Tuple Lattice Sieving

Shi Bai[1], Thijs Laarhoven[2], and Damien Stehlé[1]

[1]ENS de Lyon, Laboratoire LIP, U. Lyon, CNRS, ENSL, INRIA, UCBL, Lyon, France
`shi.bai@ens-lyon.fr`, `damien.stehle@ens-lyon.fr`
[2]IBM Research, Rüschlikon, Switzerland
`mail@thijs.com`

Abstract

Lattice sieving is asymptotically the fastest approach for solving the shortest vector problem (SVP) on Euclidean lattices. All known sieving algorithms for solving SVP require space which (heuristically) grows as $2^{0.2075n+o(n)}$, where n is the lattice dimension. In high dimensions, the memory requirement becomes a limiting factor for running these algorithms, making them uncompetitive with enumeration algorithms, despite their superior asymptotic time complexity.

We generalize sieving algorithms to solve SVP with less memory. We consider reductions of tuples of vectors rather than pairs of vectors as existing sieve algorithms do. For triples, we estimate that the space requirement scales as $2^{0.1887n+o(n)}$. The naive algorithm for this triple sieve runs in time $2^{0.5661n+o(n)}$. With appropriate filtering of pairs, we reduce the time complexity to $2^{0.4812n+o(n)}$ while keeping the same space complexity. We further analyze the effects of using larger tuples for reduction, and conjecture how this provides a continuous tradeoff between the memory-intensive sieving and the asymptotically slower enumeration.

1 Introduction

The shortest vector problem (SVP) aims to find a shortest non-zero vector in a Euclidean lattice, starting from an arbitrary basis of the lattice. Solving SVP is the cost-dominating component to cryptanalyze lattice-based cryptosystems [17,22]. The currently best known algorithm with proven correctness and complexity bounds [3] requires memory and time $2^{n+o(n)}$, where n is the lattice dimension. The best known provable algorithm requiring less space than this is due to Kannan [18]. Its space requirement is polynomial, but its running-time is $n^{n/(2e)+o(n)}$ [14, 15], which is asymptotically slower than sieving. In practice, however, the most competitive implementations rely on variants of Kannan's algorithm that are asymptotically slower and whose correctness can only be guaranteed heuristically [6, 8, 11].

Lattice sieving. The first provable lattice sieving algorithm dates back to the work of Ajtai, Kumar, and Sivakumar (AKS) [4]. The AKS algorithm has been progressively refined and simplified in a series of works [13,23,25], resulting in the ListSieve algorithms of Micciancio and Voulgaris [23]. Currently, the fastest provable variant of lattice sieving runs in time $2^{2.465n+o(n)}$ and space $2^{1.325n+o(n)}$ [26] (see [21] for a quantum acceleration).

In practice, heuristic variants of the lattice sieving algorithms are found to be more efficient. Nguyen and Vidick [25] exhibited a version of AKS that can be heuristically argued correct and which requires a running-time of $(4/3)^{n+o(n)} \approx 2^{0.4150n+o(n)}$ and space of $(4/3)^{n/2+o(n)} \approx 2^{0.2075n+o(n)}$. Micciancio and Voulgaris [23] later proposed a heuristic variant of their ListSieve algorithm, namely the GaussSieve algorithm. In practice, the GaussSieve seems to perform well compared to the other variants [23]. It has been investigated further in a series of works (see, e.g., [9]); and its variants were used to solve some lattice challenges [19]. The GaussSieve algorithm is one of the most promising candidates for lattice sieving algorithms in practice.

Recently, nearest neighbor search techniques have been used to accelerate heuristic sieving algorithms further. The technique was first used in the context of lattice sieving by Laarhoven in [20]. Currently, the best variant is due to Becker, Ducas, Gama, and Laarhoven [5], which has a time complexity of $(3/2)^{n/2+o(n)} \approx 2^{0.2925n+o(n)}$ and space complexity $(4/3)^{n/2+o(n)}$.

Reducing the space complexity. Note that the upper bound of the space requirement of all the aforementioned sieve-based algorithms is at least $(4/3)^{n/2+o(n)}$. It is conjectured (and experimentally observed [23, 25]) that such space is typically consumed during the execution of these algorithms as well. Let us explain where this bound stems from.

Before we get vectors of norms close to the lattice minimum, sieving algorithms involve lattice vectors that seem uniformly distributed on an n-dimensional sphere (or super-thin crust of a ball), and create shorter vectors by subtracting two lattice vectors on the sphere that happen to be sufficiently close. Concretely, this occurs when the angle between the two vectors is less than $\pi/3$: their difference results in a shorter vector. One way to heuristically obtain the space complexity bound is to observe that if we have a point $\boldsymbol{v}$ on a sphere, then it covers a fraction $\sin^n(\pi/3) = (3/4)^{n/2}$ of this sphere of points at angle at most $\pi/3$ from this point. If we further assume that each point covers a different part of the sphere, and overlaps are generally small and almost negligible, then we see that we need about $(4/3)^{n/2} \approx 2^{0.2075n}$ vectors to cover the sphere. With this many covering points, any extra point expects to see an existing nearby covering point; and their difference leads to a short vector.

Contributions. We propose tuple variants for the ListSieve and GaussSieve algorithms, which we call TupleSieve and TupleMinkowskiSieve, whose memory footprint is smaller than $2^{0.2075n+o(n)}$. The main idea is to attempt to create shorter vectors by looking at triples, quadruples, etc. of vectors rather than pairs of vectors. For triples of vectors, we estimate the space complexity by $2^{0.1887n+o(n)}$. For quadruples, the space complexity is about $2^{0.1724n+o(n)}$. For growing k (which remains small compared to n), the space complexity seems to scale as $k^{n/k(1+o(1))}$, while the running-time seems to scale as $k^{n(1+o(1))}$. We conjecture that TupleSieve provides a continuous tradeoff between the memory-intensive sieving algorithms and the asymptotically slower Kannan enumeration algorithm.[1]

The time complexity of TupleSieve grows very fast with the size k of tuples. If implemented naively, the algorithm has a time complexity which is the k-th power of its space complexity. We show that this naive complexity can be reduced by a filtering of pairs of vectors, to remove those that are too unlikely to be extended to a useful triple. More concretely, in the case of $k = 3$, a triple is useful if either its underlying pairs are useful, or one pair is not too close to orthogonal and the third vector is close to that pair difference. The underlying thought

[1]Recently Fouque and Kirchner [10] show it is possible to obtain a similar time/space tradeoff, by using more memory for enumeration. Our tradeoff instead follows from using less memory for sieving.

is that if two vectors are almost orthogonal, then a third vector which is far away from both vectors is unlikely to lead to a triple reduction. For $k = 3$, filtering allows to decrease the $2^{0.5661n+o(n)}$ running-time of the naive algorithm to $2^{0.4812n+o(n)}$.

On removing the heuristics. The correctness of the tuple lattice sieving algorithms presented in this paper is heuristic, as is their complexity analysis. These heuristics are backed by experiments in Section 6, but it would be preferable to also better apprehend them in theory.

The main obstacle towards proving correctness of sieving algorithms is that the vector combinations may all result in the $\mathbf{0}$ vector after some stage, leading us to miss shortest non-zero vectors. This difficulty is typically circumvented by adding perturbations to the vectors to "hide" the lattice structure to the algorithm (see, e.g., [23]). We believe such a technique could also be applied to our algorithms, although with a significant cost increase.

The problem of obtaining rigorous bounds on the space complexity is, in our opinion, much more challenging (and mathematically enticing). Bounding the space complexity of ListSieve and GaussSieve can be reformulated in terms of spherical codes: how large can a list of points on the unit sphere be if we assume that all pairs of points have angle at least $\pi/3$ (i.e., their difference has norm > 1)? In the case of k-tuple sieving, the list of points on the unit sphere is such that any sum of at most k list vectors has norm > 1. As the constraint gets stronger with increasing k, the maximum list size cannot increase. Can it be shown that it decreases?

Road-map. Section 2 gives preliminaries on the geometry of lattices. Section 3 presents the TupleSieve and its complexity analysis. Section 4 describes the filtered TupleSieve, which is designed to optimize its running-time. The TupleSieve is a generalization of ListSieve. A GaussSieve-like variant of the TupleSieve, called TupleMinkowskiSieve, is discussed in Section 5: similarly to GaussSieve and ListSieve, TupleMinkowskiSieve is more complex to study than the TupleSieve but has more potential in practice. Finally, experimental results are discussed in Section 6: we test heuristics used in the theoretical analysis of the TupleSieve, and we test the practical efficiency of TupleMinkowskiSieve.

2 Preliminaries

We first introduce some notation and recall some elementary geometric facts that we will use when analyzing tuple lattice sieving algorithms.

Let $\mathcal{B}(\boldsymbol{v}, r)$ be the ball of radius r around $\boldsymbol{v} \in \mathbb{R}^n$. Denote in short $\mathcal{B}(\boldsymbol{v}) = \mathcal{B}(\boldsymbol{v}, 1)$. and $\mathcal{B} = \mathcal{B}(\mathbf{0}, 1)$. We let $\mathcal{S}$ denote the unit sphere. Let us write $|\Omega|$ for the volume of a (measurable) set $\Omega \subset \mathbb{R}^n$. For a set Ω of finite measure, we let $U(\Omega)$ denote the uniform distribution on Ω. For two functions A, B of n, we write $A \propto B$ if there exist two constants c and c' such that $A \leq n^c \cdot B$ and $B \leq n^{c'} \cdot A$ for large n.

2.1 Geometric properties of the unit sphere

We first recall the following simple geometric observation, regarding the norm of the difference between two vectors on the sphere.

Lemma 2.1 ((Cosine law)). *For vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$ with angle θ we have*

$$\|\boldsymbol{v}_1 - \boldsymbol{v}_2\|^2 = \|\boldsymbol{v}_1\|^2 + \|\boldsymbol{v}_2\|^2 - 2\|\boldsymbol{v}_1\|\|\boldsymbol{v}_2\| \cos\theta.$$

Thus for two unit vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathcal{S}$ with angle θ, we have $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\| = \sqrt{2(1-\cos\theta)}$.

Throughout the paper, we are interested in the covering of the unit sphere $\mathcal{S}$ by unit balls $\mathcal{B}(\boldsymbol{v})$: how many vectors $\boldsymbol{v}$ are needed such that the union of several balls $\mathcal{B}(\boldsymbol{v})$ covers (most of) $\mathcal{S}$? The following lemma considers the probability mass of different portions of the unit sphere, when considering the uniform distribution over the sphere.

Lemma 2.2. *The density function $f(\theta)$ of the angle $\theta \in [0, \pi/2)$ between any fixed vector in $\mathcal{S}$ and a vector sampled independently from $U(\mathcal{S})$ satisfies $f(\theta) \propto (\sin\theta)^n$.*

A spherical cap of height h in a ball of radius r is any set that may be obtained by applying an isometry to $\{\boldsymbol{x} \in \mathcal{B}(\boldsymbol{0}, r) : x_n \geq r - h\}$. We let $C(h, r)$ denote the volume of a spherical cap with parameters h and r.

Lemma 2.3 ([23, Le. 4.1])**.** *A spherical cap of height h and radius r has volume*

$$C(h, r) \propto (r^2 - (r-h)^2)^{n/2} \cdot |\mathcal{B}|.$$

The following lemma will be useful for estimating the part of the sphere that is covered by a (unit) ball at arbitrary distance from the origin.

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

$$|\mathcal{B}(\boldsymbol{v}_1, r_1) \cap \mathcal{B}(\boldsymbol{v}_2, r_2)| \propto \left(\frac{-d^4 + 2d^2\left(r_1^2 + r_2^2\right) - \left(r_1^2 - r_2^2\right)^2}{4d^2}\right)^{n/2} \cdot |\mathcal{B}|.$$

In particular, we have $|\mathcal{B}(\boldsymbol{v}_1) \cap \mathcal{B}(\boldsymbol{v}_2)| \propto (1 - d^2/4)^{\frac{n}{2}} \cdot |\mathcal{B}|$.

Proof. Without loss of generality, we assume that $\boldsymbol{v}_1 = \boldsymbol{0}$ and $\boldsymbol{v}_2 = d\boldsymbol{e}_1$. The intersection of these two balls is partitioned in two spherical caps. Let $\boldsymbol{a}$ be a point at distance r_1 from $\boldsymbol{0}$ and distance r_2 from $d\boldsymbol{e}_1$ and consider the triangle formed by $\boldsymbol{0}, \boldsymbol{a}$ and $d\boldsymbol{e}_1$. This triangle has sides d, r_1, r_2. To find $|\langle \boldsymbol{a}, d\boldsymbol{e}_1\rangle|$, we use the law of cosines in the point $\boldsymbol{0}$ to establish that this angle in the triangle satisfies $\cos\phi = (r_1^2 + d^2 - r_2^2)/(2r_1 d)$. Using the cosine definition, we then know that $\cos\phi = |\langle \boldsymbol{a}, d\boldsymbol{e}_1\rangle|/r_1$. We define $x = |\langle \boldsymbol{a}, d\boldsymbol{e}_1\rangle| = (r_1^2 + d^2 - r_2^2)/(2d)$. The spherical cap associated to the ball of radius r_1 has volume

$$C(r_1 - x, r_1) \propto \left(\frac{-d^4 + 2d^2\left(r_1^2 + r_2^2\right) - \left(r_1^2 - r_2^2\right)^2}{4d^2}\right)^{n/2} \cdot |\mathcal{B}|.$$

The other spherical cap has the same volume $C(r_2 - (d - x), r_2) \propto C(r_1 - x, r_1)$. As the total volume of the intersection is the sum of these two values, the result follows. □

2.2 Euclidean lattices

A (full-rank) lattice of $\mathbb{R}^n$ is the set $\mathcal{L}[(\boldsymbol{b}_i)_i] = \sum_i \mathbb{Z}\boldsymbol{b}_i \subset \mathbb{R}^n$ of all integer combinations of some n linearly independent vectors $(\boldsymbol{b}_i)_{i \leq n}$ of $\mathbb{R}^n$. In this setup, the vectors $(\boldsymbol{b}_i)_{i \leq n}$ are said to form a basis of $\mathcal{L}[(\boldsymbol{b}_i)_i]$. Note that any given lattice of dimension $n \geq 2$ admits infinitely many bases. A lattice $\mathcal{L}$ contains shortest non-zero vectors, and their common norm is referred to as the lattice minimum and denoted by $\lambda(\mathcal{L})$. The algorithmic task of finding a shortest non-zero vector of $\mathcal{L}[(\boldsymbol{b}_i)_i]$ given $(\boldsymbol{b}_i)_i$ as input is known as the Shortest Vector Problem (SVP).

In tuple variants of the GaussSieve algorithm (TupleMinkowskiSieve), we consider a list of lattice vectors which are tuple-wise Minkowski-reduced.

Definition 1 ((Minkowski reduction))**.** *A basis $(\boldsymbol{b}_i)_{i\leq n}$ of a lattice $\mathcal{L}$ is Minkowski-reduced if the $\boldsymbol{b}_i$'s are sorted by non-decreasing norms and, for all $i < n$, the vector $\boldsymbol{b}_i$ has minimal norm among all vectors $\boldsymbol{b} \in \mathcal{L}$ such that $(\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_{i-1}, \boldsymbol{b})$ can be extended to a basis of $\mathcal{L}$.*

Note that Gauss reduction is Minkowski reduction in the special case of $n = 2$.

3 Tuple sieving

Let us first describe the general strategy of sieving algorithms. These algorithms start with a large number of long lattice vectors, e.g., created by sampling from a discrete Gaussian over the lattice with a large standard deviation [12]. They try to gradually reduce their norms by considering simple combinations. The process of combining vectors to create shorter lattice vectors is the sieving procedure, which produces shorter and shorter lattice vectors when applied iteratively; vector combinations whose norm is above some threshold are discarded during the sieving procedure, while short combinations are kept for the next iteration.

To make sure that we are not losing too many vectors during this sieving procedure and to bound the running-time, we use a "covering" set which provably cannot become too large (e.g., the center set in the Nguyen-Vidick sieve [25], or the list in the Micciancio-Voulgaris sieve [23]). In general, the vectors in this "covering" set are not too close to each other. If two current vectors are sufficiently close, we will reduce them before putting them into this "covering" set.

To prove correctness (i.e., that shortest non-zero vectors are not avoided in the successive sieves), the standard approach is using perturbations: one adds a small error vector to each lattice vector, so that the lattice is blurred from the perspective of the sieves. This allows to argue that the probability of finding a non-zero vector cannot be arbitrarily small compared to the probability of getting the zero vector. We refer to [23] for more details.

To ease the analysis, we consider a simplified algorithm named DoubleSieve (see Algorithm 1); and we do not consider these perturbations, which appear to be an artifact of the proof of correctness, rather than a procedure necessary to make sure the algorithm succeeds.

Algorithm 1 DoubleSieve

```
1: $\mathrm{L}' \leftarrow \{\}$
2: for each $\boldsymbol{v}, \boldsymbol{w} \in \mathrm{L}$ do
3:     if $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ then
4:         $\mathrm{L}' \leftarrow \mathrm{L}' \cup \{\boldsymbol{v} \pm \boldsymbol{w}\}^2$
```

Algorithm 2 TripleSieve

```
1: $\mathrm{L}' \leftarrow \{\}$
2: for each $\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{x} \in \mathrm{L}$ do
3:     if $\|\boldsymbol{v} \pm \boldsymbol{w} \pm \boldsymbol{x}\| \leq \gamma \cdot R$ then
4:         $\mathrm{L}' \leftarrow \mathrm{L}' \cup \{\boldsymbol{v} \pm \boldsymbol{w} \pm \boldsymbol{x}\}^2$
```

The DoubleSieve follows the same main procedure as the other sieves. In short, the main point is to reduce pairs of vectors in L to get a new list L'. The norms of the vectors in L are bounded by some constant R, and the norms of the vectors in L' are then bounded by

[2]The sign choice is the same as in the step above.

$\gamma \cdot R$ for some geometric factor $\gamma < 1$. This factor γ ensures that in each iteration of the sieve we make some progress with reducing the norms of the list vectors. In the analysis we will assume $\gamma \approx 1$, so that we do not need far more points to create a large output list. Note that we can choose $\gamma = 1 - 1/\text{poly}(n)$ while ensuring the total number of sieves remains $\text{poly}(n)$: indeed, we may start with an initial radius $R \leq 2^n \cdot \lambda(\mathcal{L})$ by pre-processing the input basis with the LLL lattice reduction algorithm, and stop with a final radius $\gamma^{\text{poly}(n)} R \approx \lambda(\mathcal{L})$.

Note that a sieving step aims at reducing the norms of the current vectors by a multiplicative factor $\gamma > 1$, in order to prevent the whole procedure from stalling. As a result, perturbation-free sieving algorithms (and our extensions) are unlikely to solve SVP and seem limited to finding non-zero lattice vectors of norms $\leq t \cdot \lambda(\mathcal{L})$ for some t that is close to 1. We make do with only finding near shortest non-zero lattice vectors.

An extension (TripleSieve) of this DoubleSieve algorithm is presented in Algorithm 2 above, where instead of pairs of vectors, we consider triples to find shorter lattice vectors. Note that for pairs of vectors, considering combinations $\boldsymbol{v} \pm \boldsymbol{w}$ is the best one can do: if there exists some integer linear combination $z_1\boldsymbol{v} + z_2\boldsymbol{w}$ which has norm $< R$ for vectors $\boldsymbol{v}, \boldsymbol{w}$ of norms R then also one of the vectors $\boldsymbol{v} \pm \boldsymbol{w}$ must have norm $< R$. For triples (or tuples) it is harder to tell which finite set of linear combinations must be considered to guarantee obtaining the strongest notion of reduction in our list. The Minkowski conditions described in [24, 28] show that for triples of vectors, it suffices to consider combinations $\boldsymbol{v} \pm \boldsymbol{w}$, $\boldsymbol{v} \pm \boldsymbol{x}$, $\boldsymbol{w} \pm \boldsymbol{x}$ and $\boldsymbol{v} \pm \boldsymbol{w} \pm \boldsymbol{x}$ for reduction to achieve Minkowski-reducedness for a triple of lattice vectors $\{\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{x}\}$. If we assume that $\mathbf{0}$ belongs to L, then these are exactly the combinations considered in Algorithm 2.

3.1 Cost analysis of the DoubleSieve

We now proceed with a sketch of the cost analysis for the Double Sieve, which will serve as a guideline for analyzing the TripleSieve in Section 3.2.

Let us suppose we started with a certain list of size $|\mathrm{L}| = N$, and let $\boldsymbol{v}, \boldsymbol{w} \in \mathrm{L}$. Let both vectors have norm approximately R; significantly shorter vectors are also shorter than γR and are immediately added to L'. Now, the condition $\|\boldsymbol{v} - \boldsymbol{w}\| < \gamma R$ for $\gamma \approx 1$ equivalently corresponds to $\boldsymbol{w} \in \mathcal{B}(\mathbf{0}, R) \cap \mathcal{B}(\boldsymbol{v}, \gamma R)$. To estimate the probability of finding a vector $\boldsymbol{v} - \boldsymbol{w}$ satisfying $\|\boldsymbol{v} - \boldsymbol{w}\| < \gamma R$, we use of the following heuristic assumption, introduced in [25].

Heuristic 1. *We assume that each time DoubleSieve is called, the vectors $\boldsymbol{v}/\|\boldsymbol{v}\|$ for $\boldsymbol{v} \in \mathrm{L}$ are i.i.d. uniformly distributed points on the unit sphere.*

We may restrict ourselves to analyzing the DoubleSieve to list vectors of near-identical norms (as they are coming from a prior sieve, and if they were much shorter than expected, they could have been kept for later sieves). Further, as the DoubleSieve is scale-invariant, we may restrict the study to the case where L consists of unit vectors.

Assuming Heuristic 1 holds, we can now estimate the probability that $\boldsymbol{w} \in \mathcal{B} \cap \mathcal{B}(\boldsymbol{v}, \gamma)$ by the relative mass of the corresponding spherical cap on the sphere. Letting $\gamma \approx 1$ and $\|\boldsymbol{v}\| \approx 1$, this probability is $p \approx \sin^n(\pi/3) = (3/4)^{n/2}$. So given any lattice vector in the list, the probability that a second vector is going to lead to a good combination which can be used for L' is proportional to $(3/4)^{n/2}$. Alternatively, one could say that each vector in the list covers a fraction $(3/4)^{n/2}$ of the sphere; vectors falling inside this spherical cap will lead to pairwise reductions, while vectors outside will not. Assuming that the intersections of these spherical caps are negligible, we therefore approximately need $1/p \approx (4/3)^{n/2}$ points to cover the entire sphere: using a list L of size $\text{poly}(n) \cdot (4/3)^{n/2}$ guarantees that with overwhelming probability, any other vector can be reduced with one of the list vectors, while if L is significantly smaller than $(4/3)^{n/2}$, then with overwhelming probability a random lattice vector is not covered by this list L, meaning we will lose many points in each iteration of the sieve.

To summarize, the crucial equation for the list size N to guarantee that $1 - o(1)$ of the sphere is covered with this list is given by

$$N \cdot \left(\frac{3}{4}\right)^{n/2} \geq 1 - o(1).$$

Finally, by taking for instance $\gamma = 1 - 1/n$, it is guaranteed that only a polynomial number of iterations is needed to go from an initial list of long lattice vectors to a list of vectors of norm at most $\lambda(\mathcal{L})$. The time complexity is therefore dominated by $\text{poly}(n)$ applications of the DoubleSieve, whose cost is quadratic in the list size N. This leads to a memory complexity of $N \propto (4/3)^{n/2}$ and a time complexity of $N^2 \propto (4/3)^n$.

3.2 Cost analysis of the TripleSieve

In the TripleSieve, not only single list vectors cover subsets of the sphere, but also sums and differences of pairs of list vectors cover parts of the sphere; if $\boldsymbol{v} - \boldsymbol{w} - \boldsymbol{x}$ is short, then $\boldsymbol{x}$ is covered by the spherical cap $\mathcal{B}(\boldsymbol{0}, R) \cap \mathcal{B}(\boldsymbol{v} - \boldsymbol{w}, \gamma R)$. So not only the N single vectors cover part of the sphere, also the roughly N^2 sums and differences of list vectors $\boldsymbol{v} \pm \boldsymbol{w}$ each cover a region of the sphere in the sense that any vector $\boldsymbol{x}$ in one of these regions can be reduced with $(\boldsymbol{v} \pm \boldsymbol{w})$. Intuitively this explains why fewer vectors will be needed to cover the entire sphere (and to guarantee that L' will not be shorter than L).

In the analysis below, we will make use of the following generalization of Heuristic 1.

Heuristic 2. *We assume that each time the TripleSieve is called, the vectors $\boldsymbol{v}/\|\boldsymbol{v}\|$ for $\boldsymbol{v} \in \mathrm{L}$ are i.i.d. uniformly distributed points on the unit sphere, and all vectors $\boldsymbol{v} \in L$ and $\boldsymbol{v} \pm \boldsymbol{w}$ for $\boldsymbol{v} \neq \boldsymbol{w} \in \mathrm{L}$, with sign choice minimizing the norm, behave as if they were independent.*

Note that the vectors $\boldsymbol{v} \pm \boldsymbol{w}$ for $\boldsymbol{v} \neq \boldsymbol{w} \in \mathrm{L}$ cannot be independent as they are deterministically obtained from a much smaller set of points. A heuristic stating that they are independent would hence be invalid. However, we still we may assume that this non-independence does not impact the analysis below.

As in the analysis of DoubleSieve, we assume that the vectors of L all lie on the unit sphere $\mathcal{S}$. Now, the vectors $\boldsymbol{v} \pm \boldsymbol{w}$ generally do not lie on $\mathcal{S}$, and the part of $\mathcal{S}$ that is covered by $\pm\boldsymbol{v} \pm \boldsymbol{w}$ depends exactly on the norm of $\boldsymbol{v} \pm \boldsymbol{w}$. Let us express this norm in terms of the angle $\theta \in [0, \pi/2]$ between $\boldsymbol{v}$ and $\boldsymbol{w}$. For unit vectors $\boldsymbol{v}$ and $\boldsymbol{w}$, we have:

$$\min \|\boldsymbol{v} \pm \boldsymbol{w}\|^2 = 2(1 - \cos\theta).$$

Next, note that if a vector has norm r, then the part of the unit sphere it covers is proportional to $(1 - r^2/4)^{n/2}$ (by Lemma 2.4). Therefore, if $\boldsymbol{v}$ and $\boldsymbol{w}$ have angle θ, then the vector $\boldsymbol{v} \pm \boldsymbol{w}$ covers a fraction of the spherical surface equal to at least:

$$g(\theta) = (1 - \|\boldsymbol{v} \pm \boldsymbol{w}\|^2/4)^{n/2} = \cos(\theta/2)^n.$$

Here we used the half-angle identity $\cos^2(\phi/2) = (1+\cos\phi)/2$ that holds for arbitrary ϕ.

We now compute the expected portion of $\mathcal{S}$ that is covered by the difference vector between two list vectors. By Lemma 2.2, the density of angles between pairs of vectors is proportional to $f(\theta) \propto \sin(\theta)^n$. The expected value of the part of $\mathcal{S}$ covered by a pair of vectors is therefore:

$$\mathbb{E}_\theta[g(\theta)] \propto \int_0^{\pi/2} f(\theta)g(\theta)d\theta = \int_0^{\pi/2} \left(\sin(\theta)\cos\left(\frac{\theta}{2}\right)\right)^n d\theta.$$

Note that the integrand is exponential in n, and so the asymptotic scaling of the entire integral is determined by the maximum value of the integrand. Ignoring polynomial terms, we have

$$\mathbb{E}_\theta[g(\theta)] \propto \left(\max_\theta \sin(\theta)\cos\left(\frac{\theta}{2}\right)\right)^n.$$

With some elementary trigonometric manipulation we see that this maximum is attained at $\theta = \arccos(1/3)$, and in this point the function takes value $4/(3\sqrt{3}) = \sqrt{16/27}$. So we obtain:

$$\mathbb{E}_\theta[g(\theta)] = \left(\frac{16}{27}\right)^{n/2}.$$

To figure out how large the list size must be to cover the entire sphere, suppose we have N points in our list. Then to guarantee that the single points and pairs of points in the list together cover a fraction $1 - o(1)$ of the sphere, assuming that the overlap between these regions is asymptotically negligible, this leads to the following condition on N:

$$N \cdot \left(\frac{3}{4}\right)^{n/2} + N^2 \cdot \left(\frac{16}{27}\right)^{n/2} \geq 1 - o(1).$$

The first term corresponds to the area covered by single list vectors, whereas the second term stems from pairs of list vectors. If N is much smaller than $(4/3)^{n/2}$, then the first term is exponentially smaller than 1 and the second term dominates.[3] So solving for N in $N^2 \cdot (16/27)^{n/2} \propto 1$, we obtain $N \propto (27/16)^{n/4} \approx 2^{0.1887n}$. Note that this list size N is strictly smaller than the DoubleSieve list size of $\propto (4/3)^{n/2} \approx 2^{0.2075n}$.

Finally, similar to the DoubleSieve, the cost of this algorithm is dominated by having to store the lists of vectors (memory), and having to consider all pairs/triples of vectors for the sieve (time). This directly leads to heuristic space and time complexities of $(27/16)^{n/4} \approx 2^{0.1887n}$ and $(27/16)^{3n/4} \approx 2^{0.5662n}$ for a naive cubic search over all triples.

3.3 Cost analysis of the k-TupleSieve

Algorithm 3 generalizes the DoubleSieve and the TripleSieve to arbitrary k-tuples. The DoubleSieve and the TripleSieve correspond to setting $k = 2$ and $k = 3$ respectively. Similarly to the DoubleSieve and the TripleSieve, the sign choice at Step 4 is identical to that of Step 3. Further, we may assume that vector $\mathbf{0}$ belongs to L so that the test of Step 3 actually allows to consider combinations of up to k vectors. Or we may argue (thanks to the analysis below) that only combinations of exactly k (non-zero) list vectors are likely to lead to norm reductions, and discard combinations of fewer vectors.

[3] The imbalance of the two terms implies that in the TripleSieve we may consider only combinations of triples of vectors and forget about combinations of pairs of vectors.

Algorithm 3 k-TupleSieve

```
1: $\mathrm{L}' \leftarrow \{\}$
2: for each $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k \in \mathrm{L}$ do
3:     if $\|\boldsymbol{v}_1 \pm \boldsymbol{v}_2 \pm \ldots \pm \boldsymbol{v}_k\| \leq \gamma \cdot R$ then
4:         $\mathrm{L}' \leftarrow \mathrm{L}' \cup \{\boldsymbol{v}_1 \pm \boldsymbol{v}_2 \pm \ldots \pm \boldsymbol{v}_k\}$
```

We now generalize the previous analyses to tuple sieving. Let k be fixed, and let $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{k-1}$ be independently sampled from $U(\mathcal{S})$. Define $\boldsymbol{v}_1' = \boldsymbol{v}_1$ and, for $1 < i \leq k$, let $\boldsymbol{v}_i'$ be either $\boldsymbol{v}_i$ or $-\boldsymbol{v}_i$, so that $\langle \boldsymbol{v}_i', \sum_{j<i} \boldsymbol{v}_j' \rangle \leq 0$. Note that the angles θ_i between $\sum_{j<i} \boldsymbol{v}_j'$ and $\boldsymbol{v}_i$ are independent and have densities proportional to $(\sin\theta)^n$ by Lemma 2.2. Further, observe that $\theta_2, \ldots, \theta_i$ fully determine $f_i := \| \sum_{j \leq i} \boldsymbol{v}_j' \|$. Indeed, we have $f_1 = 1$, and, for $i > 1$:

$$f_i^2 = f_{i-1}^2 - 2 f_{i-1} \cos\theta_i + 1.$$

Now, the proportion g_{k-1} of $\mathcal{S}$ at distance ≤ 1 from $\sum_{i<k} \boldsymbol{v}_i'$ covered by this sum is proportional to the size of the intersection of two unit balls centered at $\mathbf{0}$ and at a point at distance f_{k-1} from $\mathbf{0}$. By Lemma 2.4, we have $g_{k-1} \propto (1 - f_{k-1}^2/4)^{n/2}$. Hence we have

$$\alpha_{k-1} := \mathbb{E}_{\boldsymbol{v}_1,\ldots,\boldsymbol{v}_{k-1}}[g_{k-1}] \propto \left(\max_{\theta_2,\ldots,\theta_{k-1}} \sqrt{1 - \frac{f_{k-1}^2}{4}} \prod_{1<i<k} \sin\theta_i \right)^n . \tag{1}$$

By the above, a list size L satisfying the following suffices to cover the sphere by k-tuples.

$$\alpha_1^n L + \alpha_2^n L^2 + \cdots + \alpha_{k-1}^n L^{k-1} = 1.$$

The space requirement is $S_k = (\max_{i<k} \alpha_i^{1/i})^n$, and the time requirement T_k is S_k^k. Note that $\alpha_1 = \sqrt{4/3}$ and $\alpha_2 = \sqrt{27/16}$ as in the previous subsections.

We give estimates for the Triple and Quadruple Sieves. For $k = 3$, we have previously seen that the target function was

$$\left(\sqrt{1 - \tfrac{f_2^2}{4}} \right) \sin\theta_2 = \left(\sqrt{\tfrac{1}{2} \cos(\theta_2) + \tfrac{1}{2}} \right) \sin(\theta_2) = \cos\left(\tfrac{\theta_2}{2}\right) \sin(\theta_2) .$$

The maximum is achieved at $\theta_2 = \arccos(1/3)$ and the maximum value is $4/(3\sqrt{3})$, leading to a list size $2^{0.1887n+o(n)}$ for the TripleSieve. For $k = 4$, the target function is

$$\sqrt{\tfrac{1}{2} \sqrt{2 - 2\cos(\theta_2)} \cos(\theta_3) + \tfrac{1}{2} \cos(\theta_2) + \tfrac{1}{4}} \sin(\theta_2) \sin(\theta_3) .$$

Numerically optimizing over θ_2 and θ_3, we obtain a Quadruple Sieve list size $2^{0.1724n+o(n)}$.

For even larger tuple sizes k, Table 1 lists the numerical values for the space complexity of the k-TupleSieve for tuple sizes k from 3 to 15. The first column denotes the k-tuples used. The second column $(\log_2 |\mathrm{L}|)/n$ gives the list size estimate for the k-tuples. The columns "$\cos\theta_i$" denote the cosines for the optimized angles for the target function in Equation (1). We used function MINIMIZE in SAGEMATH [7] to optimize Equation (1), and obtained equivalent results using FINDMINIMUM in MATHEMATICA [29].

k	$\frac{\log_2(\lvert\mathrm{L}\rvert)}{n}$	$\cos\theta_2$	$\cos\theta_3$	$\cos\theta_4$	$\cos\theta_5$	$\cos\theta_6$	$\cos\theta_7$	$\cos\theta_8$	$\cos\theta_9$	$\cos\theta_{10}$	$\cos\theta_{11}$	$\cos\theta_{12}$	$\cos\theta_{13}$	$\cos\theta_{14}$
3	.1887	.333												
4	.1724	.250	.408											
5	.1587	.200	.316	.447										
6	.1473	.167	.258	.354	.471									
7	.1376	.143	.218	.293	.378	.488								
8	.1293	.125	.189	.250	.316	.395	.500							
9	.1221	.111	.167	.218	.272	.333	.408	.509						
10	.1158	.100	.149	.194	.239	.289	.346	.418	.516					
11	.1102	.091	.135	.174	.213	.255	.302	.357	.426	.522				
12	.1052	.083	.123	.158	.192	.228	.267	.312	.365	.433	.527			
13	.1007	.077	.113	.145	.175	.207	.240	.277	.320	.372	.439	.531		
14	.0967	.071	.105	.134	.161	.189	.218	.250	.286	.327	.378	.443	.534	
15	.0930	.067	.098	.124	.149	.174	.200	.228	.258	.293	.333	.383	.447	.538

Table 1: Estimates of list size for k-tuple sieving

3.4 Conjectured large-k asymptotics

Finally, we conclude this theoretical analysis with conjectured large-k asymptotics of the time and space complexities of the k-TupleSieve. For small $k = 2, 3$ we have exact algebraic expressions for the heuristic space complexities, and if we attempt to match similar expressions to the numeric data in the first column of Table 1 using the Inverse Symbolic Calculator [1], we obtain a conjectured general expression for the heuristic space complexity for arbitrary k.

Conjecture 1. *The heuristic asymptotic space complexity* $|\mathrm{L}|$ *of the* k*-TupleSieve satisfies*

$$|\mathrm{L}|^{1/n} = \frac{k^{k/(2k-2)}}{\sqrt{k+1}}.$$

This formula matches our numerically obtained results for $k = 2, \ldots, 15$ up to the first 50 digits. We plot the numerical results and the conjectured space complexity curve in Figure 1. Note that indeed for $k = 2, 3$ this formula gives $|\mathrm{L}|^{1/n} = \sqrt{4/3}$ and $|\mathrm{L}|^{1/n} = (27/16)^{1/4}$.

Assuming this formula is correct, we can study the limiting behavior of large tuples. First, observe that the expression above scales as $k^{1/k+o(1/k)}$. In other words, the list size asymptotically scales as $|\mathrm{L}| = k^{n/k+o(k)}$, and the corresponding time complexity is given by $k^{n+o(n)}$. If we let k approach n, then we see that the estimated list size approaches $|\mathrm{L}| \to n^{1+o(1)}$ while the time complexity scales as $n^{n+o(n)}$. This conjectured asymptotic scaling matches (up to constants in the exponents) the complexities of Kannan's enumeration algorithm [18]. Tuple lattice sieving could therefore be considered a way to obtain a continuous trade-off between the asymptotically fast but memory-intensive heuristic sieving algorithms (small k), and the memory-efficient and asymptotically slow enumeration methods (large k).

4 Filtered triple sieving

In the analysis of the TripleSieve above, we saw that vectors with angle θ significantly smaller than $\arccos(1/3)$ lead to a lot of the sphere being covered, but these vectors do not appear

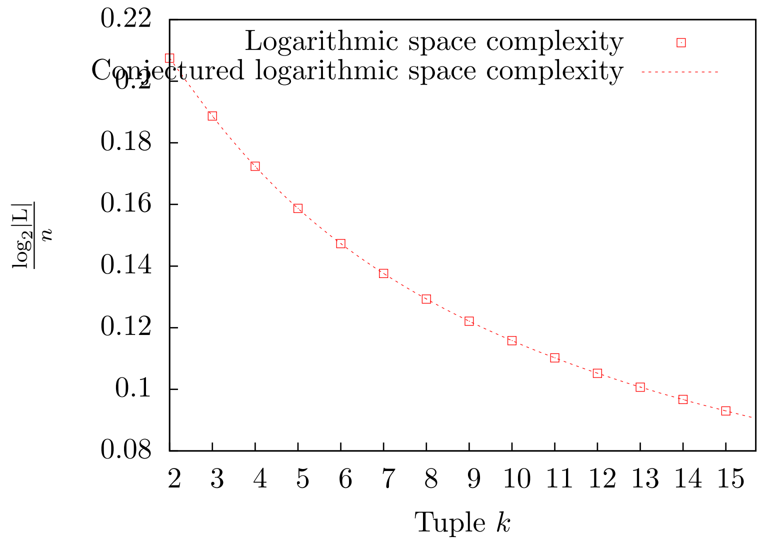

Figure 1: The numerically obtained heuristic logarithmic space complexities for k-tuple sieving up to $k = 15$, and the conjectured pattern for arbitrary k from (1).

often (thus in the algorithm, we should try to use them all); vectors at angle $\theta \approx \arccos(1/3)$ cover a decent amount of the sphere, and contribute $1 - o(1)$ of the found collisions among triples of vectors; and vectors at angle θ significantly larger than $\arccos(1/3)$ appear very often, but rarely lead to reductions even though this case appears very often (thus in the algorithm we can ignore them). This motivates the Filtered TripleSieve described in Algorithm 4.

Algorithm 4 Filtered TripleSieve

```
1: $\mathrm{L}' \leftarrow \{\}$
2: for each $\boldsymbol{v}, \boldsymbol{w} \in \mathrm{L}$ do
3:     if $|\langle \boldsymbol{v}, \boldsymbol{w} \rangle| \geq \frac{1}{3}$ then
4:         for each $\boldsymbol{x} \in \mathrm{L}$ do
5:             if $\|\boldsymbol{v} \pm \boldsymbol{w} \pm \boldsymbol{x}\| \leq \gamma R$ then
6:                 $\mathrm{L}' \leftarrow \mathrm{L}' \cup \{\boldsymbol{v} \pm \boldsymbol{w} \pm \boldsymbol{x}\}$
```

In this extension of the TripleSieve, the third search over the list is only performed if the first two vectors are sufficiently close. Asymptotically this means that still the same number of good triples is found, but this significantly reduces the cost of the algorithm: after all, out of all pairs of vectors, only those with pairwise angle less than $\theta_1 = \arccos(1/3)$ survive. The fraction of pairs that survive is proportional to $(\sin \theta_1)^n$ or, after a trigonometric exercise, a fraction $p = (2\sqrt{2}/3)^n$ of all pairs of vectors survive the first round. The time cost of the algorithm is then $N^2(1 + p \cdot N)$ which, as N is much larger than $1/p$, leads to the following result.

Proposition 4.1. *Under the aforementioned heuristic assumptions, the Filtered Triple Sieve solves SVP in time* $(27/16)^{3n/4} \cdot (2\sqrt{2}/3)^n = 2^{0.4812n}$ *and space* $(27/16)^{n/4} = 2^{0.1887n}$.

Note that such a filtering strategy may also be applied to the k-TupleSieve with larger k.

5 Tuple MinkowskiSieve

In the previous sections, we described the tuple variants of the ListSieve algorithm. In practice, it has been observed that the GaussSieve algorithm of Micciancio and Voulgaris [23] is more efficient (in terms of both time and space) than the ListSieve algorithm. Hence, for practical considerations, we devise a tuple-like GaussSieve algorithm.

5.1 GaussSieve

We recall the GaussSieve algorithm of Micciancio and Voulgaris [23]. The philosophy of the algorithm is to make every pairs of distinct vectors in a list Gauss-reduced.

In GaussSieve, we first setup and maintain a list L of vectors and a stack S of vectors in the algorithm. They are initially empty. To begin the GaussSieve algorithm, we sample a new vector $\boldsymbol{p}$. We add $\boldsymbol{p}$ to the list L. The list L consists of lattice vectors that are always pairwise Gauss-reduced and this property is maintained during the execution of the algorithm. For each new sampled vector $\boldsymbol{p}$, we may modify $\boldsymbol{p}$ (to $\boldsymbol{p}'$) and the existing vectors in the list L (to L$'$) so that $\mathrm{L}' \cup \boldsymbol{p}'$ is pairwise Gauss-reduced. Therefore, for each $\boldsymbol{p}$, we not only reduce $\boldsymbol{p}$ (to $\boldsymbol{p}'$) using the vectors $\boldsymbol{v} \in \mathrm{L}$; but also reduce the vectors $\boldsymbol{v} \in \mathrm{L}$ using $\boldsymbol{p}'$. If $\boldsymbol{p}$ (or $\boldsymbol{p}'$) equals to some $\boldsymbol{v} \in \mathrm{L}$, then we discard it and count it as a collision of zero vectors. If some list vector $\boldsymbol{v} \in \mathrm{L}$ has been reduced by $\boldsymbol{p}$, the list itself may not be pairwise Gauss-reduced anymore (since $\boldsymbol{v}$ might be modified). We then move the modified vector $\boldsymbol{v}'$ to the stack S and attempt to reduce it in the following reductions (taking $\boldsymbol{v}'$ as a new sample $\boldsymbol{p}$).

Note that, in GaussSieve, we are not Gauss-reducing the two vectors in a one-off way. Instead, we only reduce one vector at one iteration and then put the modified vector in the stack (if it is a vector in the list), and perhaps change it in the future.

In practice, we can terminate the algorithm if we have found a short enough vector; or if the number of collisions of zero vectors reach some bound.

5.2 Tuple MinkowskiSieve

The Gauss-reduced condition for two vectors can be generalized to the Minkowski-reduced condition for tuple vectors (see Definition 1). The greedy reduction algorithm of [24, 27] can be used to efficiently compute Minkowski-reduced bases for lattices of dimensions ≤ 4.

We describe the Triple MinkowskiSieve algorithm in Algorithms 5 and 6. The Quadruple MinkowskiSieve algorithm can be designed similarly.

Lines 1-5 of Algorithm 6 ensure the Gauss-reduced condition; Lines 6-10 of Algorithm 6 ensure the 3-dimensional Minkowski reducedness. Note that the Triple MinkowskiSieve resembles the GaussSieve. In particular, we do not Minkowski-reduce the three vectors in one go. Instead, we only modify one vector at each reduction step.

[3]After each pass of reducing $\boldsymbol{p}$ by all $\boldsymbol{v} \in \mathrm{L}$ (resp. by all pairs $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathrm{L}$), we repeat the procedure as the current $\boldsymbol{p}$ may not be reduced w.r.t some $\boldsymbol{v}$ (resp. some pair of $\boldsymbol{v}_1, \boldsymbol{v}_2$) in the list L anymore. Note that the vector $\boldsymbol{p}$ is being updated from every reduction by $\boldsymbol{v}$ (resp. pairs of $\boldsymbol{v}_1, \boldsymbol{v}_2$). We repeat the loop (Lines 1 and 6 of Algorithm 6) until $\boldsymbol{p}$ can not be reduced anymore with any $\boldsymbol{v} \in \mathrm{L}$ (resp. any pair $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathrm{L}$).

Algorithm 5 TripleMinkowskiSieve($(\boldsymbol{b}_i)_i$)

```
1: $L \leftarrow \{\mathbf{0}\}$, $S \leftarrow \{\}$
2: while cond do
3:     if S is not empty then
4:         $\boldsymbol{p} \leftarrow$ S.pop()
5:     else
6:         $\boldsymbol{p} \leftarrow$ SampleGaussian($(\boldsymbol{b}_i)_i$)
7:     $\boldsymbol{p} \leftarrow$ TripleReduce($\boldsymbol{p}$, L, S)
8:     if $\boldsymbol{p} \neq \mathbf{0}$ then insert $\boldsymbol{p}$ to L
```

Algorithm 6 TripleReduce(p, L, S)

```
1: Loop$^4$ {$\forall \boldsymbol{v} \in$ L, reduce $\boldsymbol{p}$ by $\boldsymbol{v}$}
2: if $\boldsymbol{p} = \mathbf{0}$ then Return $\boldsymbol{p}$
3: $\forall \boldsymbol{v} \in$ L, reduce $\boldsymbol{v}$ by $\boldsymbol{p}$
4: if any $\boldsymbol{v} \in$ L is modified then
5:     move $\boldsymbol{v}$ to S
6: Loop$^4$ {$\forall \boldsymbol{v}_1, \boldsymbol{v}_2 \in$ L, reduce $\boldsymbol{p}$ by $\boldsymbol{v}_1, \boldsymbol{v}_2$}
7: if $\boldsymbol{p} = \mathbf{0}$ then Return $\boldsymbol{p}$
8: $\forall \boldsymbol{v}_1, \boldsymbol{v}_2 \in$ L, reduce $\boldsymbol{v}_1$ by $\boldsymbol{p}$, $\boldsymbol{v}_2$
9: if any $\boldsymbol{v}_1 \in$ L is modified then
10:     move $\boldsymbol{v}_1$ to S
11: Return $\boldsymbol{p}$
```

If the modified vector comes from a new sampled vector $\boldsymbol{p}$, then we repeatedly reduce $\boldsymbol{p}$ using every $\boldsymbol{v}_1, \boldsymbol{v}_2 \in$ L (Line 6 of Algorithm 6) until it can not be reduced anymore. If the modified vector comes from an existing list vector $\boldsymbol{v} \in$ L, then we move the reduced $\boldsymbol{v}'$ from the list L to the stack S for further consideration (Lines 9-10 of Algorithm 6).

During the execution of the Algorithm 5, every triple of vectors of L is always Minkowski-reduced (except during the calls to Algorithm 6). We can terminate the algorithm if we have found a short enough vector; or if the number of zero vectors returned by Algorithm 6) reaches some threshold (this determines the "cond" in Line 2 of Algorithm 5).

As GaussSieve is one of the most promising lattice sieving candidates for use in practice, we implemented the Triple and Quadruple MinkowskiSieve algorithms. We also consider an improved variant of Triple MinkowskiSieve by applying the filtering principle (see Section 4) on pairs of vectors ($\boldsymbol{p}$, $\boldsymbol{v}_1$). We refer to Subsection 6.2 for experimental results.

6 Experiments

In this section, we describe some experimental results which support our previous analysis. In Subsection 6.1, we conduct experiments to verify the conclusions in Subsections 3.1 and 3.2. In Subsection 6.2, we describe the implementation and experimental results for Triple, Quadruple and Filtered Triple MinkowskiSieve.

6.1 Experiments on the unit sphere

We give some numerical evidence for the conclusions in Section 3, in the case of list vectors sampled uniformly on the unit sphere. Note that, even in that idealized framework, the analysis of Section 3 remains heuristic when $k \geq 3$.

We sample random vectors uniformly on the sphere, and, for each new vector, we test whether it coincides with (e.g., is close to) a combination of two vectors (resp. one vector) already present in the list, in the case of triples (resp. pairs).

We enforce Minkowski's reduction condition for the double- or triple-reduced vectors, thus this is even stricter than the cost analysis. Once the number of "collisions" (nearby vectors) exceeds a small multiple of $|\mathrm{L}|$ (here we use $4 \cdot |\mathrm{L}|$) we stop and record the list size. We repeat this process $\lceil 20000/n \rfloor$ many times for each dimension n, and take the average (logarithmic) list size and plot it in Figure 2.

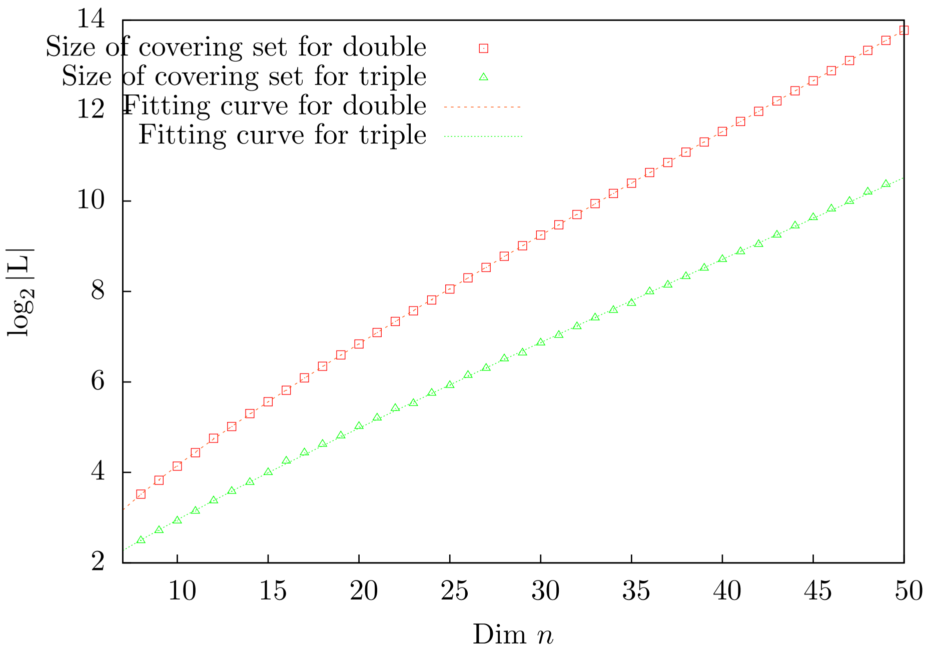

Figure 2: Log space complexity of Double and Triple Sieve.

The lines in the plot take the form $C_1 n + b\,(\log_2 n)^{e_1} + c$ and $C_2 n + d\,(\log_2 n)^{e_2} + e$. We used a least-squares fit to find the best fits for parameters b, c, d, e. For the experimental data, the values C_1 and C_2 are 0.21 and 0.17, which are close to the heuristic estimates. It is also clear that there is a significant gap between pairwise and triplewise reduced list sizes.

6.2 Implementation of the Tuple MinkowskiSieve

We have implemented GaussSieve, Triple MinkowskiSieve, Quadruple MinkowskiSieve and filtered Triple MinkowskiSieve algorithms. These algorithms were described in Section 5. The implementation can be found in FPLLL [8]. We describe some experimental results for these algorithms.

For each dimension, we use at least 10 random instances from the SVP challenge generator [2]. Before the sieving, we pre-process the input basis with lattice reduction. To prevent trivial instances (where the sieving quickly finds many collisions since the vectors are already short), we used either LLL with $\delta = 0.75, \eta = 0.51$ for smaller dimensions, and BKZ with blocksize 20 for larger dimensions. The computation are taken on Intel Xeon X5650 processors of 2.67GHz.

We give experimental results in Table 2 and Figures 3, 4 (see Appendix). We explain the notation in Table 2 (Figures 3, 4 follow a similar notation). The column n denotes the dimension. Column "**2-red**" denotes GaussSieve; Column "**3-red**" denotes Triple MinkowskiSieve; "**4-red**" denotes Quadruple MinkowskiSieve; Column "**3-red′**" is the filtered Triple Minkowski-

Sieve. Under each algorithm, the subcolumn $|\mathrm{L}|_\infty$ denotes the (average) maximum list size during the sieving. The number inside parentheses (e.g., (.251) in $n = 24$ and **2-red**) is the average value of $(\log|\mathrm{L}|_\infty)/n$ over all experiments. Subcolumn *time* is the average running-time in *seconds*. The cells with "ь" are the experiments with BKZ-20 pre-processed; other cells are LLL pre-processed.

It can be seen that there is a noticeable gap between the space requirements of GaussSieve and Triple MinkowskiSieve (resp. Quadruple MinkowskiSieve). Furthermore, the filtered Triple MinkowskiSieve variant is much more efficient than the non-filtered Triple Minkowski-Sieve.

7 Discussion

In the present work, we describe various tuple sieving algorithms which reduce the memory requirement in lattice sieving. For triple (quadruple) reduced lists, we estimate the space complexity to be $2^{0.1887n+o(n)}$ (resp. $2^{0.1724n+o(n)}$). We investigated the (heuristic) asymptotic costs of these algorithms, and verified these theoretical results through experiments.

One interesting future question is to consider nearest neighbor search techniques to speed up the search procedures, which could lead to an improvement on the overall running-time of the sieving algorithms. For triple sieving, for instance, we could consider a nest of hash functions, where the first search is to find vectors with inner product at least 1/3 and the second search is to find vectors with inner product 1/2. The two sets of hash tables $\mathcal{T}_{1/3}, \mathcal{T}_{1/2}$ are therefore different; one is optimized for finding vectors at angle ≈ 70 degrees and one for 60 degrees. Figuring out how to optimize the combination of nearest neighbor searching with these nested search procedures is left for future work.

Very recently, following up on our work, Herold and Kirshanova [16] further studied tuple lattice sieving algorithms, and how the 'filtering' procedure can be optimized to obtain even better time complexities. For triple reductions, they obtain a heuristic time complexity of only $2^{0.3962n+o(n)}$, significantly improving upon our $2^{0.4812n+o(n)}$ using filtered triple sieving. In practice, this comes at the cost of having to use the Nguyen-Vidick sieve (rather than the faster GaussSieve) to obtain these asymptotic costs. For more details on this result, as well as their improved complexities for tuple sieving with larger tuple sizes, we refer the reader to [16].

Acknowledgments

We thank the PSMN (Pôle Scientifique de Modélisation Numérique, Lyon, France) for providing computing facilities. The work of the first and third authors has been supported by an ERC Starting Grant ERC-2013-StG-335086-LATTAC. The work of the second author has been supported by the SNSF ERC Transfer Grant CRETP2-166734 FELICITY.

References

[1] Inverse Symbolic Calculator. Available at `https://isc.carma.newcastle.edu.au/index`.

[2] SVP challenge. Available at `http://latticechallenge.org/svp-challenge`.

[3] D. Aggarwal, D. Dadush, O. Regev, and N. Stephens-Davidowitz. Solving the shortest vector problem in 2^n time using discrete Gaussian sampling. In *Proc. of STOC*, pages 733–742. ACM, 2015.

[4] M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proc. of STOC*, pages 601–610. ACM, 2001.

[5] A. Becker, L. Ducas, N. Gama, and T. Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proc. of SODA*, pages 10–24. SIAM, 2016.

[6] Y. Chen and P. Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *Proc. of ASIACRYPT*, volume 7073 of *LNCS*, pages 1–20. Springer, 2011.

[7] The Sage Developers. *Sage Mathematics Software (Version 6.8)*, 2015. `http://www.sagemath.org`.

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

[9] R. Fitzpatrick, C. Bischof, J. Buchmann, Ö. Dagdelen, F. Göpfert, A. Mariano, and B.-Y. Yang. Tuning GaussSieve for speed. In *Proc. of LATINCRYPT*, volume 9230 of *LNCS*, pages 288–305. Springer, 2015.

[10] P. Fouque and P. Kirchner. Time-memory trade-off for lattice enumeration in a ball. *IACR Cryptology ePrint Archive*, 2016:222, 2016.

[11] N. Gama, P. Q. Nguyen, and O. Regev. Lattice enumeration using extreme pruning. In *Proc. of EUROCRYPT*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010.

[12] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proc. of STOC*, pages 197–206. ACM, 2008.

[13] G. Hanrot, X. Pujol, and D. Stehlé. Algorithms for the shortest and closest lattice vector problems. In *IWCC*, volume 6639 of *LNCS*, pages 159–190. Springer, 2011.

[14] G. Hanrot and D. Stehlé. Improved analysis of Kannan's shortest lattice vector algorithm. In *Proceedings of CRYPTO*, volume 4622 of *LNCS*, pages 170–186. Springer, 2007.

[15] G. Hanrot and D. Stehlé. Worst-case Hermite-Korkine-Zolotarev reduced lattice bases. *CoRR*, abs/0801.3331, 2008.

[16] G. Herold and E. Kirshanova. Improved algorithms for the approximate k-list problem in Euclidean norm. *Preprint*, 2016.

[17] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: a ring based public key cryptosystem. In *Proc. of ANTS*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.

[18] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proc. of STOC*, pages 99–108. ACM, 1983.

[19] T. Kleinjung. *Private communication*, 2015.

[20] T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *Proc. of CRYPTO*, volume 9215 of *LNCS*, pages 3–22. Springer, 2015.

[21] T. Laarhoven, M. Mosca, and J. van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2-3):375–400, 2015.

[22] D. Micciancio and O. Regev. Lattice-based cryptography. In *Post-Quantum Cryptography, D. J. Bernstein, J. Buchmann, E. Dahmen (Eds)*, pages 147–191. Springer, 2009.

[23] D. Micciancio and P. Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proc. of SODA*. ACM, 2010.

[24] P. Q. Nguyen and D. Stehlé. Low-dimensional lattice basis reduction revisited. *ACM Transactions on Algorithms*, 5(4), 2009. Article 46.

[25] P. Q. Nguyen and T. Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2), 2008.

[26] X. Pujol and D. Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605, 2009. `http://eprint.iacr.org/2009/605`.

[27] I. Semaev. A 3-dimensional lattice reduction algorithm. In *Proc. of CALC*, volume 2146 of *LNCS*, pages 181–193. Springer, 2001.

[28] P. P. Tammela. On the reduction theory of positive quadratic forms. *Soviet Mathematics Doklady*, 14:651–655, 1973.

[29] Wolfram Research, Inc. Mathematica (version 10.3). Champaign, IL, 2015.

Appendix

The appendix contains experimental results from Subsection 6.2.

	2-red		**3-red**		**3-red′**		**4-red**	
n	$\lvert\mathrm{L}\rvert_\infty$	time	$\lvert\mathrm{L}\rvert_\infty$	time	$\lvert\mathrm{L}\rvert_\infty$	time	$\lvert\mathrm{L}\rvert_\infty$	time
24	70 (.251)	.02	42 (.222)	.11	50 (.233)	.03	33 (.208)	1.1e1
26	89 (.238)	.04	49 (.209)	.21	57 (.217)	.04	34 (.190)	2.5e1
28	130 (.246)	.05	74 (.222)	.41	83 (.227)	.07	55 (.206)	6.8e1
30	165 (.238)	.08	81 (.207)	.95	94 (.214)	.13	58 (.192)	1.5e2
32	244 (.248)	.11	97 (.200)	1.8	123 (.213)	.22	62 (.183)	3.2e2
34	319 (.245)	.19	148 (.212)	4.8	169 (.217)	.48	97 (.194)	9.6e2
36	435 (.243)	.31	176 (.207)	9.9	223 (.217)	.89	118 (.191)	2.6e3
38	571 (.241)	.61	230 (.206)	2.7e1	284 (.214)	1.9	134 (.186)	6.5e3
40	741 (.238)	1.1	298 (.205)	6.3e1	361 (.212)	4.4	191 (.189)	1.9e4
42	1021 (.238)	2.3	377 (.204)	1.6e2	476 (.212)	9.4	♭ 246 (.189)	2.0e4
44	1390 (.237)	5.1	484 (.203)	4.0e2	602 (.219)	2.3e1	♭ 263 (.182)	5.2e4
46	1777 (.235)	9.6	638 (.202)	9.9e2	777 (.209)	4.7e1	♭ 352 (.184)	1.4e5
48	2400 (.234)	2.0e1	795 (.201)	2.5e3	1063 (.209)	1.1e2	—	—
50	3254 (.233)	4.1e1	1104 (.202)	6.6e3	1328 (.207)	2.5e2	—	—
52	4219 (.232)	9.5e1	1324 (.199)	1.7e4	1742 (.207)	5.6e2	—	—
54	5879 (.232)	2.2e2	1700 (.199)	3.9e4	2234 (.206)	1.2e3	—	—
56	7574 (.230)	5.3e2	♭ 2163 (.198)	2.7e4	♭ 2878 (.205)	8.5e2	—	—
58	10539 (.230)	1.2e3	♭ 2877 (.198)	7.1e4	♭ 3804 (.205)	1.9e3	—	—
60	♭ 13433 (.229)	9.1e2	♭ 3664 (.197)	1.7e5	♭ 4879 (.204)	4.3e3	—	—
62	♭ 18251 (.228)	2.0e3	♭ 5102 (.199)	4.6e5	♭ 6475 (.204)	1.0e4	—	—
64	♭ 24223 (.228)	4.5e3	♭ 6604 (.198)	1.1e6	♭ 8338 (.204)	2.2e4	—	—
66	♭ 32587 (.227)	9.1e3	—	—	♭ 10994 (.203)	5.1e4	—	—
68	♭ 43887 (.227)	2.0e4	—	—	♭ 14297 (.203)	1.1e5	—	—
70	♭ 58912 (.227)	3.8e4	—	—	♭ 18973 (.203)	2.5e5	—	—
72	♭ 79521 (.226)	7.2e4	—	—	—	—	—	—
74	♭ 107050 (.226)	1.5e5	—	—	—	—	—	—
76	♭ 142504 (.225)	2.9e5	—	—	—	—	—	—
78	♭ 192141 (.225)	6.0e5	—	—	—	—	—	—
80	♭ 256343 (.225)	1.2e6	—	—	—	—	—	—

Table 2: Experimental results for (filtered) Tuple MinkowskiSieve (Subsection 6.2).

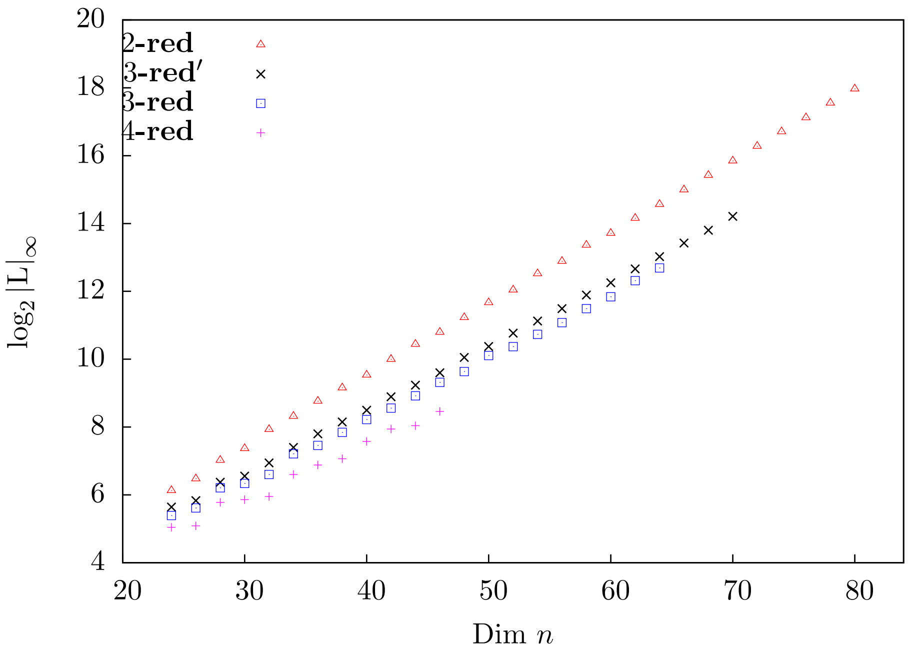

Figure 3: Plot of logarithmic average maximum list size ($\log_2 |\mathrm{L}|_\infty$) for Table 2.

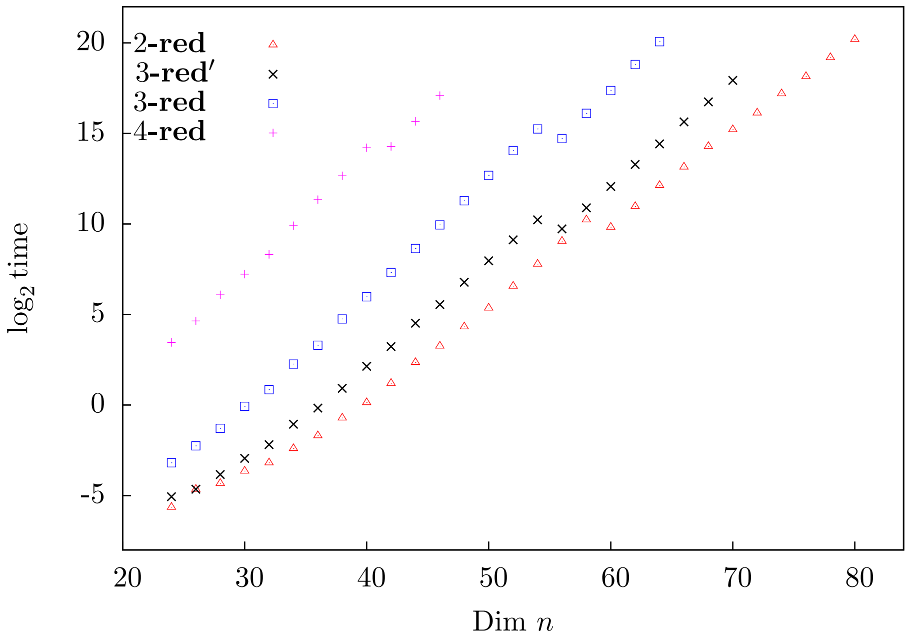

Figure 4: Plot of logarithmic average time ($\log_2$ time) for Table 2. Note that we used either LLL or BKZ-20 for the pre-processing depending on the dimension of the input instance.