsymbol{x}\|$ for all $\boldsymbol{x} \in \mathcal{L} \setminus \{\boldsymbol{0}\}$.*

The Voronoi cell of a lattice is a convex polytope, and its facets are closely related to the *relevant vectors*, defined below.

**Definition 8 (Relevant vectors).** *Given a lattice $\mathcal{L}$, a vector $\boldsymbol{r} \in \mathcal{L}$ is a relevant vector of the lattice iff $\mathcal{V}$ and $\boldsymbol{r} + \mathcal{V}$ share a non-empty boundary. We denote the set of all relevant vectors by $\mathcal{R}$.*

The relevant vectors of the lattice shape the boundary of $\mathcal{V}$, and the set $\mathcal{R}$ can be seen as a way to represent/store the Voronoi cell of the lattice. Fortunately the size of $\mathcal{R}$ is finite, and can be bounded as follows (see e.g. [MV10a, Corollary 2.5] for a proof).

**Lemma 4 (Number of relevant vectors).** *The set $\mathcal{R}$ has cardinality less than $2^{d+1}$.*

As a result, a description of the Voronoi cell of a lattice can be stored in $2^{d+o(d)}$ memory. Given this description, Micciancio and Voulgaris [MV10a] described algorithms with proven time complexities at most quadratic in the space complexity (i.e. $2^{2d+o(d)}$) for solving SVP, CVP and CVPP, and described a way to construct $\mathcal{R}$ in $2^{2d+o(d)}$ time and $2^{d+o(d)}$ space. Bonifas and Dadush later showed how to improve the time complexity for CVPP to $2^{d+o(d)}$, by bounding the number of iterations of the algorithm to $\operatorname{poly}(d)$.

For completeness, Algorithm 3 describes the lattice slicer of Sommer–Feder–Shalvi [SFS09] for solving CVPP for a target vector $\boldsymbol{t}$, given the Voronoi cell of the lattice as input, within a finite number of steps [SFS09, Theorem 1]. Micciancio–Voulgaris later showed that by selecting relevant vectors for reduction in a specific order, the number of iterations can be bounded by $2^{d+o(d)}$ [MV10a, Lemma 3.2]. As described in [SFS09, Lemma 5], using a different list $L$ instead of $\mathcal{R}$, this algorithm succeeds in solving CVPP for arbitrary targets iff $\mathcal{R} \subseteq L$.

**Algorithm 3** The lattice slicer of [SFS09] for finding closest vectors

```
Require: The relevant vectors $\mathcal{R} \subset \mathcal{L}$ and a target $\boldsymbol{t}$
Ensure: The algorithm finds a closest lattice vector $\boldsymbol{s}$ to $\boldsymbol{t}$
1: Initialize $\boldsymbol{t}' \leftarrow \boldsymbol{t}$
2: for each $\boldsymbol{r} \in \mathcal{R}$ do
3:     if $\|\boldsymbol{t}' \pm \boldsymbol{r}\| < \|\boldsymbol{t}'\|$ then
4:         Replace $\boldsymbol{t}' \leftarrow \boldsymbol{t}' \pm \boldsymbol{r}$ and restart the for-loop
5: return $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}'$
```

By similar techniques as in heuristic lattice sieving (or as in [BD15]), one can bound the number of iterations of this slicer until termination. Given a target $\boldsymbol{t}$, one can use e.g. Babai rounding on an LLL-reduced basis of the lattice to get an initial guess $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$ satisfying $\|\boldsymbol{t}'\| \leq 2^{O(d)} \min_{\boldsymbol{s} \in \boldsymbol{t}+\mathcal{L}} \|\boldsymbol{s}\|$. Then, by only performing reductions whenever $\|\boldsymbol{t}' \pm \boldsymbol{r}\| < \gamma \|\boldsymbol{t}'\|$ for some geometric factor $\gamma = 1 - 1/d^k$ for certain $k > 1$, one can ensure that the number of iterations is polynomially bounded by $\log_{1/\gamma} \|\boldsymbol{t}'\| = O(d^{1+k})$. At the same time, due to this geometric factor $\gamma$, after the algorithm terminates we might only have $\boldsymbol{t}' \in (1/\gamma)\mathcal{V}$ instead of $\boldsymbol{t}' \in \mathcal{V}$. Since $\mathrm{Vol}(\mathcal{V}/\gamma) = \mathrm{Vol}(\mathcal{V})/\gamma^d$, we therefore expect this algorithm to succeed with probability proportional to $\gamma^d = 1 - O(d^{1-k}) = 1 - o(1)$ over the randomness of $\boldsymbol{t}$. As $k$ increases, the (polynomial) number of iterations increases further, while the success probability becomes (polynomially) overwhelming.

Bonifas and Dadush [BD15] described a different method to bound the number of iterations to $\mathrm{poly}(d)$, by carefully choosing which relevant vectors to use for reduction in each step. Although there is no formal proof that other approaches allow for solving exact CVP as well, in the remainder of this paper we will assume (heuristically) that the number of iterations of this slicer (until termination) is only $\mathrm{poly}(d)$.

## 3 Solving CVP

For solving CVP directly without any preprocessing, we adapt the entire Nguyen–Vidick sieve to the target vector $\boldsymbol{t}$, to obtain the best overall time complexity for solving one problem instance. When solving several CVP instances on the same lattice, the costs roughly scale linearly with the number of instances.

**Using one list.** The main idea behind this method is to translate the sieving algorithm of Nguyen and Vidick (for solving SVP) by the target vector $\boldsymbol{t}$; instead of generating a long list of lattice vectors reasonably close to $\mathbf{0}$, we generate a list of lattice vectors close to $\boldsymbol{t}$, and combine lattice vectors to find lattice vectors even closer to $\boldsymbol{t}$. The final list then hopefully contains a closest vector to $\boldsymbol{t}$.

One quickly realizes that the naive way to do this does not work, as the fundamental property of lattices does not hold for the lattice coset $\boldsymbol{t} + \mathcal{L}$: if $\boldsymbol{w}_1, \boldsymbol{w}_2 \in \boldsymbol{t} + \mathcal{L}$, then $\boldsymbol{w}_1 \pm \boldsymbol{w}_2 \notin \boldsymbol{t} + \mathcal{L}$. If we have two lattice vectors close to $\boldsymbol{t}$, then we can only combine them to form lattice vectors close to $\mathbf{0}$ or to $2\boldsymbol{t}$, but not to $\boldsymbol{t}$.

**Using two lists.** To make the idea of translating the whole problem by $\boldsymbol{t}$ work, we make the following modification: rather than using one list, we keep track of two lists $L = L_{\mathbf{0}}$ and $L_{\boldsymbol{t}}$ of lattice vectors close to $\mathbf{0}$ and $\boldsymbol{t}$ respectively, and we construct a sieve which maps two input lists $L_{\mathbf{0}}, L_{\boldsymbol{t}}$ to two output lists $L'_{\mathbf{0}}, L'_{\boldsymbol{t}}$ of lattice vectors slightly closer to $\mathbf{0}$ and $\boldsymbol{t}$. Similar to the original Nguyen-Vidick sieve, we then apply this sieve several times to two initial lists $(L_{\mathbf{0}}, L_{\boldsymbol{t}})$ with a large radius $R$, to end up with two final lists $L_{\mathbf{0}}$ and $L_{\boldsymbol{t}}$ of lattice vectors very close to $\mathbf{0}$ and $\boldsymbol{t}$. The reasoning that this algorithm works is almost identical to that for solving SVP with the Nguyen-Vidick sieve, where we now make the following slightly different heuristic assumption.

**Heuristic assumption 2.** *When normalized, the list vectors $L_{\mathbf{0}}$ and $L_{\boldsymbol{t}}$ in the modified Nguyen-Vidick sieve are distributed as i.i.d. uniformly random vectors from $\mathcal{S}^{d-1}$.*

The resulting algorithm, based on the Nguyen-Vidick sieve, is presented in Algorithm 4. Here the first for-loop is essentially identical to the Nguyen–Vidick sieve for solving SVP, while the second for-loop shows how to also construct a list of lattice vectors close to a given target $\boldsymbol{t}$. If one would like to solve CVP for $k$ targets $\boldsymbol{t}_1, \dots, \boldsymbol{t}_k$, one would simply adapt Algorithm 4 to use $k+1$ lists $L_{\mathbf{0}}, L_{\boldsymbol{t}_1}, \dots, L_{\boldsymbol{t}_k}$.

The (heuristic) correctness of this algorithm follows directly from the correctness of the original NV-sieve, together with the slightly different heuristic assumption described above. As nearest neighbor techniques can be applied to this algorithm in a similar way as for solving SVP, we immediately obtain the following result. Note that as we are using

**Algorithm 4** The Nguyen-Vidick sieve for finding closest vectors

```
Require: An LLL-reduced basis $B$ of a lattice $\mathcal{L}(B)$, a target $\boldsymbol{t} \in \mathbb{R}^d$
Ensure: The algorithm finds a closest lattice vector to $\boldsymbol{t}$
 1: Initialize empty lists $L_{\boldsymbol{0}}, L'_{\boldsymbol{0}}, L_{\boldsymbol{t}}, L'_{\boldsymbol{t}}$ and set $\gamma \leftarrow 1 - 1/d$
 2: Sample $(4/3)^{d/2+o(d)}$ vectors from $\mathcal{L}$ (resp. $\boldsymbol{t} + \mathcal{L}$) and add them to $L_{\boldsymbol{0}}$ (resp. $L_{\boldsymbol{t}}$)
 3: Set $R \leftarrow \max\{\max_{\boldsymbol{w} \in L_{\boldsymbol{0}}} \|\boldsymbol{w}\|, \max_{\boldsymbol{w} \in L_{\boldsymbol{t}}} \|\boldsymbol{w} - \boldsymbol{t}\|\}$
 4: repeat
 5:     for each $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L_{\boldsymbol{0}}$ do                ▷ NNS can be used to speed this up
 6:         if $\|\boldsymbol{w}_1 - \boldsymbol{w}_2\| < \gamma R$ then
 7:             Add $\boldsymbol{w}_1 - \boldsymbol{w}_2$ to the list $L'_{\boldsymbol{0}}$
 8:     for each $\boldsymbol{w}_1 \in L_{\boldsymbol{t}}$ and $\boldsymbol{w}_2 \in L_{\boldsymbol{0}}$ do    ▷ NNS can be used to speed this up
 9:         if $\|(\boldsymbol{w}_1 - \boldsymbol{w}_2) - \boldsymbol{t}\| < \gamma R$ then
10:             Add $\boldsymbol{w}_1 - \boldsymbol{w}_2$ to the list $L'_{\boldsymbol{t}}$
11:     Update $(L_{\boldsymbol{0}}, L'_{\boldsymbol{0}}, L_{\boldsymbol{t}}, L'_{\boldsymbol{t}}) \leftarrow (L'_{\boldsymbol{0}}, \emptyset, L'_{\boldsymbol{t}}, \emptyset)$, and recompute $R$ as above
12: until $L_{\boldsymbol{t}}$ contains a closest lattice vector to $\boldsymbol{t}$
```

the Nguyen-Vidick sieve, the space complexity does not increase using nearest-neighbor techniques, as we can process the hash tables sequentially, rather than simultaneously.

**Theorem 1 (Complexities of the optimized Nguyen–Vidick sieve for CVP).** *Assuming that heuristic assumption 2. holds, the Nguyen-Vidick sieve with spherical LSF solves CVP in space* S *and time* T, *with*

$$\mathrm{S} = (4/3)^{d/2+o(d)} \approx 2^{0.208d+o(d)}, \qquad \mathrm{T} = (3/2)^{d/2+o(d)} \approx 2^{0.292d+o(d)}. \tag{4}$$

An open question remains whether these techniques can also be applied to the faster GaussSieve algorithm, to solve CVP with better practical complexities. As the GaussSieve seems to make even more use of the property that the sum of two lattice vectors is also in the lattice, and operations in the GaussSieve in $\mathcal{L}$ cannot as easily be translated to the lattice coset $\boldsymbol{t} + \mathcal{L}$ as for the Nguyen–Vidick sieve, it is not clear if this can be done.

## 4 Solving CVPP – Approximate Voronoi cells

The main results of this paper concern solving CVP *with preprocessing* with lattice sieving, i.e. reducing the amortized complexity for CVP beyond the complexities stated in the previous section. The first method we present is deterministic; although the preprocessing is potentially randomized, after being given the target vector the algorithm follows a deterministic procedure to find a closest vector in the lattice. For analyzing the performance of our CVPP algorithms, we split the algorithm in two phases, and we keep track of four costs of the algorithm:

- **Preprocessing phase**: Preprocess the lattice $\mathcal{L}$, without knowledge of the target $\boldsymbol{t}$;
  - $\mathrm{S}_1$: The memory used during the preprocessing phase;
  - $\mathrm{T}_1$: The time used during the preprocessing phase;
- **Query phase**: Process the query $\boldsymbol{t}$ and output a closest lattice vector $\boldsymbol{s} \in \mathcal{L}$ to $\boldsymbol{t}$;
  - $\mathrm{S}_2$: The memory used during the query phase;
  - $\mathrm{T}_2$: The time used during the query phase.

Intuitively the main goal is to reduce the complexities of the query phase $(\mathrm{S}_2, \mathrm{T}_2)$ compared to a non-preprocessed CVP algorithm. However, in any practical application we need to perform the preprocessing at least once, and therefore CVPP algorithms with enormous preprocessing costs may be useless even if the query complexities are great.

Also note that as we are interested in reducing the query complexity compared to solving CVP, and this usually comes at the cost of a higher preprocessing cost, we generally have $\mathrm{T}_2 \leq \mathrm{T} \leq \mathrm{T}_1$, where T is the corresponding asymptotic time complexity for CVP. As we will see later (cf. Figure 1), it is asymptotically possible to achieve $\mathrm{T}_2 \ll \mathrm{T} = \mathrm{T}_1$, i.e. achieve significantly better amortized CVP time complexities with preprocessing time and space complexities comparable to the costs for solving a single CVP instance.

## 4.1 Approximate Voronoi cells

Before describing the algorithm, we start with the definition of approximate Voronoi cells. Below we use the following definition of half spaces $\mathcal{H}(\boldsymbol{x}) := \{\boldsymbol{v} \in \mathbb{R}^d : \|\boldsymbol{v}\| \leq \|\boldsymbol{v} - \boldsymbol{x}\|\}$, whose boundaries are the hyperplanes normal to $\frac{1}{2}\boldsymbol{x}$.

**Definition 9 (Approximate Voronoi cells).** *Let $L \subseteq \mathcal{L} \setminus \{\mathbf{0}\}$. Then the approximate Voronoi cell generated by $L$ is defined as $\mathcal{V}_L := \bigcap_{\pm \boldsymbol{r} \in L} \mathcal{H}(\boldsymbol{r})$.*

Note that if $\mathcal{R} \subseteq L$, the approximate Voronoi cell generated by $L$ is exactly $\mathcal{V}$, while $\mathcal{R}$ is the smallest set with this property. To quantize the 'quality' of an approximate Voronoi cell $\mathcal{V}_L$ (or a list $L$), recall that the volume of the exact Voronoi cell $\mathcal{V}$, denoted $\mathrm{Vol}(\mathcal{V})$, is equal to the volume of the lattice $\mathrm{Vol}(\mathcal{L}) = \det(\mathcal{L}) = \sqrt{\det(B^T B)}$ where $B$ is any basis matrix of the lattice. If $\mathcal{R}$ is not contained in $L$, then $\mathcal{V}_L$ will have a larger volume, and so the following quantity can serve as a guideline as to how well $\mathcal{V}_L$ (or $L$) approximates $\mathcal{V}$ ($\mathcal{R}$).

**Definition 10 (Approximation factor).** *Given a lattice $\mathcal{L}$ and a list $L \subset \mathcal{L}$, we define the approximation factor for the approximate Voronoi cell $\mathcal{V}_L$ generated by $L$ as $A_L := \mathrm{Vol}(\mathcal{V}_L) / \mathrm{Vol}(\mathcal{V})$.*

Clearly $A_L \geq 1$ with equality iff $\mathcal{R} \subset L$. Now, the main result regarding approximate Voronoi cells which the heuristic algorithm from this section builds on, is the following, stating how big $L$ must be to obtain approximation factors approaching 1.

**Lemma 5 (Achieving approximation factor $1 + o(1)$).** *Let $L$ consist of the $\alpha^{d+o(d)}$ shortest vectors in a lattice $\mathcal{L}$. Then heuristically, $A_L = 1 + o(1)$ iff $\alpha \geq \sqrt{2} + o(1)$.*

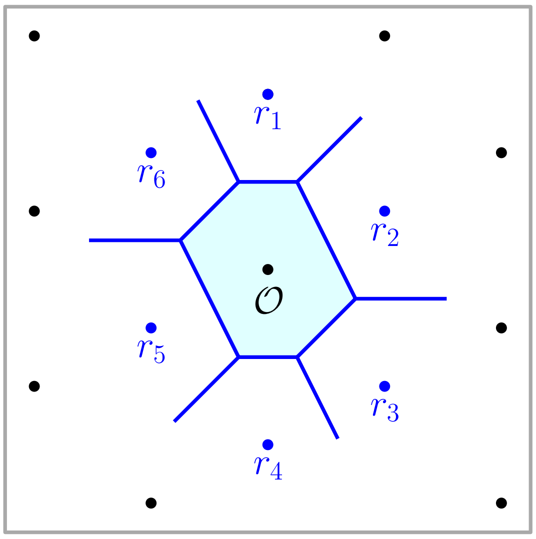
Lemma 5 essentially states that $|L| = \sqrt{|\mathcal{R}|}$ of the relevant vectors suffice to obtain a very good approximation of the Voronoi cell of the lattice – the other $|\mathcal{R}| - \sqrt{|\mathcal{R}|}$ relevant vectors only help to smoothen the corners, shaving off a negligible total volume. An example of exact and approximate Voronoi cells is given in Figure 2, where the approximate Voronoi cell is generated by a fraction 1/3 fewer vectors than the exact Voronoi cell, leading to a 1/4 increase in the volume of the approximate cell relative to the exact Voronoi cell.

Note that in the example of Figure 2, by taking $L = \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_5, \boldsymbol{r}_6\} \subset \mathcal{R}$ as the set of the four shortest vectors in $\mathcal{R}$, one can do even better and achieve approximation factor $A_L = 10/9$, using the same number of vectors as in Figure 2b.

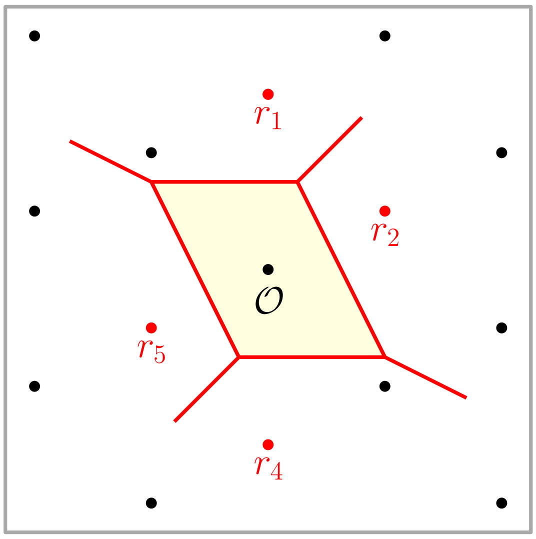
## 4.2 Proof of Lemma 5

To prove Lemma 5, we will prove the equivalent statement that with high probability, the slicing procedure returns a closest vector to the target with high probability if the preprocessed list contains the $2^{d/2+o(d)}$ shortest lattice vectors.

Suppose we have a target vector $\boldsymbol{t}$, which after reductions with $L$ turns into $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$, and suppose that $\boldsymbol{t}'$ is no longer reducible with $L$. In other words, $\boldsymbol{t}'$ is contained in $\mathcal{V}_L$. First, by the Gaussian heuristic (restated below), we expect the distances from $\boldsymbol{t}$ and $\boldsymbol{t}'$ to the lattice to be approximately $\lambda_1(\mathcal{L})$.



(a) The exact Voronoi cell $\mathcal{V}$ of the lattice $\mathcal{L}$, with volume $\mathrm{Vol}(\mathcal{V}) = \det(\mathcal{L}) = 6$.



(b) An approximate Voronoi cell $\mathcal{V}_L$ of $\mathcal{L}$ with volume $\mathrm{Vol}(\mathcal{V}_L) = \frac{15}{2}$, corresponding to $A_L = \frac{5}{4}$.

**Fig. 2.** Exact and approximate Voronoi cells of the lattice $\mathcal{L} \subset \mathbb{R}^2$ generated by $B = \{(2,1),(0,3)\}$.

**Heuristic assumption 3. (Gaussian heuristic)** *For random $\boldsymbol{t} \in \mathbb{R}^d$, a ball of radius $r \cdot \lambda_1(\mathcal{L})$ around $\boldsymbol{t}$ contains $r^{d+o(d)}$ lattice points. In particular, this ball is non-empty iff $r \geq 1 + o(1)$.*

To guarantee that $\mathbf{0}$ is the closest lattice vector to the reduced vector $\boldsymbol{t}'$, so that also $\boldsymbol{t}' \in \mathcal{V}$, we therefore heuristically need $\boldsymbol{t}'$ to have norm at most approximately $\lambda_1(\mathcal{L})$. We start with the following lemma regarding the probability of reduction between two uniformly random vectors with given norms.

**Lemma 6.** *Let $v, w > 0$ and let $\boldsymbol{v} = v \cdot \boldsymbol{e}_v$ and $\boldsymbol{w} = w \cdot \boldsymbol{e}_w$. Then:*

$$\mathbb{P}_{\boldsymbol{e}_v, \boldsymbol{e}_w \sim \mathcal{S}^{d-1}}\left(\|\boldsymbol{v} - \boldsymbol{w}\|^2 \leq \|\boldsymbol{v}\|^2\right) \sim \left[1 - \left(\frac{w}{2v}\right)^2\right]^{d/2+o(d)}. \tag{5}$$

*Proof.* Expanding $\|\boldsymbol{v} - \boldsymbol{w}\|^2 = v^2 + w^2 - 2vw \langle \boldsymbol{e}_v, \boldsymbol{e}_w \rangle$ and $\|\boldsymbol{v}\|^2 = v^2$, the condition $\|\boldsymbol{v} - \boldsymbol{w}\|^2 \leq \|\boldsymbol{v}\|^2$ equals $\frac{w}{2v} \leq \langle \boldsymbol{e}_v, \boldsymbol{e}_w \rangle$. The result follows from [BDGL16, Lemma 2.1].

For now, suppose that $|L| = \alpha^{d+o(d)}$, where we will later conclude that $\alpha \geq \sqrt{2}$ is necessary to guarantee correctness. Assuming Heuristic 1. holds, we obtain a relation between the choice of $\alpha$ for the input list size and the expected norm $\beta \cdot \lambda_1(\mathcal{L})$ of the reduced vector $\boldsymbol{t}'$ as follows.

**Lemma 7.** *Let $L \subset \alpha \cdot \mathcal{S}^{d-1}$ be a list of $\alpha^{d+o(d)}$ uniformly random vectors of norm $\alpha > 1$, and let $\boldsymbol{t} \in \beta \cdot \mathcal{S}^{d-1}$ be sampled uniformly at random. Then, for high dimensions $d$, with non-negligible probability there exists a $\boldsymbol{v} \in L$ such that $\|\boldsymbol{t} - \boldsymbol{v}\| \leq \|\boldsymbol{t}\|$ if and only if*

$$\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 \leq 0. \tag{6}$$

*Furthermore, if $\theta_{\boldsymbol{t},\boldsymbol{v}}$ denotes the angle between $\boldsymbol{t}$ and $\boldsymbol{v}$, then $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = 0$ and $\|\boldsymbol{v} - \boldsymbol{w}\| \leq \|\boldsymbol{v}\|$ together imply that $\theta_{\boldsymbol{t},\boldsymbol{v}} \leq \arcsin(1/\alpha)$.*

*Proof.* By Lemma 6 we can reduce $\boldsymbol{t}$ with $\boldsymbol{v} \in L$ with probability $p = [1 - \frac{\alpha^2}{4\beta^2}]^{d/2+o(d)}$. Since we have $n = \alpha^{d+o(d)}$ such vectors $\boldsymbol{v} \in L$, the probability that none of them can reduce $\boldsymbol{t}$ is $(1-p)^n$, which is $o(1)$ if $n \gg 1/p$ and $1 - o(1)$ if $n \ll 1/p$. Expanding $n \cdot p$, we obtain the given equation (6), where $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 > 0$ implies $n \ll 1/p$.

For the second part, consider the triangle formed by $\boldsymbol{0}, \boldsymbol{t}, \boldsymbol{v}$. If $\|\boldsymbol{t} - \boldsymbol{v}\| = \|\boldsymbol{t}\|$, then this triangle has two sides $\beta$ and one side $\alpha$, and two angles $\theta_{\boldsymbol{t},\boldsymbol{v}}$ and one angle $\pi - 2\theta_{\boldsymbol{t},\boldsymbol{v}}$. By the sine law, $\alpha \sin\theta_{\boldsymbol{t},\boldsymbol{v}} = \beta \sin(\pi - 2\theta_{\boldsymbol{t},\boldsymbol{v}}) = 2\beta \sin\theta_{\boldsymbol{t},\boldsymbol{v}} \cos\theta_{\boldsymbol{t},\boldsymbol{v}}$. Simplifying, we get $\cos\theta_{\boldsymbol{t},\boldsymbol{v}} = \alpha/(2\beta)$, or $\sin^2\theta_{\boldsymbol{t},\boldsymbol{v}} = 1 - \alpha^2/(4\beta^2)$. Multiplying by $\alpha^2$ yields

$$\alpha^2 \sin^2\theta_{\boldsymbol{t},\boldsymbol{v}} = \frac{4\beta^2\alpha^2 - \alpha^4}{4\beta^2} = \frac{4\beta^2}{4\beta^2} = 1, \tag{7}$$

where the next-to-last equality follows from $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = 0$. Therefore $\|\boldsymbol{t} - \boldsymbol{v}\| = \|\boldsymbol{t}\|$ corresponds to $\sin\theta_{\boldsymbol{t},\boldsymbol{v}} = 1/\alpha$, or equivalently $\theta_{\boldsymbol{t},\boldsymbol{v}} = \arcsin(1/\alpha)$. If $\|\boldsymbol{t} - \boldsymbol{v}\| < \|\boldsymbol{t}\|$, then the angle $\theta_{\boldsymbol{t},\boldsymbol{v}}$ further decreases, leading to $\theta_{\boldsymbol{t},\boldsymbol{v}} \le \arcsin(1/\alpha)$.

As a result of the second part of the previous lemma, reducing $\boldsymbol{t}$ with $L$ can be done by searching for vectors $\boldsymbol{v} \in L$ at angle at most $\theta_{\boldsymbol{t},\boldsymbol{v}} = \arcsin(1/\alpha)$ from $\boldsymbol{t}$: if $\boldsymbol{t}$ can be reduced with some vector $\boldsymbol{v} \in L$, then with high probability a vector $\boldsymbol{v}$ at this angle from $\boldsymbol{t}$ exists which can reduce $\boldsymbol{t}$. This will be necessary for applying Lemma 1 later, as that lemma assumes that the list size $|L| = n$ and target angle $\theta$ satisfy the relation $n = (1/\sin\theta)^{d+o(d)}$. In our setting $n = \alpha^{d+o(d)}$ and $\theta = \arcsin(1/\alpha)$, which means this relation is satisfied.

Note that we do not just assume that $L$ contains $\alpha^{d+o(d)}$ lattice vectors of norm approximately $\alpha \cdot \lambda_1(\mathcal{L})$: we heuristically expect to find almost all shortest vectors in $\mathcal{L}$, including all those vectors even shorter than $\alpha \cdot \lambda_1(\mathcal{L})$. In other words, for any $\alpha_0 \in [1, \alpha]$ we expect $L$ to contain $\alpha_0^{d+o(d)}$ lattice vectors of norm at most $\alpha_0 \cdot \lambda_1(\mathcal{L})$. To obtain a reduced vector $\boldsymbol{t}'$ of norm $\beta \cdot \lambda_1(\mathcal{L})$, we therefore obtain the condition that for *some* value $\alpha_0 \in [1, \alpha]$, it must hold that $\alpha_0^4 - 4\beta^2\alpha_0^2 + 4\beta_0^2 \le 0$. Factoring the LHS of (6) in terms of its roots for $\alpha$ yields

$$p(\alpha) = \alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = \left(\alpha^2 - 2\beta\left(\beta - \sqrt{\beta^2 - 1}\right)\right)\left(\alpha^2 - 2\beta\left(\beta + \sqrt{\beta^2 - 1}\right)\right). \tag{8}$$

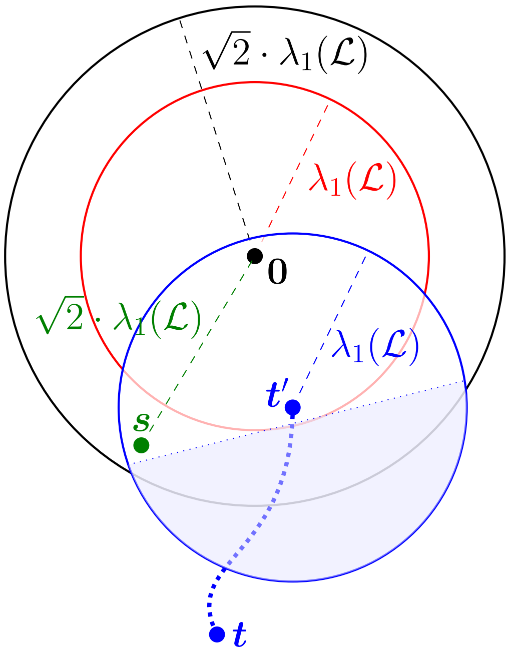
The polynomial $p(\alpha)$ has two roots $r_1 < \sqrt{2} < r_2$ (both with multiplicity 2), which both lie close to $\sqrt{2}$ if $\beta \approx 1$. The condition that $p(\alpha_0) \le 0$ for some $\alpha_0 \le \alpha$ is equivalent to the condition that $\alpha$ is at least equal to the smallest root:

$$\alpha \ge r_1 = \sqrt{2\beta\left(\beta - \sqrt{\beta^2 - 1}\right)}. \tag{9}$$

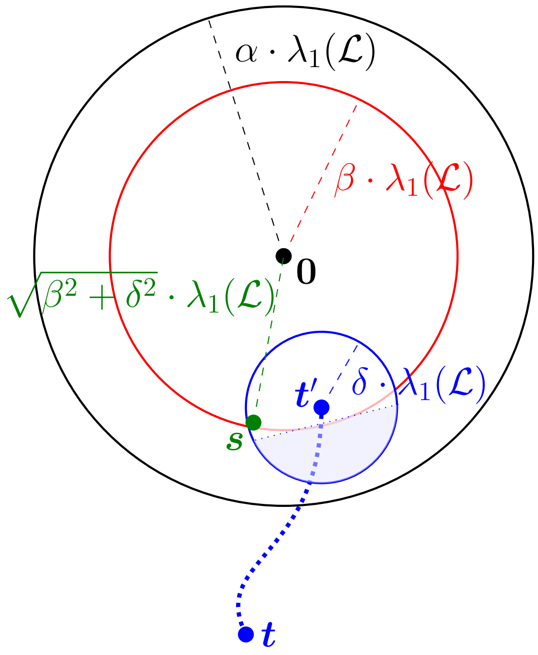
For $\beta = 1 + o(1)$ this implies that $\alpha \ge \sqrt{2} + o(1)$, and we must use $n = 2^{d/2+o(d)}$ initial vectors to guarantee that w.h.p. the algorithm succeeds. This proves that (i) using $2^{d/2+o(d)}$ vectors suffices to guarantee correctness with high probability, and (ii) using $|L| = 2^{(1/2-\varepsilon)d+o(d)}$ vectors will lead to an exponentially small success probability, for fixed $\varepsilon > 0$. A sketch of the above analysis is also given in Figure 3a.

## 4.3 Algorithm description

The query phase of the CVPP algorithm now follows directly from applying the slicer of [SFS09] described in Algorithm 3 (or equivalently the GaussSieve reduction step [MV10b]) to the target vector $\boldsymbol{t}$ and a preprocessed list $L \subset \mathcal{L}$ of size $2^{d/2+o(d)}$. Algorithm 5 describes



(a) For solving CVPP, we must reduce $\boldsymbol{t}$ to a vector $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$ of norm at most $\lambda_1(\mathcal{L})$. The nearest lattice point to $\boldsymbol{t}'$ lies in a ball of radius approximately $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$ (blue), and almost all the mass of this ball is contained in the (black) ball around $\mathbf{0}$ of radius $\sqrt{2} \cdot \lambda_1(\mathcal{L})$. So if $\boldsymbol{s} \in \mathcal{L} \setminus \{\mathbf{0}\}$ had lain closer to $\boldsymbol{t}'$ than $\mathbf{0}$, we would have reduced $\boldsymbol{t}'$ with $\boldsymbol{s}$, since $\boldsymbol{s} \in L$.



$\alpha \cdot \lambda_1(\mathcal{L})$
$\beta \cdot \lambda_1(\mathcal{L})$
$\mathbf{0}$
$\sqrt{\beta^2 + \delta^2} \cdot \lambda_1(\mathcal{L})$
$\boldsymbol{t}'$
$\delta \cdot \lambda_1(\mathcal{L})$
$\boldsymbol{s}$
$\boldsymbol{t}$

(b) For variants of CVPP, a choice $\alpha$ for the list size implies a norm $\beta \cdot \lambda_1(\mathcal{L})$ of $\boldsymbol{t}'$. The nearest lattice vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ lies within $\delta \cdot \lambda_1(\mathcal{L})$ of $\boldsymbol{t}'$ ($\delta = 1$ for approx-CVP), so with high probability $\boldsymbol{s}$ has norm approximately $(\sqrt{\beta^2 + \delta^2}) \cdot \lambda_1(\mathcal{L})$. For BDD$_\delta$, if $\sqrt{\beta^2 + \delta^2} \leq \alpha$ then we expect the nearest point $\boldsymbol{s}$ to be in the list $L$. For CVP$_\kappa$, if $\beta \leq \kappa$, then the lattice vector $\boldsymbol{t} - \boldsymbol{t}'$ has norm at most $\kappa \cdot \lambda_1(\mathcal{L})$.

**Fig. 3.** Comparison of the list size complexity analyses for CVPP (left) and BDD/approximate CVPP (right), without rerandomizations. The point $\boldsymbol{t}$ represents the target vector, and after a series of reductions with the sieve of Algorithm 5, we obtain a shorter vector $\boldsymbol{t}' \in \boldsymbol{t} + \mathcal{L}$, for which we hope that the closest lattice vector is $\mathbf{0}$. The blue balls around $\boldsymbol{t}'$ illustrate the region where we expect the closest lattice point $\boldsymbol{s}$ to $\boldsymbol{t}'$ to lie, where the blue shaded area indicates a negligible part of this ball in high dimensions by [BDGL16, Lemma 2]. Using rerandomizations, we treat $\boldsymbol{s}$ as if it were sampled uniformly at random from the blue ball, and use this distribution to compute the probability $p$ that the algorithm succeeds.

**Algorithm 5** The query phase for solving CVPP

```
Require: A list $L \subset \mathcal{L}$ and a target $\boldsymbol{t} \in \mathbb{R}^d$
Ensure: The output vector $\boldsymbol{s}$ is a close lattice vector to $\boldsymbol{t}$
1: Initialize $\boldsymbol{t}' \leftarrow \boldsymbol{t}$
2: for each $\boldsymbol{w} \in L$ do                                  ▷ NNS can be used to speed this up
3:     if $\|\boldsymbol{t}' - \boldsymbol{w}\| < \|\boldsymbol{t}'\|$ then
4:         Replace $\boldsymbol{t}' \leftarrow \boldsymbol{t}' - \boldsymbol{w}$ and restart the for-loop
5: return $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}'$
```

how to answer queries, given a preprocessed list containing the $\alpha^{d+o(d)}$ shortest lattice vectors – in our applications, $\alpha = \sqrt{2}$.

For the preprocessing phase, i.e. generating the list of the $\alpha^{d+o(d)}$ lattice vectors of norm at most $\alpha \cdot \lambda_1(\mathcal{L})$, we can use different methods. In moderate dimensions, the fastest way may be to use lattice enumeration [GNR10,KF16], but asymptotically heuristic lattice sieving will most likely lead to the best complexities. As we are interested in (heuristic) asymptotics, let us consider the preprocessing costs for sieving.

**Algorithm 6** The GaussSieve-based preprocessing phase for solving CVPP

```
Require: A basis $B$ of a lattice $\mathcal{L}(B)$, a parameter $\alpha \geq \sqrt{4/3}$
Ensure: The output list $L \subset \mathcal{L}$ contains $\alpha^{d+o(d)}$ vectors of norm at most $\alpha \cdot \lambda_1(\mathcal{L})$
 1: Initialize an empty list $L$ and an empty stack $S$
 2: repeat
 3:     Get a vector $\boldsymbol{v}$ from the stack (or sample a new one if $S = \emptyset$)
 4:     for each $\boldsymbol{w} \in L$ do                              ▷ NNS can be used to speed this up
 5:         if $\|\boldsymbol{v} - \boldsymbol{w}\|^2 < (2 - \frac{2}{\alpha}\sqrt{\alpha^2 - 1}) \cdot \|\boldsymbol{v}\|^2$ then
 6:             Replace $\boldsymbol{v} \leftarrow \boldsymbol{v} - \boldsymbol{w}$
 7:         if $\|\boldsymbol{w} - \boldsymbol{v}\|^2 < (2 - \frac{2}{\alpha}\sqrt{\alpha^2 - 1}) \cdot \|\boldsymbol{w}\|^2$ then
 8:             Replace $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{v}$
 9:             Move $\boldsymbol{w}$ from the list $L$ to the stack $S$ (unless $\boldsymbol{w} = \mathbf{0}$)
10:     if $\boldsymbol{v}$ has changed then
11:         Add $\boldsymbol{v}$ to the stack $S$ (unless $\boldsymbol{v} = \mathbf{0}$)
12:     else
13:         Add $\boldsymbol{v}$ to the list $L$ (unless $\boldsymbol{v} = \mathbf{0}$)
14: until $\boldsymbol{v}$ is a shortest vector
15: return $L$
```

Recall that with standard heuristic sieving methods, we reduce pairs of lattice vectors iff their angle is at most $\theta = \frac{\pi}{3}$, resulting in a list of size $(\sin\theta)^{-d+o(d)}$. To generate a list of the $\alpha^{d+o(d)}$ shortest lattice vectors with e.g. the GaussSieve, rather than the $(4/3)^{d/2+o(d)}$ lattice vectors one would normally get, we relax the reduction step in sieving: we reduce a list vector $\boldsymbol{v}$ with another list vector $\boldsymbol{w}$ only if their pairwise angle is less than $\theta = \arcsin(1/\alpha)$, which for vectors $\boldsymbol{v}, \boldsymbol{w}$ of similar norm corresponds to the condition $\|\boldsymbol{v} - \boldsymbol{w}\| < \sqrt{2(1-\cos\theta)} \cdot \|\boldsymbol{v}\| = \sqrt{2 - \frac{2}{\alpha}\sqrt{\alpha^2 - 1}} \cdot \|\boldsymbol{v}\|$. This leads to the modified GaussSieve described in Algorithm 6. Note that for $\alpha = \sqrt{2}$, the reduction criterium becomes $\|\boldsymbol{v} - \boldsymbol{w}\| < \sqrt{2 - \sqrt{2}} \cdot \|\boldsymbol{v}\|$.

Observe that reductions between vectors only make sense if vectors get shorter; if $\boldsymbol{v}$ and $\boldsymbol{w}$ have similar norms, then one cannot reduce $\boldsymbol{v}$ with $\boldsymbol{w}$ if their pairwise angle is larger than $\frac{\pi}{3}$. To generate a list with $\alpha^{d+o(d)}$ short vectors with $\alpha < \sqrt{4/3}$, one can just run the algorithm for $\alpha = \sqrt{4/3}$ (corresponding to the regular GaussSieve), and afterwards discard lattice vectors which are too long. Alternatively, if one is interested in minimizing the memory complexity, one could use tuple lattice sieving approaches discussed in [BLS16, HK16]. Here we restrict our attention to sieving using pairwise reductions, and we leave a complexity analysis based on tuple reductions to future work.

## 4.4 Main result

With the algorithm in place, the optimized complexities for solving CVP now follow from applying nearest neighbor searching with the right parameters. In the preprocessing phase the algorithm first generates a list of size $2^{d/2+o(d)}$ by combining pairs of vectors, and naively this can be done in time $\mathrm{T}_1 = 2^{d+o(d)}$ and space $\mathrm{S}_1 = 2^{d/2+o(d)}$. The query phase corresponds to a polynomial number of reductions of the target with the list, which naively can be done with query time and space complexities $\mathrm{T}_2 = \mathrm{S}_2 = 2^{d/2+o(d)}$.[1] Using nearest neighbor searching (Lemma 1), the query time complexity $\mathrm{T}_2$ as well as the preprocessing time complexity $\mathrm{T}_1$ can be further reduced, at the cost of slightly larger space requirements $\mathrm{S}_{1,2}$, as explained in the following theorem.

[1] To guarantee a polynomial number of reductions, again one can replace the condition $\|\boldsymbol{t}' - \boldsymbol{v}\| \leq \|\boldsymbol{t}'\|$ by $\|\boldsymbol{t}' - \boldsymbol{v}\| \leq \gamma \cdot \|\boldsymbol{t}'\|$ for $\gamma = 1 - o(1)$ as in the Nguyen-Vidick sieve, as explained in the preliminaries, or use techniques as described in [BD15].

**Theorem 2.** *Let $u \in (\frac{1}{2}\sqrt{2}, \sqrt{2})$. Using approximate Voronoi cells, we can heuristically solve CVPP with high probability (over the randomness of $\boldsymbol{t}$) with preprocessing time and space* $\mathrm{T}_1$ *and* $\mathrm{S}_1$*, and query time and space complexities* $\mathrm{T}_2$ *and* $\mathrm{S}_2$ *as follows:*

$$\mathrm{S}_1 = \mathrm{S}_2 = \mathrm{T}_1 = \left(\frac{1}{u(\sqrt{2}-u)}\right)^{d/2+o(d)}, \qquad \mathrm{T}_2 = \left(\frac{\sqrt{2}+u}{2u}\right)^{d/2+o(d)}. \tag{10}$$

*Proof.* These complexities follow from Lemma 1 with $\theta = \frac{\pi}{4}$, noting that the first phase can be performed in time and space $\mathrm{T}_1 = \mathrm{S}_1 = n^{1+\rho_\mathrm{u}}$ (plus the initial costs of the GaussSieve, which are smaller), and the second phase in time $\mathrm{T}_2 = n^{\rho_\mathrm{q}}$ and space $\mathrm{S}_2 = n^{1+\rho_\mathrm{u}}$. Subexponential terms, such as the number of iterations, are omitted here.

To illustrate the time and space complexities of Theorem 2, we highlight three special cases $u$ as follows. The full tradeoff curve for $u \in (\frac{1}{2}\sqrt{2}, \sqrt{2})$ is depicted in Figure 1.

- Setting $u = \frac{1}{2}\sqrt{2}$, we obtain $\mathrm{S}_{1,2} = \mathrm{T}_1 = 2^{d/2+o(d)}$ and $\mathrm{T}_2 = \left(\frac{3}{2}\right)^{d/2+o(d)} \approx 2^{0.2925d+o(d)}$.
- Setting $u = 1$, we obtain $\mathrm{S}_{1,2} = \mathrm{T}_1 \approx 2^{0.6358d+o(d)}$ and $\mathrm{T}_2 \approx 2^{0.1358d+o(d)}$.
- Setting $u = \frac{1}{2}(\sqrt{2}+1)$, we get $\mathrm{S}_{1,2} = \mathrm{T}_1 = 2^{d+o(d)}$ and $\mathrm{T}_2 \approx 2^{0.0594d+o(d)}$.

The first result shows that the query complexity is never worse than for solving CVP directly; only the space and preprocessing complexities are potentially worse. Note also that setting $u = \frac{1}{2}\sqrt{2}$ ensures that the data structure for the nearest neighbor searching does not use more data than storing the list in memory, corresponding to the quasi-linear space regime. The second and third results show that CVPP can be solved in significantly less time using more preprocessing, even with preprocessing and space complexities bounded by $2^{d+o(d)}$.

**Minimizing the query complexity.** As $u \to \sqrt{2}$, the query complexity keeps decreasing while the memory and preprocessing costs increase. For arbitrary $\varepsilon > 0$, we can set $u = u_\varepsilon \approx \sqrt{2}$ as a function of $\varepsilon$, resulting in asymptotic complexities $\mathrm{T}_1, \mathrm{S}_{1,2} = (1/\varepsilon)^{O(d)}$ and $\mathrm{T}_2 = 2^{\varepsilon d+o(d)}$. This shows that it is possible to obtain a slightly subexponential query complexity, at the cost of superexponential space and preprocessing costs, by taking $\varepsilon = o(1)$ as a function of $d$.

**Corollary 1 (Subexponential query complexities for CVPP).** *For arbitrary $\varepsilon > 0$, we can heuristically solve CVPP with preprocessing time and space complexities* $\mathrm{T}_1, \mathrm{S}_{1,2} = (1/\varepsilon)^{O(d)}$*, in time* $\mathrm{T}_2 = 2^{\varepsilon d+o(d)}$*. In particular, we can solve CVPP in* $\mathrm{T}_2 = 2^{o(d)}$ *time, using* $\mathrm{T}_1, \mathrm{S}_{1,2} = 2^{\omega(d)}$ *space and preprocessing.*

Being able to solve CVPP in subexponential time with superexponential preprocessing and memory is neither trivial nor quite surprising. A naive approach to the problem, with a large amount of memory, could for instance be to index the entire fundamental domain of $\mathcal{L}$ in a hash table. One could partition this domain into small regions, solve CVP for the centers of each of these regions, and store all the solutions in memory. Then, given a query, one looks up which region $\boldsymbol{t}$ lies in, and returns the answer corresponding to that vector. With a sufficiently fine-grained partitioning of the fundamental domain, the answers given by the look-ups are accurate, and such an algorithm likely also runs in subexponential time, given enough space and preprocessing time.

Although it may not be surprising that it is possible to solve CVPP in subexponential time with (super)exponential space, it is not clear what the complexities of other methods would be. Our method presents a clear tradeoff between the complexities, where the constants in the preprocessing exponent are quite small; for instance, we can solve CVPP in time less than $2^{0.06d+o(d)}$ with less than $2^{d+o(d)}$ memory, which is the same amount of memory/preprocessing of the best provable SVP and CVP algorithms [ADRS15, ADS15]. Indexing the fundamental domain (or any other CVPP method with a subexponential query complexity) may well require much more memory than this.

## 5 Solving CVPP – Randomized slicing

As described in the previous section, to solve exact CVPP with high probability using a single application of the slicing algorithm, we need a preprocessed list size/memory complexity of at least $2^{d/2+o(d)}$, where to obtain better query complexities we need to further increase the memory for the nearest neighbor data structure. As the memory complexity is often the biggest bottleneck in exponential-space algorithms for SVP/CVP, being able to use a smaller list size would significantly increase the practicability of the algorithm. However, to guarantee a constant success probability of finding the nearest lattice vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ in $L$ with the slicer, we inevitably need $L$ to contain $2^{d/2+o(d)}$ vectors. Otherwise, there is only a small probability that the closest vector to the reduced vector $\boldsymbol{t}'$ is $\mathbf{0}$.

### 5.1 Randomized slicing

To improve upon both the time and space complexities of Theorem 1, we use two ideas. First, similar to e.g. pruning and rerandomizing the basis in fast heuristic lattice enumeration methods [GNR10], we allow for arbitrary success probabilities $p$ of correctly finding the closest vector to the target with the slicing algorithm, and we measure the query time complexity by the expected time complexity $\mathbb{E}[\mathrm{T}_2/p]$. This however does not solve the issue that the probability $p$ is computed only over the randomness of the targets – it might therefore happen that for certain targets, the algorithm never succeeds, even after several attempts. The second idea is a heuristic way to overcome this problem: given a target $\boldsymbol{t}$, we *rerandomize* the target by shifting it with a random lattice vector, before proceeding with the reductions. Here we hope that these rerandomizations of the same target indeed produce fresh, independent results, so that repeating the algorithm $O(1/p)$ times indeed leads to a large/constant success probability.

Let us start by analyzing the success probability $p$ of the slicer, as a function of $|L| = \alpha^{d+o(d)}$, where the randomness is over the targets $\boldsymbol{t}$ sampled uniformly at random from a very large region. By the Gaussian heuristic we again assume that $\boldsymbol{s}$ lies at distance $(1+o(1))\lambda_1(\mathcal{L})$ from $\boldsymbol{t}'$ as in Figure 3a, but instead of assuming that with high probability $\boldsymbol{s}-\boldsymbol{t}'$ is approximately orthogonal to $\boldsymbol{t}'$, we make the following additional assumption. Here $\mathcal{B}(\boldsymbol{x}, r) := \{\boldsymbol{y} \in \mathbb{R}^d : \|\boldsymbol{y}-\boldsymbol{x}\| \leq r\}$ denotes a ball of radius $r$ around $\boldsymbol{x}$.

**Heuristic assumption 4.** *After reducing $\boldsymbol{t}$ to $\boldsymbol{t}' \in \boldsymbol{t}+\mathcal{L}$, the closest lattice vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ is uniformly distributed on the sphere of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$. Furthermore, reducing $\boldsymbol{t}+\boldsymbol{p}$ to $\boldsymbol{t}''$ for sufficiently large $\boldsymbol{p} \in \mathcal{L}$, the resulting vectors $\boldsymbol{t}'$ and $\boldsymbol{t}''$ are statistically independent and with high probability follow the same distribution over $\mathcal{R} = (\boldsymbol{t}+\mathcal{L}) \cap \mathcal{B}(\mathbf{0}, \beta \cdot \lambda_1(\mathcal{L}))$.*

The first part states that $\boldsymbol{s}-\boldsymbol{t}'$ is usually orthogonal to $\boldsymbol{t}'$, and provides us with asymptotics on the distribution of $\langle \boldsymbol{s}-\boldsymbol{t}', \boldsymbol{t}' \rangle$. It can also be interpreted as giving us the probability that the closest lattice vector to $\boldsymbol{t}'$ is contained in the ball of radius $\alpha \cdot \lambda_1(\mathcal{L})$ around the origin, in which case $\boldsymbol{s}$ would be contained in $L$.

**Algorithm 7** The randomized query phase for solving CVPP

```
Require: A list $L \subset \mathcal{L}$ and a target $\boldsymbol{t} \in \mathbb{R}^d$
Ensure: The output vector $\boldsymbol{s}$ is a close lattice vector to $\boldsymbol{t}$
1: repeat
2:     Sample a random $\boldsymbol{p} \in \mathcal{L}$ (e.g. from a discrete Gaussian with large variance)
3:     Initialize $\boldsymbol{t}' \leftarrow \boldsymbol{t} + \boldsymbol{p}$
4:     for each $\boldsymbol{w} \in L$ do
5:         if $\|\boldsymbol{t}' - \boldsymbol{w}\| \leq \|\boldsymbol{t}'\|$ then
6:             Replace $\boldsymbol{t}' \leftarrow \boldsymbol{t}' - \boldsymbol{w}$ and restart the for-loop
7:     $\boldsymbol{s} \leftarrow \boldsymbol{t} - \boldsymbol{t}'$
8: until $\boldsymbol{s}$ is a closest lattice vector to $\boldsymbol{t}$
9: return $\boldsymbol{s}$
```

The second statement in Heuristic 4. is what we will use for rerandomizations: if we fail, and the reduced vector $\boldsymbol{t}'$ is not the shortest vector in $\boldsymbol{t} + \mathcal{L}$, then we can start over by adding a random, sufficiently large perturbation vector $\boldsymbol{p} \in \mathcal{L}$ to $\boldsymbol{t}$ and running the same reduction algorithm from Algorithm 5 again with this shifted target vector, pretending that this gives us a fresh, independent result. This leads to Algorithm 7. Note that the second assumption is far from trivial, but as our experiments will show, if $\boldsymbol{p}$ is large, then rerandomizing indeed leads to a roughly linear increase in the success probability.

To compute the asymptotics of the probability $p = p_\alpha$ that the vector $\boldsymbol{s}$ has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$, given that the initial list had size $\alpha^{d+o(d)}$ and the reduced vector $\boldsymbol{t}'$ has norm $\beta_\alpha \cdot \lambda_1(\mathcal{L})$ (where $\beta_\alpha$ follows from Equation (6)), we make use of the following lemma, restated from [BLS16, Lemma 2.4]. Here $|\mathcal{B}|$ denotes the volume of the ball $\mathcal{B}$.

**Lemma 8.** *[BLS16, Lemma 2.4] Two balls $\mathcal{B}(\boldsymbol{v}_1, r_1)$ and $\mathcal{B}(\boldsymbol{v}_2, r_2)$ at distance $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\| = D$ with centers $\boldsymbol{v}_1, \boldsymbol{v}_2$ and radii $r_1, r_2$, such that $\sqrt{|r_1^2 - r_2^2|} < D < r_1 + r_2$, satisfy*

$$\frac{|\mathcal{B}(\boldsymbol{v}_1, r_1) \cap \mathcal{B}(\boldsymbol{v}_2, r_2)|}{|\mathcal{B}(\boldsymbol{0}, 1)|} = \left( \frac{-D^4 + 2D^2\left(r_1^2 + r_2^2\right) - \left(r_1^2 - r_2^2\right)^2}{4D^2} \right)^{d/2 + o(d)}. \tag{11}$$

Lemma 8 shows that if $\boldsymbol{x} \in \mathcal{B}(\boldsymbol{v}_1, r_1)$ is drawn uniformly at random, then the probability that it lies at distance at most $r_2$ from $\boldsymbol{v}_2$ is given by the RHS of (11), multiplied by a factor $|\mathcal{B}(\boldsymbol{0}, 1)|/|\mathcal{B}(\boldsymbol{0}, r_1)| = r_1^{-d+o(d)}$.

To apply this lemma, recall that we assumed that $\boldsymbol{s}$ is essentially uniformly distributed in the ball of radius $\lambda_1(\mathcal{L})$ around $\boldsymbol{t}'$, while $\boldsymbol{t}'$ lies at distance at most $\beta \cdot \lambda_1(\mathcal{L})$ from the origin after reductions, where $\beta$ is a function of $\alpha$ as in Lemma 7. To obtain the probability that $\boldsymbol{s}$ has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$, so that $\boldsymbol{s} \in L$, we therefore apply the lemma with $(\boldsymbol{v}_1, \boldsymbol{v}_2, r_1, r_2, D) = (\boldsymbol{t}', \boldsymbol{0}, \lambda_1(\mathcal{L}), \alpha\lambda_1(\mathcal{L}), \beta\lambda_1(\mathcal{L}))$ leading to

$$p_\alpha = \frac{|\mathcal{B}(\boldsymbol{t}', \lambda_1(\mathcal{L})) \cap \mathcal{B}(\boldsymbol{0}, \alpha\lambda_1(\mathcal{L}))|}{|\mathcal{B}(\boldsymbol{t}', \lambda_1(\mathcal{L}))|} = \left( \frac{-\beta^4 + 2\beta^2\left(1 + \alpha^2\right) - \left(\alpha^2 - 1\right)^2}{4\beta^2} \right)^{d/2 + o(d)}. \tag{12}$$

Using the relation $\alpha^2 = 2\beta(\beta - \sqrt{\beta^2 - 1})$, or equivalently $\beta^2 = \alpha^4/(4\alpha^2 - 4)$, we can express the probability $p_\alpha$ in terms of $\alpha$:

$$p_\alpha = \left( \frac{-9\alpha^8 + 64\alpha^6 - 104\alpha^4 + 64\alpha^2 - 16}{16\alpha^4\left(\alpha^2 - 1\right)} \right)^{d/2 + o(d)}. \tag{13}$$

Observe that the denominator is non-zero for arbitrary $\alpha \in (1, \sqrt{2})$, while the numerator has one root in this interval, at $\alpha \approx 1.03396$. For this value of $\alpha$, we have $\beta = \alpha + 1$ and so the two balls around $\boldsymbol{t}'$ and $\boldsymbol{0}$ of radii $\{1, \alpha\} \cdot \lambda_1(\mathcal{L})$ are disjoint, resulting in $p = 0$. For $\alpha = \sqrt{2}$ the expression between brackets evaluates to 1 as expected, while for $\alpha = \beta = \sqrt{4/3}$ (using the same list size as in sieving for SVP) we obtain $p_{\sqrt{4/3}} = (13/16)^{d/2+o(d)}$. So if we used a standard GaussSieve as preprocessing for CVPP, we would expect the success probability of a single reduction to be $(13/16)^{d/2+o(d)} \approx 2^{-0.150d+o(d)}$. We illustrate this with a simple corollary as follows:

**Proposition 1.** *Using the output list from a standard GaussSieve with spherical LSF as the preprocessed list $L$, the randomized slicer succeeds with probability $p = (13/16)^{d/2+o(d)}$, leading to preprocessing and query time and space complexities for solving CVPP of*

$$\mathrm{S}_{1,2} = \mathrm{T}_1 = (3/2)^{d/2+o(d)}, \qquad \mathrm{T}_2 = (18/13)^{d/2+o(d)}. \tag{14}$$

These complexities follow directly from [BDGL16], where we note that the query complexity for a single query is $(9/8)^{d/2+o(d)}$, which together with the rerandomization costs of $1/p = (16/13)^{d/2+o(d)}$ lead to $\mathrm{T}_2 = (9/8 \cdot 16/13)^{d/2+o(d)}$. In Figure 1, this would correspond to the point $(2^{0.2925d}, 2^{0.2347d})$, which is slightly above the curve with the best complexities, obtained by optimizing both the nearest neighbor search parameters and the preprocessed list size.

### 5.2 Main result

Let us now try to obtain the optimized complexities for randomized slicing. Compared to the previous section, the costs of the algorithm are mostly the same in terms of $\alpha$, except that the (expected) time complexity $\mathrm{T}_2$ for the query phase is multiplied by a factor $1/p$, to account for the expected number of trials necessary to find a closest vector. On the positive side, this means that we do not need to fix $\alpha = \sqrt{2}$ in advance. In its most general form, including the nearest neighbor parameter $u$ from Lemma 1, we obtain the following result. Note that $\mathrm{S}_1$ and $\mathrm{T}_1$ are lower bounded by the costs for solving SVP, which based on the current best complexities for (pairwise) sieving are $(4/3)^{d/2+o(d)}$ and $(3/2)^{d/2+o(d)}$ respectively. Using tuple sieving [BLS16,HK16], it is possible to eliminate this lower bound on $\mathrm{S}_1$, at the cost of worse preprocessing time complexities.

**Theorem 3.** *Let $\alpha \in (1.03996, \sqrt{2})$ and $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$, and let $p_\alpha$ be as defined in* (13)*. With randomized slicing, we can heuristically solve CVPP with preprocessing time and space $\mathrm{T}_1$ and $\mathrm{S}_1$, and query time and space $\mathrm{T}_2$ and $\mathrm{S}_2$, where*

$$\mathrm{S}_1 = \max\left\{\mathrm{S}_2, \left(\frac{4}{3}\right)^{d/2+o(d)}\right\}, \qquad \mathrm{T}_1 = \max\left\{\mathrm{S}_2, \left(\frac{3}{2}\right)^{d/2+o(d)}\right\}, \tag{15}$$

$$\mathrm{S}_2 = \left(\frac{\alpha}{\alpha - (\alpha^2-1)(\alpha u^2 - 2u\sqrt{\alpha^2-1} + \alpha)}\right)^{d/2+o(d)}, \tag{16}$$

$$\mathrm{T}_2 = \left(\frac{\alpha + u\sqrt{\alpha^2-1}}{p_\alpha\left(-\alpha^3 + \alpha^2 u\sqrt{\alpha^2-1} + 2\alpha\right)}\right)^{d/2+o(d)}. \tag{17}$$

*Proof.* These complexities follow from applying Lemma 1 with $\theta = \arcsin(1/\alpha)$, and again observing that the preprocessing can be performed in time $\mathrm{T}_1 = n^{1+\rho_\mathrm{u}} + (3/2)^{d/2}$ and space $\mathrm{S}_1 = n^{1+\rho_\mathrm{u}} + (4/3)^{d/2}$, where the additive terms come from the minimum costs for solving SVP. For the query phase we can potentially discard part of the preprocessed space (if $\alpha, u$ are small), and the query time is $\mathrm{T}_2 = n^{\rho_\mathrm{q}}$.

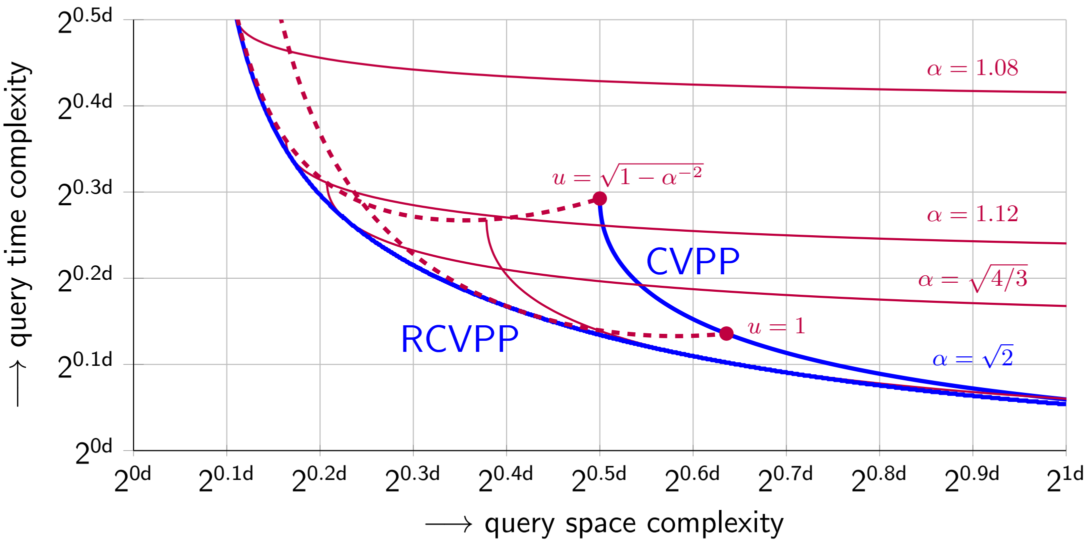


**Fig. 4.** Complexities for randomized slicing. Different curves correspond to different values $\alpha$ and different success probabilities $p_\alpha$. The right blue curve corresponds to $\alpha = \sqrt{2}$ and $p_\alpha \approx 1$, i.e. not using randomized slicing as in Section 4, and purple curves inbetween correspond to smaller values $\alpha$ with smaller values $p_\alpha$. Dashed purple curves correspond to fixing the nearest neighbor parameter $u$ and varying $\alpha$. No single curve lies below all others, and the minimum over all curves is depicted by the bottom blue curve.

For $\alpha = \sqrt{2}$, Theorem 3 leads to the exact same complexities as without rerandomizations, while for instance for $\alpha = \sqrt{4/3}$ we can vary the parameter $u$ to obtain a different tradeoff:

– Setting $u = \frac{1}{2}$ leads to $\mathrm{S}_2 \approx 2^{0.2075d+o(d)}$ and $\mathrm{T}_2 = (20/13)^{d/2+o(d)} \approx 2^{0.3107d+o(d)}$.
– Setting $u = 1$ leads to $\mathrm{S}_2 \approx 2^{0.2925d+o(d)}$ and $\mathrm{T}_2 = (18/13)^{d/2+o(d)} \approx 2^{0.2347d+o(d)}$.
– Letting $u \to 2$, the preprocessing and space complexities become $2^{\omega(d)}$, while the query time complexity becomes $2^{o(d)}$.

The time-space curve $\mathcal{C}_\alpha$ corresponding to $\alpha = \sqrt{4/3}$, as well as a few other values $\alpha$, is shown in Figure 4. By taking the minimum over all these curves $\{\mathcal{C}_\alpha\}_{\alpha \in (1.03396, \sqrt{2})}$, where curves are defined by varying $u \in (\sqrt{1 - 1/\alpha^2}, \sqrt{1 + 1/(\alpha^2 - 1)})$, we obtain the thick blue curve in Figure 4, which is also depicted in Figure 1. There seems to be no simple expression for this curve; for a particular choice of the space complexity, the best query time complexity $\mathrm{T}_2$ can be found by considering all different $\alpha$, and for each $\alpha$ computing the value $u$ such that the space complexity is as desired, and taking the minimum over all these values. Note that due to the condition $\alpha > 1.03396$ (which follows from $p_\alpha > 0$), the curve terminates on the left side at a minimum space complexity of $1.03396^{d+o(d)} \approx 2^{0.0482d+o(d)}$; with this method we cannot obtain a space complexity $\mathrm{S}_2 = 2^{o(d)}$ for exact CVPP.

## 6 Solving CVPP – Experiments

To verify the heuristic assumptions, as well as to provide a preliminary assessment of the practicality of the proposed CVPP methods, we tested the approximate Voronoi cell algorithm with randomized slicing for finding closest vectors on the 50-dimensional lattice of the SVP challenge [SG15] with seed 0. The experiments were conducted by (1) generating a large set of short lattice vectors (by using the algorithm described in Algorithm 6); (2) indexing these in a nearest neighbor data structure; and (3) running the randomized slicer to find closest vectors for random targets.

### 6.1 Nearest neighbor data structure

For nearest neighbor indexing, in our experiments we chose to use the HashSieve data structure as described in [Laa15a] rather than the LDSieve of Lemma 1 for several reasons. First, for the HashSieve, the only parameters that need to be chosen are $k$ (the number of hyperplanes) and $t$ (the number of hash tables). As described in [Laa15a, MLB15], the asymptically optimal parameter choices $k = 0.2206d$ and $t = 2^{0.1290d}$ seem quite accurate as near-optimal parameters for small/moderate dimensions as well, which essentially means fewer parameters need to be chosen for our experiments, and there will be a smaller variance in the results. This in contrast to the asymptotically superior LDSieve of [BDGL16, Laa15b], where several parameters must be chosen with less clear optimal choices in moderate dimensions. Secondly, for solving SVP in dimension 50, a proof-of-concept HashSieve outperforms the LDSieve [BDGL16, Figure 3] by a factor more than 2. Using the LDSieve may ultimately lead to better timings in higher dimensions, with optimized code and an accurate analysis of how to choose parameters, but that lies beyond the scope of this section. The main target here is to verify/disprove the heuristic assumptions, and get an idea of the speedup compared to heuristic sieving for SVP/CVP.

For the experiments in dimension 50, we used the HashSieve with the same parameter choice of $k = 11$ hyperplanes and $t = 87$ hash tables as in [Laa15a]. We varied both the number of lattice vectors indexed in the data structure (out of all the vectors obtained from the preprocessing stage), and the number of rerandomizations before calling the search for a closest vector to this target a failure. We naturally sorted the preprocessed vectors by norm, so that using fewer vectors means that only the shortest of the lattice vectors found during the preprocessing phase are used. For randomizations, we sampled a random lattice vector from a discrete Gaussian over the lattice (similar to how lattice vectors are sampled for the sieve), added it to the target vector, and reduced this new target instead. All experiments were further performed on a Medion Erazer P6661 laptop with an Intel Core i7-6500U processor (2.50GHz), with more than enough RAM for these experiments. Experiments typically consumed about 25% of the total CPU power, i.e. 50% of one of the two cores.

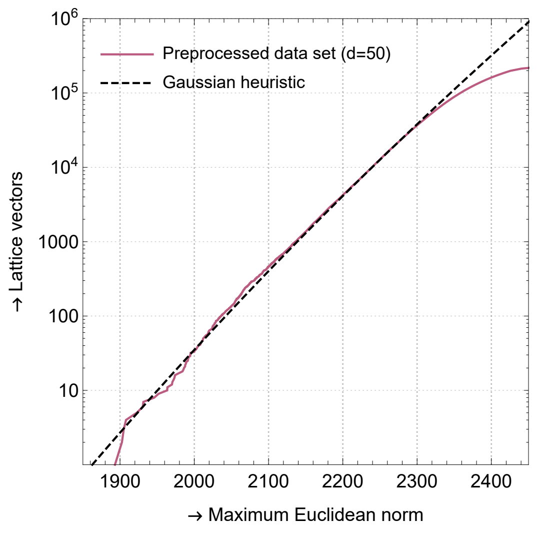
### 6.2 Validating the heuristic assumptions

First, let us describe the preprocessed data set in more detail. Our complete preprocessed data set consists of about 250 000 lattice vectors of norm less than 3000, with the majority of them having norm less than 2500. Figure 5a shows the number of vectors in the data set below a certain norm, and compares it to the prediction based on the Gaussian heuristic. If the Gaussian heuristic is accurate, then the data set indeed contains almost all of the lattice vectors of norm less than 2300.

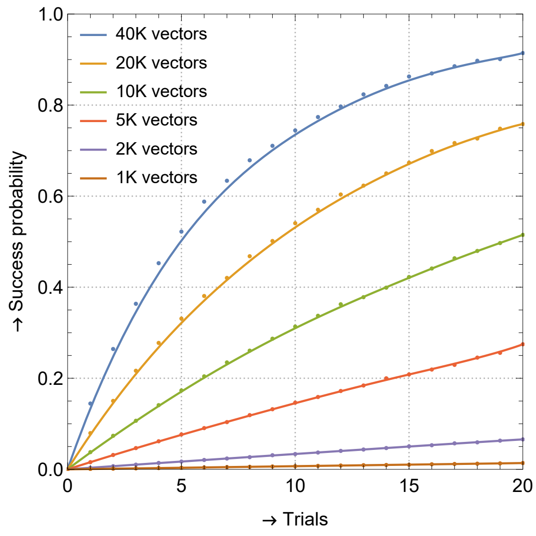
A critical assumption we made in Section 5 is that rerandomizations (reducing a target vector again by first adding a random lattice vector to it) lead to independent successes and roughly a linear increase in the success probability as more trials are performed (Heuristic 4.). Figure 5b plots the total success probability against the number of rerandomizations (trials), for various sizes of the list indexed in the data structure. Figure 5b seems to indicate that indeed, if the success probability is small, then this probability roughly increases linearly with the number of trials.

### 6.3 Experimental results

As stated before, for the CVPP experiments we fixed the HashSieve parameters as $k = 11$ and $t = 87$. Focusing on the query costs, what remains is optimizing the size of the list $L$



(a) Norms of vectors in the data set



(b) Rerandomization heuristics

**Fig. 5.** Verifying the heuristic assumptions. Figure 5a compares the norms of the vectors in the data set to the expected norms of the shortest vectors in the lattice, based on the Gaussian heuristic. Figure 5b depicts how rerandomizations affect the success probability. Different curves in Figure 5b correspond to a different number of preprocessed vectors being used for reductions.

to use for the reductions, and assessing the precise practical impact of rerandomizations on the success probability and the time complexity. Note that by Heuristic 4. the number of rerandomizations should not severely impact the normalized time complexity (the time complexity divided by the success probability), although in practice it might.

Figures 6a–6d show the results of these experiments, where we measured the average time complexity per target vector (Figure 6a), the average success probability for each instance (Figure 6b), the normalized time complexity per instance (Figure 6c), and the preprocessed space complexity for these experiments (Figure 6d). Different curves in Figures 6a–6c correspond to different numbers of rerandomizations, and although this affects the success probability and the time complexity, in Figure 6c we once again see a confirmation that the normalized time complexity is essentially independent of the number of trials. Figure 6c further shows that the normalized time complexity seems to be smallest when the list size is between 10 000 and 15 000, in which case the query time complexity $\mathrm{T}_2$ for solving one CVPP instance is approximately 0.002 seconds. The memory complexity $\mathrm{S}_2$ when using this number of list vectors is approximately 10MB.

## 6.4 Comparison with sieving for solving SVP

To put these time complexities into perspective, let us compare the (normalized) time complexities for CVPP with the complexities of sieving for solving SVP, which by Section 3 we estimate are comparable to the costs for sieving for CVP. First, we observe that with the same amount of optimization, the HashSieve algorithm solves SVP in approximately 4 seconds on the same machine. This means that in dimension 50, the expected time complexity for CVPP with the HashSieve is approximately 2000 times smaller than the time for solving SVP. To explain this gap, observe that the list size for solving SVP is approximately 4000, and so roughly speaking the HashSieve algorithm needs to perform 4000 reductions of newly sampled vectors with a list of maximum size 4000. For solving CVPP, we only need to reduce 1 target vector with the list, but the list is now between

10 000 and 15 000 vectors long. This means that we save a factor 4000 on the number of searches through the list, but the searches are slightly more expensive as the list size is longer, leading to a speed-up of a factor slightly less than 4000.

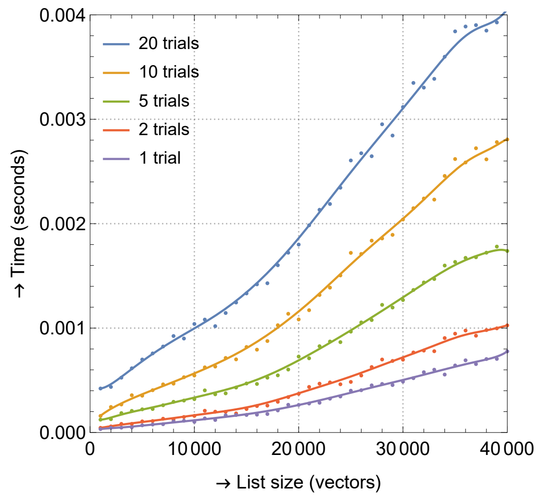
To make these estimates more precise, note that the HashSieve for solving SVP [Laa15a] reported time complexities in dimension $d$ of approximately $2^{0.45d-19}$ seconds, which corresponds to approximately 11.3 seconds in dimension 50, i.e. a factor 3 slower than our implementation. As explained above, this is based on doing $n = 2^{0.21d+o(d)}$ reductions. If for simplicity we assume that doing only one of these searches in a slightly larger list takes a factor $2^{0.21d}$ less time, and we take into account that for SVP the time complexity is now a factor 3 less than in [Laa15a], then we obtain an estimated complexity in dimension $d$ of $2^{0.24d-19}/3$, which for $d = 50$ corresponds to approximately 0.0026 seconds with our implementation, closely matching the observed time complexity. A rough extrapolation would then lead to a time complexity in dimension 100 of approximately 11 seconds.

## 6.5 Comparison with enumeration for SVP/CVP

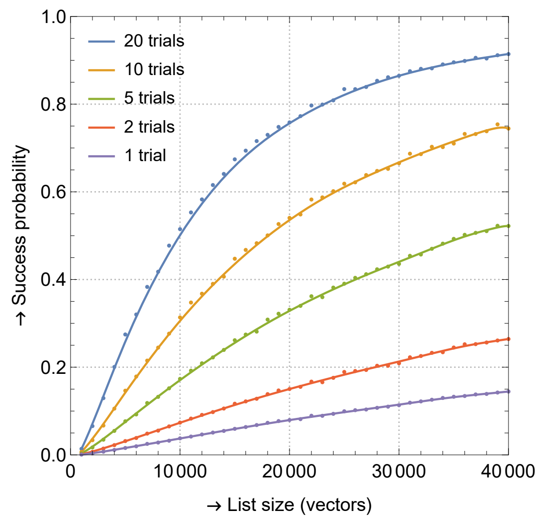
In low dimensions, the fastest algorithms for solving SVP and CVP are based on enumeration. To compare the preprocessing approach to sieving with enumeration-based methods, we list several of the reported complexities for SVP and CVP with enumeration from the literature below, in chronological order. The list complexities are all for lattices in dimension 50.

- Agrell–Eriksson–Vardy–Zeger [AEVZ02, Figure 2] give costs for CVP which, when extrapolated to dimension 50, would correspond to between 10 and 20 seconds.
- Nguyen–Vidick [NV08, Figure 4] report costs of Schnorr-Euchner enumeration with BKZ-20 preprocessing between 2 and 3 minutes.
- Hermans–Schneider–Buchmann–Vercauteren–Preneel [HSB+10, Table 2] give an estimate of between 5 and 7 seconds for enumeration with BKZ-20 preprocessing.
- Gama–Nguyen–Regev [GNR10, Table 1] give four data points for the number of nodes processed during enumeration for three different versions of enumeration, which when fitted to the model $2^{ad^2+bd}$, give $2^{0.00416d^2+0.255d}$ (full enumeration), $2^{0.00379d^2+0.115d}$ (Schnorr-Hörner pruning), and $2^{0.00387d^2+0.059d}$ (linear pruning). Taking into account their estimated rate of $10^7$ nodes processed per second, in dimension $d = 50$ this leads to a sequential time complexity of approximately 0.94 seconds (full enumeration), 0.0038 seconds (Schnorr-Hörner pruning) and 0.00062 seconds (linear pruning). For extreme pruning, only two data points are provided, which is not enough to extrapolate to dimension 50.
- Dagdelen–Schneider [ODS10, Table 1] report timings between 6 and 8 minutes for running their sequential implementation and for running fplll's enumeration with LLL preprocessing.
- Micciancio–Walter [MW15, Figure 7] give an experimental time complexity of Fincke-Pohst enumeration of approximately 30 seconds.
- Correia–Mariano–Proenca–Bischof–Agrell [CMP+16, Figure 6b] state a time complexity of enumeration for solving CVP in dimension 50 of approximately 10 seconds with BKZ-20 preprocessing.

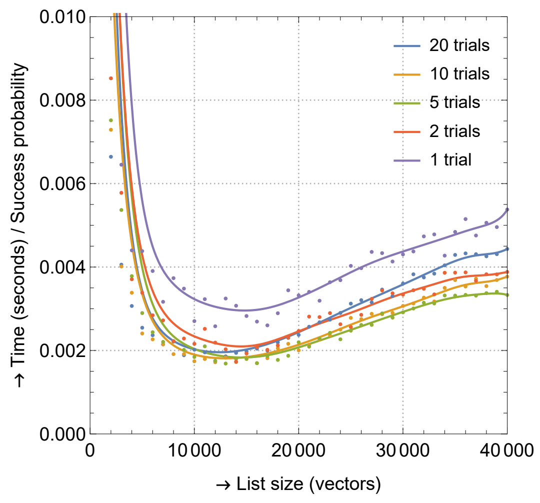
Calling `shortest_vector()` within fplll 4.0 on the machine used for the experiments in this section (on a BKZ-20 reduced basis), the algorithm returns a shortest vector in approximately 30 seconds. In the most recent release of fplll (version 5) [dt16], this currently takes approximately 5 seconds.



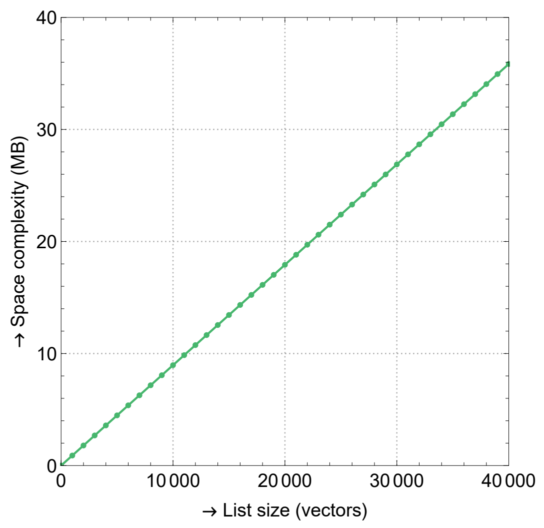
(a) Time complexity per CVP instance



(b) Success probability after rerandomizations



(c) Expected time complexity per CVP instance



(d) Space complexity

**Fig. 6.** Experimental results for solving CVPP with randomized sieving. Figure 6a displays the average time complexity per instance. Figure 6b shows the average success probability after rerandomizations. Figure 6c displays expected time costs, taking into account the success probability. Figure 6d displays the space complexity of the data set and the indexed data structure. Each data point corresponds to 10 000 random target vectors for this choice of parameters.

To summarize, all reported experimental time complexities for enumeration in dimension 50 are significantly worse than our 0.002 seconds per target. On the other hand, enumeration with linear pruning (and likely also extreme pruning) is still expected to solve more SVP (CVP) instances per second than our proof-of-concept CVPP algorithm based on the HashSieve, in dimension 50. Based on the very rough estimates of $2^{0.00387d^2+0.059d}/10^7$ seconds for enumeration with linear pruning and $2^{0.24d-19}/3$ for a HashSieve-based CVPP solver, the crossover point however may already lie around dimension 60. We therefore expect that this algorithm will already clearly outperform enumeration with linear pruning in terms of the CVPP complexity in dimensions of interest in practice (e.g. $d = 100$).

Note that for state-of-the-art enumeration methods based on extreme pruning, giving a fair comparison is not so easy, as little explicit experimental data using extreme pruning is known, and most data points are in much higher dimensions; extrapolating backwards to dimension 50 or 60 might not give reliable estimates.

## 7 Solving BDD(P)$_\delta$

In this section and the next, we take a look at specific instances of CVP which are easier to solve than the general problem, such as when the target $\boldsymbol{t}$ lies unusually close to the lattice. This problem naturally appears in lattice-based cryptography, when a private key consists of a *good basis* of a lattice with short basis vectors, and the public key is a *bad basis* of the same lattice. An encryption of a message could then consist of the message being mapped to a lattice point $\boldsymbol{s} \in \mathcal{L}$, and a small error vector $\boldsymbol{e}$ being added to $\boldsymbol{s}$ ($\boldsymbol{t} = \boldsymbol{s} + \boldsymbol{e}$) to hide $\boldsymbol{s}$. If the noise $\boldsymbol{e}$ is small enough, then with a good basis one can decode $\boldsymbol{t}$ to the closest lattice vector $\boldsymbol{s}$, while someone with the bad basis cannot decode correctly. As decoding for arbitrary $\boldsymbol{t}$ (solving CVP) is known to be hard even with knowledge of a good basis [Mic01, FM02, Reg04, AKKV05], $\boldsymbol{e}$ needs to be very short for decryptions to work, and $\boldsymbol{t}$ must lie very close to the lattice. So instead of assuming that target vectors $\boldsymbol{t}$ are sampled at random, suppose that $\boldsymbol{t}$ lies at distance at most $\delta \cdot \lambda_1(\mathcal{L})$ from $\mathcal{L}$, for $\delta \in (0, 1)$.

**Without preprocessing.** For the CVP algorithm from Section 3, recall that the list size $(4/3)^{d/2+o(d)}$ is the minimum initial list size one can hope to use to obtain a list of short lattice vectors with pairwise sieving (again, see the recent [BLS16, HK16] for promising directions in using even less space and more time for heuristic lattice sieving), and it remains an open problem to study if the complexities for sieving (without preprocessing) can be improved beyond the costs for SVP or CVP. As CVP is not harder than BDD we may obtain the same complexities for BDD as for CVP, but future work might focus on solving BDD faster with sieving, without preprocessing.

**With preprocessing.** For the approximate Voronoi cell approach for CVPP, it may well be possible to reduce the complexities for $\delta < 1$ by modifying the analysis. Let us again start by assuming that the preprocessed list $L$ contains almost all $\alpha^{d+o(d)}$ lattice vectors of norm at most $\alpha \cdot \lambda_1(\mathcal{L})$. The choice of $\alpha$ implies a maximum norm $\beta \cdot \lambda_1(\mathcal{L})$ of the reduced vector $\boldsymbol{t}'$, as described in Lemma 7. We assume the nearest lattice vector $\boldsymbol{s}$ to $\boldsymbol{t}'$ lies within radius $\delta \cdot \lambda_1(\mathcal{L})$ of $\boldsymbol{t}'$, and using Lemma 8 we then find the (heuristic) probability of finding the closest lattice vector among the list to be:

$$p_{\alpha,\delta} = \frac{|\mathcal{B}(\boldsymbol{t}', \delta) \cap \mathcal{B}(\boldsymbol{0}, \alpha)|}{|\mathcal{B}(\boldsymbol{t}', \delta)|} = \left(\frac{-\beta^4 + 2\beta^2\left(\delta^2 + \alpha^2\right) - \left(\alpha^2 - \delta^2\right)^2}{4\beta^2\delta^2}\right)^{d/2+o(d)}. \tag{18}$$

See also Figure 3b, where the shaded area can be considered asymptotically negligible to obtain success probability $p \approx 1$, or we can assume that $\boldsymbol{s}$ is sampled at random from the blue ball to obtain smaller success probabilities depending on the fraction of the blue ball that is covered by the black ball.

Without rerandomizations, to achieve $p \approx 1$, we need $\sqrt{\beta_\alpha^2 + \delta^2} \leq \alpha$ to expect the nearest lattice vector to $\boldsymbol{t}'$ to be contained in $L$, so that ultimately $\boldsymbol{0}$ is nearest to $\boldsymbol{t}'$ after reductions. Substituting $\alpha^4 - 4\beta^2\alpha^2 + 4\beta^2 = 0$ and $\beta^2 + \delta^2 \leq \alpha^2$, and solving for $\alpha > 1$, without rerandomizing this leads to the condition $\alpha^2 \geq \frac{2}{3}(1 + \delta^2) + \frac{2}{3}\sqrt{(1 + \delta^2)^2 - 3\delta^2}$. Taking $\delta = 1$, corresponding to exact CVP, leads to the condition $\alpha \geq \sqrt{2}$ as expected, while in the limiting case of $\delta \to 0$ we obtain the condition $\alpha \geq \sqrt{4/3}$. This matches experimental observations using the GaussSieve, where after finding the shortest vector, newly sampled lattice vectors often cause *collisions* (i.e. being reduced to the $\mathbf{0}$-vector). In other words, Algorithm 5 quite often reduces target vectors $\boldsymbol{t}$ which essentially lie *on* the lattice ($\delta \to 0$) to the $\mathbf{0}$-vector when the list has size $(4/3)^{d/2+o(d)}$. This explains why collisions in the GaussSieve are common when the list size grows to size $(4/3)^{d/2+o(d)}$.

**Randomized slicing.** With rerandomizations, a choice of $\alpha$ implies a norm $\beta$ of the reduced vector $\boldsymbol{t}'$, and a probability $p_{\alpha,\delta}$ that the closest lattice vector is then found with the algorithm. For each $\alpha$ we can further use nearest neighbor searching with varying parameters $u$ as in Lemma 1, and we can vary $\alpha \in (\alpha_0, \alpha_1)$ where $\alpha_0, \alpha_1$ follow from the equations $p_{\alpha_0,\delta} = 0$ and $p_{\alpha_1,\delta} = 1$ respectively. In other words, $\alpha_1$ satisfies $\beta_{\alpha_1}^2 + \delta^2 = \alpha_1^2$, and $\alpha_0$ is a root of the denominator of (18). This leads to the following theorem.

**Theorem 4.** *Let $\alpha \in (\alpha_0, \alpha_1)$ where $\alpha_0, \alpha_1$ are such that $p_{\alpha_0,\delta} = 0$ and $p_{\alpha_1,\delta} = 1$, with $p_{\alpha,\delta}$ as described in* (18). *Let $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. Using approximate Voronoi cells with randomized slicing, we can heuristically solve BDDP$_\delta$ with complexities* $\mathrm{S}_{1,2}, \mathrm{T}_{1,2}$ *described in Theorem 3 with $p_\alpha$ replaced by $p_{\alpha,\delta}$.*

For arbitrary $\delta$, similar to Section 5 we can do a search over all values of $\alpha$ and $u$ to obtain the best time/space tradeoff. For various $\delta$ the resulting tradeoffs are depicted in Figure 7. Note that in the limit $\delta \to 0$, we have $\alpha_0, \alpha_1 \to \sqrt{4/3}$: for $\delta \to 0$, either the blue ball in Figure 3b is completely contained in the black ball, or (by slightly decreasing $\alpha$) it is completely outside the black ball. In other words, we then always have $p \to 0$ or $p \to 1$ as $\delta \to 0$, which means that rerandomizations do not help; either the algorithm almost always succeeds, or it always fails.

To further illustrate the behavior of the limiting case $\delta \to 0$ (with $\alpha \to \sqrt{4/3}$), note:

- For $u = \frac{1}{2}$, we have $\mathrm{S}_{1,2} \approx 2^{0.2075d+o(d)}$ and $\mathrm{T}_2 = (5/4)^{d/2+o(d)} \approx 2^{0.1610d+o(d)}$.
- For $u = 1$, we have $\mathrm{S}_{1,2} \approx 2^{0.2925d+o(d)}$ and $\mathrm{T}_2 = (9/8)^{d/2+o(d)} \approx 2^{0.0850d+o(d)}$.

In the limit of large $u \to \sqrt{\frac{\alpha^2}{\alpha^2-1}}$ we again obtain $\mathrm{S}_{1,2}, \mathrm{T}_1 \to 2^{\omega(d)}$ and $\mathrm{T}_2 \to 2^{o(d)}$ (regardless of $\delta$) similar to solving CVPP without a distance guarantee.

## 8 Solving CVP(P)$_\kappa$

Besides BDD, where $\boldsymbol{t}$ lies unusually close to the lattice, another easier variant of CVP is the Approximate Closest Vector Problem. Given a lattice $\mathcal{L}$ and a target vector $\boldsymbol{t} \in \mathbb{R}^d$, approximate CVP with approximation factor $\kappa$ asks to find a vector $\boldsymbol{s} \in \mathcal{L}$ such that $\|\boldsymbol{s}-\boldsymbol{t}\|$ is at most a factor $\kappa$ larger than the real distance from $\boldsymbol{t}$ to $\mathcal{L}$. For random instances $\boldsymbol{t}$, by the Gaussian heuristic this means a lattice vector $\boldsymbol{s}$ counts as a solution for approximate CVP with approximation factor $\kappa$ iff $\boldsymbol{s}$ lies at distance at most $\kappa \cdot \lambda_1(\mathcal{L})$ from $\boldsymbol{t}$.

**Without preprocessing.** Similar to BDD, it seems impossible to improve upon the complexities of sieving for solving CVP directly (without preprocessing), unless $\kappa$ is very large (at least superconstant) – with less time than $2^{0.292d+o(d)}$ and less memory than $2^{0.208d+o(d)}$, we cannot even solve approximate SVP with constant approximation factors $\kappa$, let alone solve approximate CVP. This again seems closely related to a long-standing open problem in (heuristic) lattice sieving: is it possible to obtain significantly better asymptotic time/space complexities for sieving (for SVP) when an approximate solution suffices? We leave this problem for future work.

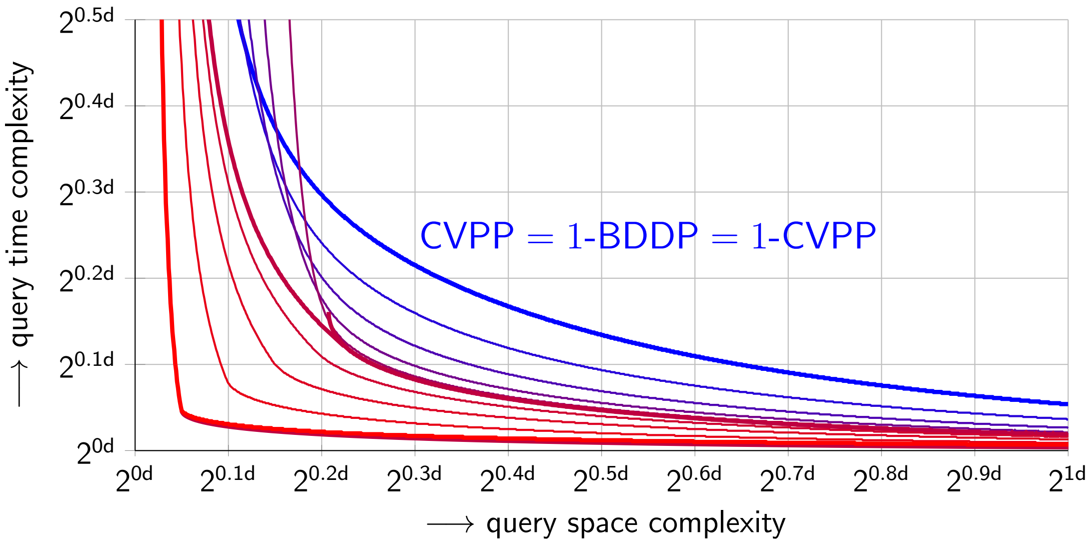


**Fig. 7.** Heuristic complexities for solving Bounded Distance Decoding (BDD) and approximate CVP with preprocessing. The top curves (blue-purple) correspond to BDD for different $\delta \in \{0, 0.2, \dots, 1\}$, where smaller $\delta$ correspond to better complexities when using more space, but also a larger lower bound $\alpha_0$ on $\alpha$ leading to potentially worse query time complexities when the space complexity is small. For $\delta \to 0$, there is only one allowed value $\alpha = \sqrt{4/3}$, and the tradeoff follows from varying $u$; this tradeoff is indicated by the thick purple line. The curves below this line (purple-red) correspond to approximate CVP with approximation factors $\{\sqrt{4/3}, 1.2, 1.3, 1.5, 2\}$. Larger approximation factors correspond to better complexities, while the (tail of the) curve for $\kappa = \sqrt{4/3}$ overlaps with the BDD curve for $\delta = 0$.

**With preprocessing.** For the approach from Sections 4–6, we may hope to further improve upon the query complexities (after the preprocessing phase), similar to BDD. Without rerandomizations, instead of reducing $\boldsymbol{t}$ to a vector $\boldsymbol{t}'$ of norm at most $\lambda_1(\mathcal{L})$, as is needed for solving exact CVP ($\beta = 1$), we now update the analysis to take into account that we want to make sure that the reduced vector $\boldsymbol{t}'$ has norm at most $\kappa \cdot \lambda_1(\mathcal{L})$ ($\beta = \kappa$). If this is the case, then the vector $\boldsymbol{t} - \boldsymbol{t}'$ is a lattice vector lying at distance at most $\kappa \cdot \lambda_1(\mathcal{L})$ from $\boldsymbol{t}$, which w.h.p. qualifies as a solution. This means that instead of substituting $\beta = 1$ in Lemma 7 for exact CVPP (without rerandomizations), we now substitute $\beta = \kappa$, leading to the condition that $\alpha^4 - 4\kappa^2\alpha^2 + 4\beta^2 \leq 0$. By a similar analysis $\alpha$ must therefore be larger than the smallest root $r_1 = \sqrt{2\kappa(\kappa - \sqrt{\kappa^2 - 1})}$ of this quartic polynomial. A sanity check shows that $\kappa = 1$, corresponding to exact CVP, indeed results in $\alpha \geq \sqrt{2}$, while in the limit of $\kappa \to \infty$ a value $\alpha \approx 1$ suffices to obtain a vector $\boldsymbol{t}'$ of norm at most $\kappa \cdot \lambda_1(\mathcal{L})$. In other words, to solve approximate CVP with very large (constant) approximation factors, a preprocessed list of size $(1 + \varepsilon)^{d+o(d)}$ suffices. Further note that $\kappa = \sqrt{4/3}$ leads to the same value $\alpha = \sqrt{4/3}$ as in BDD with $\delta \to 0$.

**Randomized slicing.** With rerandomizations, the analysis can be updated as follows. Instead of asking that the single closest vector $\boldsymbol{s}$ at distance $\lambda_1(\mathcal{L})$ from $\boldsymbol{t}'$ is contained in the list $L$ (and has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$), we now want that at least one of the $\kappa^{d+o(d)}$ lattice vectors $\boldsymbol{s}$ at distance at most $\kappa \cdot \lambda_1(\mathcal{L})$ from $\boldsymbol{t}'$ has norm at most $\alpha \cdot \lambda_1(\mathcal{L})$. This leads to the following alternative definition for the success probability:

$$p_{\alpha,\kappa} = \kappa^d \cdot \frac{|\mathcal{B}(\boldsymbol{t}', \kappa) \cap \mathcal{B}(\boldsymbol{0}, \alpha)|}{|\mathcal{B}(\boldsymbol{t}', \kappa)|} = \left(\frac{-\beta^4 + 2\beta^2\left(\kappa^2 + \alpha^2\right) - \left(\alpha^2 - \kappa^2\right)^2}{4\beta^2}\right)^{d/2+o(d)}. \quad (19)$$

The conditions on the parameter $\alpha$ are analogous to BDD: we require that the asymptotic formulas for $p$ lie in the range $[0,1]$. More precisely, if this asymptotic expression exceeds 1, then the conditions of Lemma 8 are not met, and we instead have $p = 1-o(1)$. As increasing $\alpha$ beyond the smallest value for which $p \approx 1$ only leads to worse complexities, we can simply assume that $\alpha$ is chosen such that for these asymptotic expressions, $p \leq 1$. Substituting the above expressions for $p$, with rerandomizations we now obtain the following result.

**Theorem 5.** *Let $\alpha \in (\alpha_0, \alpha_1)$ where $\alpha_0, \alpha_1$ are the smallest values larger than* 1 *such that $p_{\alpha_0,\kappa} = 0$ and $p_{\alpha_1,\kappa} = 1$ respectively, with $p_{\alpha,\kappa}$ as in* (19). *Let $u \in (\sqrt{\frac{\alpha^2-1}{\alpha^2}}, \sqrt{\frac{\alpha^2}{\alpha^2-1}})$. Using approximate Voronoi cells with randomized slicing, we can heuristically solve CVPP$_\kappa$ with complexities $\mathrm{S}_{1,2}, \mathrm{T}_{1,2}$ described in Theorem 3 with $p_\alpha$ replaced by $p_{\alpha,\kappa}$.*

Various optimized tradeoffs for different values $\kappa$ are depicted in Figure 7, with the curve for $\kappa = \sqrt{4/3}$ partially overlapping with the BDD-curve for $\delta \to 0$. Recall that for 0-BDD, the interval for $\alpha$ only contains one value, resulting in one tradeoff. For $\sqrt{4/3}$-CVP, this interval is not just one value, and choosing $\alpha < \sqrt{4/3}$ leads to better space complexities than for 0-BDD.

Also observe that from the bottom (thick, red) curve in Figure 7, we can see that (after preprocessing the lattice) we can solve 2-CVPP in time and space both less than $2^{0.05d+o(d)}$, which follows from setting $\alpha \approx 1.035$ and $u \approx 0.381$, and only storing a small list of vectors $L$ in memory. As $\kappa$ increases, both $\alpha_0, \alpha_1$ tend to $1 + 1/(8\kappa^2) + O(\kappa^{-4})$, and for arbitrary superconstant $\kappa$ we therefore obtain query time and space complexities both tending to $2^{o(d)}$.

**Corollary 2.** *For arbitrary $\varepsilon > 0$, for sufficiently large $\kappa$ we can use approximate Voronoi cells to heuristically solve approximate CVPP with approximation factor $\kappa$ with preprocessing time $\mathrm{T}_1 = (3/2)^{d/2+o(d)}$, preprocessing space $\mathrm{S}_1 = (4/3)^{d/2+o(d)}$, and query time and space complexities $\mathrm{T}_2, \mathrm{S}_2 = 2^{\varepsilon d+o(d)}$. In particular, for $\kappa = \omega(1)$, we can solve approximate CVPP in $2^{o(d)}$ time and space.*

The corresponding algorithm is rather simple as well: (1) run a standard sieve for solving SVP; (2) discard all but the $2^{\varepsilon d+o(d)}$ shortest vectors found by the algorithm; and (3) use Algorithm 5 to find a sufficiently close lattice vector to $\boldsymbol{t}$. To obtain slightly subexponential query complexities one does not even need rerandomizations or nearest neighbor searching; these subexponential costs follow directly from $\kappa = \omega(1)$.

To compare Corollary 2 with previous work, note that $\alpha_0, \alpha_1$ both tend to $1+1/(8\kappa^2)+O(\kappa^{-4})$ as $\kappa$ grows. The query space and time complexities are both proportional to $\alpha^{\Theta(d)}$. To obtain polynomial query complexities, we can solve for $\kappa$, leading to the following result.

**Corollary 3.** *Using approximate Voronoi cells we can heuristically solve approximate CVPP with approximation factor $\kappa$ in polynomial query time and space iff $\kappa = \Omega(\sqrt{d/\log d})$.*

*Proof.* The query time and space complexities are given by $\alpha^{\Theta(d)}$, where $\alpha = 1 + \Theta(1/\kappa^2)$. To obtain polynomial complexities in $d$, we must have $\alpha^{\Theta(d)} = d^{O(1)}$, or equivalently:

$$1 + \Theta\left(\frac{1}{\kappa^2}\right) = \alpha = d^{O(1/d)} = \exp O\left(\frac{\log d}{d}\right) = 1 + O\left(\frac{\log d}{d}\right). \tag{20}$$

Solving for $\kappa$ leads to the given relation between $\kappa$ and $d$.

This is equivalent (minus the heuristic assumptions) to a result of Aharonov and Regev [AR04], who previously showed that the decision version of CVPP with approximation factor $\kappa = \Omega(\sqrt{d/\log d})$ can provably be solved in polynomial time. This also heuristically improves upon results of [LLS90, DRS14], who showed how to solve the search-version of CVPP with polynomial time and space complexities for $\kappa = O(d^{3/2})$ and $\kappa = \Omega(d/\sqrt{\log d})$ respectively. These comparisons suggest that this sieving-based method may well be optimal from a theoretical point of view as well.

## 9 Open problems

**Faster enumeration with sieving as a CVPP subroutine.** As stated in the introduction, the most likely practical application of the approximate Voronoi cell approach is as a subroutine within enumeration methods, to speed up the searches in the bottom part of the tree. An open question remains whether this would indeed lead to faster algorithms for SVP/CVP in practice, or if the preprocessing/query costs are too high. As a concrete example, one might for instance try running enumeration (with pruning) in dimension 120, where approximate Voronoi cells (with randomized slicing) is used in a sublattice of dimension 80 as a subroutine. Note however that the CVP instances in the 80-dimensional sublattice might actually be BDD instances as well, and the pruning used in enumeration heavily influences the type of CVP instances and the number of CVP instances encountered in the bottom part of the tree. Optimizing all parameters involved may be a complex task.

**Sieving in the dual lattice.** For the application of CVPP within enumeration, observe that a decisional CVPP oracle may actually be sufficient; most branches of the enumeration tree will not lead to a solution, and therefore in most cases running an accurate decision-CVPP oracle is enough to determine that this subtree is not the right subtree. For those few subtrees which potentially do contain a solution, one could then run a full CVP(P) algorithm at a slightly higher cost. Improving the complexities for the decision-version of CVPP may therefore be an interesting future direction, and perhaps one approach could be to combine this with ideas from [AR04], by running a lattice sieve on the dual lattice to find many short vectors in the dual lattice, which can then be used to check if a target vector lies close to the primal lattice or not.

**Quantum complexities.** As lattice-based cryptography is often advertised as being *quantum-resistant* [BBD09], it is also important to study the (potential) asymptotic complexities of SVP/CVP-algorithms on quantum computers, so that the parameters can be chosen to be secure in a quantum world as well. For lattice sieving for solving SVP, the time complexity exponent potentially decreases by approximately 25% [LMvdP15], and so for CVP(P) one might also expect the exponents to decrease by approximately 25%. Studying the exact quantum asymptotics of sieving for CVP(P) is left for future work.

## Acknowledgments

The author is indebted to Léo Ducas, whose ideas and suggestions on this topic motivated work on this paper. The author further thanks Vadim Lyubashevsky and Oded Regev for their comments on the relevance of a subexponential time CVPP algorithm requiring (super)exponential space. The author is supported by the SNSF ERC Transfer Grant CRETP2-166734 FELICITY.

## References

ADRS15. Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in $2^n$ time via discrete Gaussian sampling. In *STOC*, pages 733–742, 2015.

ADS15. Divesh Aggarwal, Daniel Dadush, and Noah Stephens-Davidowitz. Solving the closest vector problem in $2^n$ time – the discrete Gaussian strikes again! In *FOCS*, pages 563–582, 2015.

AEVZ02. Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

AI06. Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

AIL$^+$15. Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, pages 1225–1233, 2015.

AKKV05. Misha Alekhnovich, Subhash Khot, Guy Kindler, and Nisheeth Vishnoi. Hardness of approximating the closest vector problem with pre-processing. In *FOCS*, pages 216–225, 2005.

AKS01a. Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. An overview of the sieve algorithm for the shortest lattice vector problem. In *CALC*, pages 1–3, 2001.

AKS01b. Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.

ALRW17. Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *SODA*, 2017.

AR04. Dorit Aharonov and Oded Regev. Lattice problems in $\mathsf{NP} \cap \mathsf{coNP}$. In *FOCS*, pages 362–371, 2004.

AR15. Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *STOC*, pages 793–801, 2015.

BBD09. Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-quantum cryptography*. Springer, 2009.

BD15. Nicolas Bonifas and Daniel Dadush. Short paths on the Voronoi graph and the closest vector problem with preprocessing. In *SODA*, pages 295–314, 2015.

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, pages 10–24, 2016.

BGJ14. Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. In *ANTS*, pages 49–70, 2014.

BGJ15. Anja Becker, Nicolas Gama, and Antoine Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. *Cryptology ePrint Archive, Report 2015/522*, pages 1–14, 2015.

BL16. Anja Becker and Thijs Laarhoven. Efficient (ideal) lattice sieving using cross-polytope LSH. In *AFRICACRYPT*, pages 3–23, 2016.

BLS16. Shi Bai, Thijs Laarhoven, and Damien Stehlé. Tuple lattice sieving. In *ANTS*, pages 146–162, 2016.

BNvdP16. Joppe W. Bos, Michael Naehrig, and Joop van de Pol. Sieving for shortest vectors in ideal lattices: a practical perspective. *International Journal of Applied Cryptography*, pages 1–23, 2016.

Cha02. Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

Chr17. Tobias Christiani. A framework for similarity search with space-time tradeoffs using locality-sensitive filtering. In *SODA*, 2017.

CMP$^+$16. Fábio Correia, Artur Mariano, Alberto Proenca, Christian Bischof, and Erik Agrell. Parallel improved Schnorr-Euchner enumeration SE++ for the CVP and SVP. In *PDP*, pages 596–603, 2016.

CN11. Yuanmi Chen and Phong Q. Nguyên. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, pages 1–20, 2011.

DRS14. Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. On the closest vector problem with a distance guarantee. In *CCC*, pages 98–109, 2014.

dt16. The FPLLL development team. fplll, a lattice reduction library. Available at `https://github.com/fplll/fplll`, 2016.

FBB$^+$14. Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for speed. In *LATINCRYPT*, pages 288–305, 2014.

FM02. Ulrich Feige and Daniele Micciancio. The inapproximability of lattice and coding problems with preprocessing. In *CCC*, pages 32–40, 2002.

FP85. Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice. *Mathematics of Computation*, 44(170):463–471, 1985.

GNR10. Nicolas Gama, Phong Q. Nguyên, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

HK16. Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate $k$-list problem in Euclidean norm. *Preprint*, pages 1–35, 2016.

HSB$^+$10. Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In *AFRICACRYPT*, pages 52–68, 2010.

IKMT14. Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel Gauss Sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In *PKC*, pages 411–428, 2014.

IM98. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

Kan83. Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC*, pages 193–206, 1983.

KF16. Paul Kirchner and Pierre-Alain Fouque. Time-memory trade-off for lattice enumeration in a ball. *Cryptology ePrint Archive, Report 2016/222*, 2016.

Laa15a. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.

Laa15b. Thijs Laarhoven. Tradeoffs for nearest neighbors on the sphere. pages 1–16, 2015.

Laa16. Thijs Laarhoven. Sieving for closest lattice vectors (with preprocessing). In *SAC*, 2016.

LdW15. Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In *LATINCRYPT*, pages 101–118, 2015.

LLS90. Jeffrey C. Lagarias, Hendrik W. Lenstra, and Claus-Peter Schnorr. Korkin-Zolotarev bases and successive minima of a lattice and its reciprocal lattice. *Combinatorica*, 10(4):333–348, 1990.

LMvdP15. Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2):375–400, 2015.

MB16. Artur Mariano and Christian Bischof. Enhancing the scalability and memory usage of Hash-Sieve on multi-core CPUs. In *PDP*, pages 545–552, 2016.

Mic01. Daniele Micciancio. The hardness of the closest vector problem with preprocessing. *IEEE Transactions on Information Theory*, 47(3):1212–1215, 2001.

Mic08. Daniele Micciancio. Efficient reductions among lattice problems. In *SODA*, pages 84–93, 2008.

MLB15. Artur Mariano, Thijs Laarhoven, and Christian Bischof. Parallel (probable) lock-free Hash-Sieve: a practical sieving algorithm for the SVP. In *ICPP*, pages 590–599, 2015.

MLB17. Artur Mariano, Thijs Laarhoven, and Christian Bischof. A parallel variant of LDSieve for the SVP on lattices. *PDP*, 2017.

MODB14. Artur Mariano, Özgür Dagdelen, and Christian Bischof. A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In *Euro-Par 2014*, pages 48–59, 2014.

MS11. Benjamin Milde and Michael Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In *PACT*, pages 452–458, 2011.

MTB14. Artur Mariano, Shahar Timnat, and Christian Bischof. Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In *SBAC-PAD*, pages 278–285, 2014.

MV10a. Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In *STOC*, pages 351–358, 2010.

MV10b. Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA*, pages 1468–1480, 2010.

MW15. Daniele Micciancio and Michael Walter. Fast lattice point enumeration with minimal overhead. In *SODA*, pages 276–294, 2015.

NV08. Phong Q. Nguyên and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.

ODS10. Özgür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In *EURO-PAR*, pages 211–222, 2010.

PS09. Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. *Cryptology ePrint Archive, Report 2009/605*, pages 1–7, 2009.

Reg04. Oded Regev. Improved inapproximability of lattice and coding problems with preprocessing. *IEEE Transactions on Information Theory*, 50(9):2031–2037, 2004.

Sch87. Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2–3):201–224, 1987.

Sch11. Michael Schneider. Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In *WALCOM*, pages 89–97, 2011.

Sch13. Michael Schneider. Sieving for short vectors in ideal lattices. In *AFRICACRYPT*, pages 375–391, 2013.

SE94. Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(2–3):181–199, 1994.

SFS09. Naftali Sommer, Meir Feder, and Ofir Shalvi. Finding the closest lattice point by iterative slicing. *SIAM Journal of Discrete Mathematics*, 23(2):715–731, 2009.

SG15. Michael Schneider and Nicolas Gama. SVP challenge, 2015.

Ste16. Noah Stephens-Davidowitz. Dimension-preserving reductions between lattice problems. *Available at http://noahsd.com/latticeproblems.pdf.*, pages 1–6, 2016.

VB96. Emanuele Viterbo and Ezio Biglieri. Computing the Voronoi cell of a lattice: the diamond-cutting algorithm. *IEEE Transactions on Information Theory*, 42(1):161–171, 1996.

vdP11. Joop van de Pol. Lattice-based cryptography. Master's thesis, Eindhoven University of Technology, 2011.

WLTB11. Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In *ASIACCS*, pages 1–9, 2011.

WSSJ14. Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *arXiv:1408.2927 [cs.DS]*, pages 1–29, 2014.

ZPH13. Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. In *SAC*, pages 29–47, 2013.