Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing

Thijs Laarhoven and Benne de Weger

Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
mail@thijs.com, b.m.m.d.weger@tue.nl

Abstract. Recently, it was shown that angular locality-sensitive hashing (LSH) can be used to significantly speed up lattice sieving, leading to a heuristic time complexity for solving the shortest vector problem (SVP) of $2^{0.337n+o(n)}$ (and space complexity $2^{0.208n+o(n)}$. We study the possibility of applying other LSH methods to sieving, and show that with the spherical LSH method of Andoni et al. we can heuristically solve SVP in time $2^{0.298n+o(n)}$ and space $2^{0.208n+o(n)}$. We further show that a practical variant of the resulting SphereSieve is very similar to Wang et al.'s two-level sieve, with the key difference that we impose an order on the outer list of centers.

Keywords: shortest vector problem (SVP), sieving algorithms, nearest neighbor problem, locality-sensitive hashing (LSH), lattice cryptography

1 Introduction

Lattice cryptography. Lattice-based cryptography has recently received wide attention from the cryptographic community, due to e.g. its presumed resistance against quantum attacks [10], the existence of lattice-based fully homomorphic encryption schemes [18], and efficient cryptographic primitives like NTRU [20] and LWE [40]. An important problem in the study of lattices is the shortest vector problem (SVP): given a lattice, find a shortest non-zero lattice vector. Although SVP is well-known to be NP-hard under randomized reductions [2, 30], the computational complexity of finding short(est) vectors is still not well understood, even though it is crucial for applications in lattice-based cryptography [26,38].

Finding shortest vectors. Currently the four main methodologies for solving SVP are enumeration [15,23,37], sieving [3], constructing the Voronoi cell of the lattice [31], and a recent method based on discrete Gaussian sampling [1]. Enumeration has a low space complexity, but a time complexity superexponential in the dimension n, which is suboptimal as the

other methods all run in single exponential ($2^{\Theta(n)}$) time. Drawbacks of the latter methods are that their space complexities are $2^{\Theta(n)}$ as well, and that the hidden constants in the exponents are relatively big. As a result, enumeration (with extreme pruning [17]) is commonly still considered the most practical method for finding shortest vectors in high dimensions [33].

Sieving algorithms. On the other hand, these other SVP methods are less explored than enumeration, and recent improvements in sieving have considerably narrowed the gap with enumeration. Following the groundbreaking work of Ajtai et al. [3], it was later shown that with sieving one can provably solve SVP in arbitrary lattices in time $2^{2.465n+o(n)}$ [19, 35, 39]. Heuristic analyses further suggest that with sieving one can solve SVP in time $2^{0.415n+o(n)}$ and space $2^{0.208n+o(n)}$ [7,32,35], or optimizing for time, in time $2^{0.378n+o(n)}$ and space $2^{0.293n+o(n)}$ [7, 46, 47]. Various papers further studied how to speed up sieving in practice [11, 16, 22, 25, 27, 28, 34, 41, 42], and currently the highest dimension in which sieving was used to solve SVP is 116 for arbitrary lattices [43], and 128 for ideal lattices [11, 22, 36].

Locality-sensitive hashing. Since sieving algorithms store long lists of high-dimensional vectors in memory, and the main procedure of sieving is to go through this list to find vectors nearby a target vector, one might ask whether this can be done faster than with a linear search. This problem is related to the nearest neighbor problem [21], and a well-known method for solving this problem faster is based on locality-sensitive hashing (LSH). Recently, it was shown that the efficient angular LSH technique of Charikar [12] can be used to significantly speed up sieving, both in theory and in practice [24, 29], with heuristic time and space complexities of $2^{0.337n+o(n)}$ and $2^{0.208n+o(n)}$ respectively [24]. An open problem of [24] was whether using other LSH techniques would lead to even better results.

Contributions. In this work we answer the latter question in the affirmative. With spherical LSH [5, 6] we obtain heuristic time and space complexities for solving SVP of $2^{0.2972n+o(n)}$ and $2^{0.2075n+o(n)}$ respectively, achieving the best asymptotic time complexity for SVP to date. We obtain the space/time trade-off depicted in Figure 1, and show how the trade-off can be turned into a clean speed-up leading to the blue point in Figure 1. We further show that a practical variant of our algorithm appears to be very similar to the two-level sieve of Wang et al. [46], with the key difference that the outer list of centers is ordered.

Outline. In Section 2 we first provide some background on (spherical) LSH. Section 3 describes how to apply spherical LSH to the NV-sieve [35],

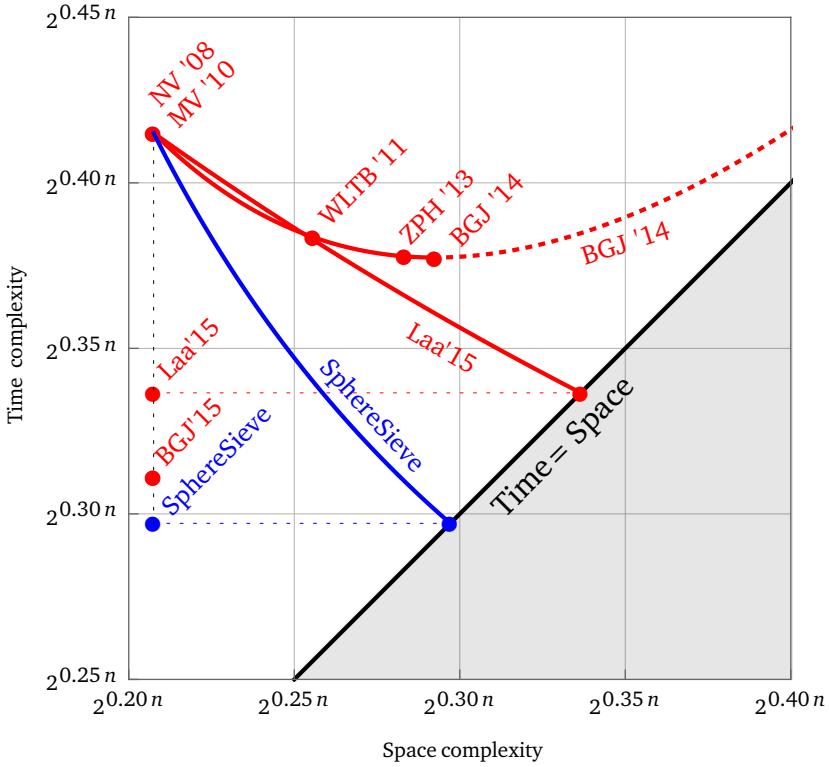

Fig. 1. The space/time trade-offs of various heuristic sieve algorithms from the literature (red), the trade-off for spherical LSH (blue, cf. Proposition 1), and the speedup when using hash tables sequentially rather than in parallel (the point at $(2^{0.208n}, 2^{0.298n})$, cf. Theorem 1). The referenced papers are: NV'08: [35], MV'10: [32], WLTB'11: [46], ZPH'13: [47], BGJ'14: [7], Laa'15: [24], BGJ'15: [8].

and Section 4 states the main result. In Section 5 we describe a practical variant of our algorithm, and we discuss its relation with Wang et al.'s two-level sieve [46]. In Section 6 we discuss practical implications of our results, and remaining open problems for future work.

2 Locality-sensitive hashing

2.1 Locality-sensitive hash families

The nearest neighbor problem is the following [21]: Given a list of n-dimensional vectors of cardinality N, e.g., $L = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_N\} \subset \mathbb{R}^n$, preprocess L in such a way that given a target vector $\boldsymbol{v} \notin L$, we can efficiently find an element $\boldsymbol{w} \in L$ closest to $\boldsymbol{v}$. A common variant of this

problem is the approximate nearest neighbor problem, where an acceptable solution is a vector "nearby" the target vector (and a solution is unacceptable if it is "far away"). While for low dimensions n there exist ways to answer these queries in time sub-linear or even logarithmic in the list size N, for high dimensions it generally seems hard to do better than with a naive brute-force list search of time $O(N)$. This inability to efficiently store and query lists of high-dimensional data is sometimes referred to as the "curse of dimensionality" [21].

Fortunately, if we know that the list L has a certain structure, or if there is a significant gap between what is meant by "nearby" and "far away," then there are ways to preprocess L such that queries can be answered in time sub-linear in N. One of the most well-known methods for this is locality-sensitive hashing (LSH), introduced by Indyk and Motwani [21]. Locality-sensitive hash functions are functions h which map n-dimensional vectors $\boldsymbol{w}$ to low-dimensional *sketches* $h(\boldsymbol{w})$, such that vectors which are nearby in $\mathbb{R}^n$ have a high probability of having the same sketch and vectors which are far apart have a low probability of having the same image under h. Formalizing this property leads to the following definition of a locality-sensitive hash family $\mathcal{H}$. Here D is a similarity measure[1] on $\mathbb{R}^n$, and U is commonly a finite subset of $\mathbb{N}$.

Definition 1. *[21] A family $\mathcal{H} = \{h : \mathbb{R}^n \to U\}$ is called (r_1, r_2, p_1, p_2)-sensitive for similarity measure D if for any $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$:*

- *if $D(\boldsymbol{v}, \boldsymbol{w}) < r_1$ then $\mathbb{P}_{h\in\mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] \geq p_1$;*
- *if $D(\boldsymbol{v}, \boldsymbol{w}) > r_2$ then $\mathbb{P}_{h\in\mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] \leq p_2$.*

Note that if there exists an LSH family $\mathcal{H}$ which is (r_1, r_2, p_1, p_2)-sensitive with $p_1 \gg p_2$, then (without computing $D(\boldsymbol{v}, \cdot)$) we can use $\mathcal{H}$ to distinguish between vectors which are at most r_1 away from $\boldsymbol{v}$, and vectors which are at least r_2 away from $\boldsymbol{v}$ with non-negligible probability.

2.2 Amplification

In general it is not known whether efficiently computable (r_1, r_2, p_1, p_2)-sensitive hash families even exist for the ideal setting of $r_1 \approx r_2$ and $p_1 \approx 1$ and $p_2 \approx 0$. Instead, one commonly first constructs an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$ with $p_1 \approx p_2$, and then uses several AND- and OR-compositions to turn it into an (r_1, r_2, p_1', p_2')-sensitive hash family $\mathcal{H}'$ with $p_2' < p_2 < p_1 < p_1'$, thereby amplifying the gap between p_1 and p_2.

[1] A similarity measure D may informally be thought of as a "slightly relaxed" metric, which may not satisfy all properties associated to metrics; see e.g. [21] for details.

AND-composition. Given an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$, we can construct an (r_1, r_2, p_1^k, p_2^k)-sensitive hash family $\mathcal{H}'$ by taking a bijective function $\alpha : U^k \to U$ and k functions $h_1, \ldots, h_k \in \mathcal{H}$ and defining $h \in \mathcal{H}'$ as $h(\boldsymbol{v}) = \alpha(h_1(\boldsymbol{v}), \ldots, h_k(\boldsymbol{v}))$. This increases the relative gap between p_1 and p_2 but decreases their absolute values.

OR-composition. Given an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$, we can construct an $(r_1, r_2, 1-(1-p_1)^t, 1-(1-p_2)^t)$-sensitive hash family $\mathcal{H}'$ by taking $h_1, \ldots, h_t \in \mathcal{H}$, and defining $h \in \mathcal{H}'$ by the relation $h(\boldsymbol{v}) = h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) = h_i(\boldsymbol{w})$ for some $i \in \{1, \ldots, t\}$. This compensates the decrease of the absolute values of the probabilities.

Combining a k-wise AND- with a t-wise OR-composition, we can turn an (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$ into an (r_1, r_2, p_1^*, p_2^*)-sensitive hash family $\mathcal{H}'$ with $p^* \stackrel{\text{def}}{=} 1-(1-p^k)^t$ for $p = p_1, p_2$. Note that for $p_1 > p_2$ we can always find values k and t such that $p_1^* \approx 1$ and $p_2^* \approx 0$.

2.3 Finding nearest neighbors

To use these hash families to find nearest neighbors, we can use the following method first described in [21]. First, choose $t \cdot k$ random hash functions $h_{i,j} \in \mathcal{H}$, and use the AND-composition to combine k of them at a time to build t different hash functions $h_1, \ldots, h_t$. Then, given the list L, build t different hash tables $T_1, \ldots, T_t$, where for each hash table T_i we insert $\boldsymbol{w}$ into the bucket labeled $h_i(\boldsymbol{w})$. Finally, given the target vector $\boldsymbol{v}$, compute its t images $h_i(\boldsymbol{v})$, gather all the candidate vectors that collide with $\boldsymbol{v}$ in at least one of these hash tables (an OR-composition), and search this list of candidates for the nearest neighbor.

Clearly, the quality of this algorithm for finding nearest neighbors depends on the quality of the underlying hash family and on the parameters k and t. Larger k and t amplify the gap between the probabilities of finding nearby and faraway vectors as candidates, but this comes at the cost of having to compute many hashes (both during the preprocessing phase and in the querying phase) and having to store many hash tables, each containing all vectors from L. The following lemma shows how to balance k and t such that the overall query time complexity of finding near(est) neighbors is minimized.

Lemma 1. *[21] Suppose there exists a (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$. Then, with*

$$\rho = \frac{\log(1/p_1)}{\log(1/p_2)}, \qquad k = \frac{\log(N)}{\log(1/p_2)}, \qquad t = O(N^\rho), \tag{1}$$

with high probability we can find an element $\boldsymbol{w}^* \in L$ *with* $D(\boldsymbol{v}, \boldsymbol{w}^*) \leq r_2$ *or (correctly) conclude that no element* $\boldsymbol{w}^* \in L$ *with* $D(\boldsymbol{v}, \boldsymbol{w}^*) \leq r_1$ *exists, with the following costs:*

1. *Time for preprocessing the list:* $O(kN^{1+\rho})$.
2. *Space complexity of the preprocessed data:* $O(N^{1+\rho})$.
3. *Time for answering a query* $\boldsymbol{v}$*:* $O(N^{\rho})$.
   (a) *Hash evaluations of the query vector* $\boldsymbol{v}$*:* $O(N^{\rho})$.
   (b) *Candidates to compare to the query vector* $\boldsymbol{v}$*:* $O(N^{\rho})$.

Although Lemma 1 only shows how to choose k and t to minimize the time complexity, we can generally tune k and t to use slightly more time and less space. In a sense this algorithm can be seen as a generalization of the naive brute-force search method, as $k = 0$ and $t = 1$ corresponds to checking the whole list in linear time with linear space. Note that the main costs of the algorithm are determined by the value of ρ, which is therefore often considered the central parameter of interest in LSH literature. The goal is to design $\mathcal{H}$ so that ρ is as small as possible.

2.4 Spherical locality-sensitive hashing

In [24] the family of hash functions that was considered was Charikar's cosine hash family [12] based on angular distances. In the same paper it was suggested that other hash families, such as Andoni and Indyk's celebrated Euclidean LSH family [4], may lead to even better results. The latter method however does not seem to work well in the context of sieving[2], and instead we will focus on yet another LSH family, recently proposed by Andoni et al. [5, 6] and coined spherical LSH.

Hash family. In spherical LSH, we assume[3] that all points in the data set L lie on the surface of a hypersphere $S^{n-1}(R) = \{\boldsymbol{v} \in \mathbb{R}^n : \|\boldsymbol{x}\| = R\}$. In the following description of the hash family we further assume that all vectors lie on $S^{n-1}(1)$, although these definitions can trivially be generalized to the general case $S^{n-1}(R)$.

[2] Technically speaking, [4] uses the Johnson-Lindenstrauss lemma to project n- to n_0-dimensional vectors with $n_0 = o(n)$, so that single-exponential costs in n_0 ($2^{\Theta(n_0)}$) are sub-exponential in n ($2^{o(n)}$). However, this projection only preserves inter-point distances up to small errors if the length of the list is sufficiently small ($N = 2^{o(n)}$), which is not the case in sieving. Moreover, we estimated the potential improvement using Euclidean LSH to be smaller than the improvement we obtain here.

[3] In Section 3 we will justify why this assumption makes sense in sieving.

First, we sample $U = 2^{\Theta(\sqrt{n})}$ vectors $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_U \in \mathbb{R}^n$ from an n-dimensional Gaussian distribution with average norm $\mathbb{E}\|\boldsymbol{s}_i\| = 1$.[4] This equivalently corresponds to drawing each vector entry from a univariate Gaussian distribution $\mathcal{N}(0, \frac{1}{n})$. To each $\boldsymbol{s}_i$ we associate a hash region H_i:

$$H_i = \{\boldsymbol{v} \in S^{n-1}(1) : \langle \boldsymbol{v}, \boldsymbol{s}_i \rangle \geq n^{-1/4}\} \setminus \bigcup_{j=1}^{i-1} H_j. \qquad (i = 1, \ldots, U) \quad (2)$$

Since we assume that $\boldsymbol{v} \in S^{n-1}(1)$ and w.h.p. we have $\|\boldsymbol{s}_i\| \approx 1$, the condition $\langle \boldsymbol{v}, \boldsymbol{s}_i \rangle \geq n^{-1/4}$ is equivalent to $\|\boldsymbol{v} - \boldsymbol{s}_i\| \leq \sqrt{2} - \Theta(n^{-1/4})$, i.e., $\boldsymbol{v}$ lies in the *almost-hemisphere* of radius $\sqrt{2} - \Theta(n^{-1/4})$ defined by $\boldsymbol{s}_i$.

Note that the parts of $S^{n-1}(1)$ that are covered by multiple hash regions are assigned to the *first* region H_i that covers the point. As a result, the size of hash regions generally decreases with i. Also note that the choice of $U = 2^{\Theta(\sqrt{n})}$ guarantees that with high probability, at the end the entire sphere is covered by these hash regions $H_1, H_2, \ldots, H_U$; informally, each hash region covers a $2^{-\Theta(\sqrt{n})}$ fraction of the sphere, so we need $2^{\Theta(\sqrt{n})}$ regions to cover the entire hypersphere. Finally, taking $U = 2^{\Theta(\sqrt{n})}$ also guarantees that computing hashes can trivially be done in $2^{\Theta(\sqrt{n})} = 2^{o(n)}$ time by going through each of the hash regions $H_1, H_2, \ldots, H_U$ and checking whether it contains a given point $\boldsymbol{v}$.

In our analysis we will use the following result, which is implicitly stated in [5, Lemma 3.3] and [6, Appendix B.1]. Note that in the application of sieving later on, vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ are not assumed to lie on the surface of a sphere, but inside a thin spherical shell with some inner radius γR and outer radius R, with $\gamma = 1 - o(1)$. We can however still apply spherical hashing, due to the observation that $\|\frac{\boldsymbol{v}}{R} - \frac{\boldsymbol{w}}{R}\| - \|\frac{\boldsymbol{v}}{\|\boldsymbol{v}\|} - \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\| = O(1-\gamma) = o(1)$. In other words, by applying the hash method to normalized vectors $\tilde{\boldsymbol{x}} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}$ which all do lie on a hypersphere, the inter-point distances are preserved up to a negligible additive term $o(1)$, which translates to an $o(1)$ term in the application of LSH.

Lemma 2. *Let $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ with $\|\boldsymbol{v}\|, \|\boldsymbol{w}\| \in [\gamma R, R]$ and $\gamma = 1 - o(1)$, and let θ denote the angle between $\boldsymbol{v}$ and $\boldsymbol{w}$. Then spherical LSH satisfies:*

$$\mathbb{P}_{h \in \mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] = \exp\left[-\frac{\sqrt{n}}{2} \tan^2\left(\frac{\theta}{2}\right)(1 + o(1))\right]. \qquad (3)$$

Note that for $\theta_1 = \frac{\pi}{3}$ and $\theta_2 = \frac{\pi}{2}$ this leads to $\rho = \frac{\ln(p_1)}{\ln(p_2)} = \frac{\tan^2(\pi/6)}{\tan^2(\pi/4)}(1 + o(1)) = \frac{1}{3} + o(1)$. This is significantly smaller than the related value of ρ for angular hashing with $\theta_1 = \frac{\pi}{3}$ and $\theta_2 = \frac{\pi}{2}$, which is $\rho' = \log_2(\frac{3}{2}) \approx 0.585$.

[4] Note that Andoni et al. sample vectors with average norm $\sqrt{n}$ instead, which means that everything in our description is scaled by a factor $\sqrt{n}$.

Algorithm 1 The Nguyen-Vidick sieve algorithm (sieving step)

Compute the maximum norm $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Initialize empty lists L_{m+1} and C_{m+1}
for each $\boldsymbol{v} \in L_m$ **do**
    if $\|\boldsymbol{v}\| \leq \gamma R$ **then**
        Add $\boldsymbol{v}$ to the list L_{m+1}
    else
        for each $\boldsymbol{w} \in C_{m+1}$ **do**
            if $\|\boldsymbol{v} - \boldsymbol{w}\| \leq \gamma R$ **then**
                Add $\boldsymbol{v} - \boldsymbol{w}$ to the list L_{m+1} and continue the outer loop
        Add $\boldsymbol{v}$ to the centers C_{m+1}

3 From the Nguyen-Vidick sieve to the SphereSieve

We will now describe how spherical LSH can be applied to sieving. More precisely, we will show how spherical LSH can be applied to the heuristic sieve algorithm of Nguyen and Vidick [35]. Applying the same technique to the practically superior GaussSieve [32] seems difficult, and whether this is at all possible is left as an open problem.

3.1 The Nguyen-Vidick sieve

Initially the Nguyen-Vidick sieve starts with a long list L_0 of long lattice vectors (generated from a discrete Gaussian distribution on the lattice), and it iteratively builds shorter lists of shorter lattice vectors L_{m+1} by applying a sieve to L_m. After $\mathrm{poly}(n)$ applications of the sieve, one hopes to be left with a list L_M containing a shortest non-zero lattice vector. At the heart of the heuristic sieve algorithm of Nguyen and Vidick lies the sieving step, mapping a list L_m to the next list L_{m+1}, and this sieving step is described in Algorithm 1.

The sieving step in Algorithm 1 can be described as follows. We start with an exponentially long list of vectors L_m, and we assume the longest of these vectors has length R; computing R can trivially be done in $\tilde{O}(|L_m|)$ time. Then, for a given parameter $\gamma < 1$ close to 1, we immediately add all vectors of norm less than γR to the next list L_{m+1}; these vectors are not modified in this iteration of the sieve. In the next iteration we want all vectors in L_{m+1} to have norm less than γR, and the remaining vectors in the spherical shell $\mathcal{S} = \{\boldsymbol{v} \in L_m : \gamma R < \|\boldsymbol{v}\| \leq R\}$ do not satisfy this condition, so the main task of the sieving step is to combine lattice vectors in $L_m \cap \mathcal{S}$ to make shorter vectors, which can then be added to L_{m+1}. To do this, we first initialize an empty list of *centers* C_{m+1}, and for each vector $\boldsymbol{v} \in \mathcal{S}$ we do one of the following:

– If $\boldsymbol{v}$ is far away from all center vectors $\boldsymbol{w} \in C_{m+1}$, we add $\boldsymbol{v}$ to C_{m+1};
– If $\boldsymbol{v}$ is close to a center vector $\boldsymbol{w} \in C_{m+1}$, we add $\boldsymbol{v} - \boldsymbol{w}$ to L_{m+1}.

We go through all vectors in L_m one by one, each time deciding whether to add something to L_{m+1} or to C_{m+1}. Note that for each list vector, this decision can be made in $O(|C_{m+1}|) = O(|L_m|)$ time by simply going through all vectors $\boldsymbol{w} \in C_{m+1}$ and checking whether it is close to $\boldsymbol{v}$. Finally, we obtain a set $C_{m+1} \subset L_m$ which intuitively *covers* $\mathcal{S}$ with balls of radius γR, and we obtain a set L_{m+1} of short vectors. Since the size of C_{m+1} is bounded from above by $2^{\Theta(n)}$, we know that if $|L_m|$ is large enough, many vectors will be included in $|L_{m+1}|$ as well.

To analyze their heuristic sieve algorithm, Nguyen and Vidick used (a slightly stronger version of) the following heuristic assumption.

Heuristic 1 *The angle $\Theta(\boldsymbol{v}, \boldsymbol{w})$ between two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ in Line 8 in Algorithm 1 follows the same distribution as the distribution of angles $\Theta(\boldsymbol{v}, \boldsymbol{w})$ obtained by drawing $\boldsymbol{v}$ and $\boldsymbol{w}$ at random from the unit sphere.*

Using this heuristic assumption, Nguyen and Vidick showed that an initial list of size $|L_0| = (4/3)^{n/2+o(n)} \approx 2^{0.2075n+o(n)}$ suffices to find a shortest vector if $\gamma \approx 1$ [35]. Since the time complexity is dominated by comparing almost every pair of vectors in L_i in each sieving step, this leads to a time complexity quadratic in $|L_i|$. Overall, this means that under the above heuristic assumption, the Nguyen-Vidick sieve solves SVP in time $2^{0.415n+o(n)}$ and space $2^{0.2076n+o(n)}$.

3.2 The SphereSieve

Algorithm 2 describes how we can apply spherical LSH to the sieve step of Nguyen and Vidick's heuristic sieve algorithm, in a similar fashion as how angular LSH was applied to sieving in [24].

To apply spherical LSH to sieving efficiently, there are some subtle issues that we need to consider. For instance, while the angular hashing technique of Charikar considered in [24] is scale invariant, the parameters of spherical LSH slightly change if all vectors in L and the target vector are multiplied by a scalar. This means that for each application of the sieving step, the parameters might change and we must build fresh hash tables. Although this might increase the practical time and space complexities, this does not affect the algorithm's asymptotics.

More importantly, to justify that we can apply spherical LSH (i.e., to justify the application of Lemma 2), we need to guarantee that $\|\boldsymbol{v}\| \approx$

Algorithm 2 The SphereSieve algorithm (sieving step)

1: Compute the maximum norm $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
2: Initialize an empty list L_{m+1}
3: Initialize t empty hash tables T_i
4: Sample $k \cdot t$ random spherical hash functions $h_{i,j} \in \mathcal{H}$
5: **for each** $\boldsymbol{v} \in L_m$ **do**
6: **if** $\|\boldsymbol{v}\| \leq \gamma R$ **then**
7: Add $\boldsymbol{v}$ to the list L_{m+1}
8: **else**
9: Obtain the set of candidates $C = \bigcup_{i=1}^{t} T_i[h_i(\pm \boldsymbol{v})]$
10: **for each** $\boldsymbol{w} \in C$ **do**
11: **if** $\|\boldsymbol{v} - \boldsymbol{w}\| \leq \gamma R$ **then**
12: Add $\boldsymbol{v} - \boldsymbol{w}$ to the list L_{m+1}
13: Continue the outermost loop
14: Add $\boldsymbol{v}$ to all t hash tables T_i

$\|\boldsymbol{w}\|$ for targets $\boldsymbol{v}$ and (candidate) near neighbors $\boldsymbol{w}$, i.e., that all these vectors approximately lie on the surface of a sphere. To see why this is true, consider a target vector $\boldsymbol{v}$ and a list vector $\boldsymbol{w}$. By definition of R, we know that $\boldsymbol{v}$ and $\boldsymbol{w}$ both have norm at most R. Moreover, the case $\|\boldsymbol{v}\| \leq \gamma R$ is handled separately (in polynomial time) in Lines 6–7, and the fact that $\boldsymbol{w} \in C_{m+1}$ implies that $\|\boldsymbol{w}\| > \gamma R$ as well. So when we get to the search in Lines 10–13, we know that the norms of both vectors satisfy $\|\boldsymbol{v}\|, \|\boldsymbol{w}\| \in [\gamma R, R]$. To get the optimal asymptotic time and space complexities, Nguyen and Vidick further let $\gamma \to 1$, which we needed to apply Lemma 2.

4 Theoretical results

To obtain a first basic estimate of the potential improvements to the time and space complexities using spherical LSH, we first note that in high dimensions "almost everything is orthogonal." In other words, angles close to 90° are much more likely to occur between two random vectors than much smaller angles. So one might guess that for a target vector $\boldsymbol{v}$ and a random list vector $\boldsymbol{w}$, with high probability their angle is close to 90°. On the other hand, two non-reduced vectors $\boldsymbol{v}, \boldsymbol{w}$ of similar norm for which the if-clause in Line 8 is true (i.e., for which $\|\boldsymbol{v} - \boldsymbol{w}\| \leq \gamma R = R(1 - o(1))$ and $\|\boldsymbol{v}\|, \|\boldsymbol{w}\| \approx R$), always have a common angle of at most $60° + o(1)$. We therefore expect this angle to be close to 60° with high probability. Under the extreme (and imprecise) assumption that all angles between pairwise reduced vectors are *exactly* 90°, and non-reduced angles are at

most 60°, we obtain the following estimate for the optimized time and space complexities using spherical LSH.

Estimate 1 *Assuming that all reduced pairs of vectors are exactly orthogonal, under Heuristic 1 the SphereSieve solves SVP in time and space at most* $(4/3)^{2n/3+o(n)} = 2^{0.2767n+o(n)}$*, using the following parameters:*

$$k = \Theta(\sqrt{n}), \qquad t = (4/3)^{n/6+o(n)} = 2^{0.0692n+o(n)}. \tag{4}$$

Proof. Assuming all reduced pairs of vectors are orthogonal, we obtain $\rho = \frac{1}{3}$ as described in Section 2.4. Since the time complexity is dominated by performing $O(N)$ nearest-neighbor searches on a list of size $O(N)$, with $N = (4/3)^{n/2+o(n)} \approx 2^{0.2075n+o(n)}$, the result follows from Lemma 1.

Of course in practice not all reduced angles are actually 90°, and one should carefully analyze what is the real probability that a vector $\boldsymbol{w}$ whose angle with $\boldsymbol{v}$ is more than 60°, is found as a candidate due to a collision in one of the hash tables. In that sense, Estimate 1 should only be considered a rough estimate, and it gives a lower bound on the best time complexity that we may hope to achieve with this method. Note however that the estimated time complexity is significantly better than the similar estimate obtained for the angular LSH-based HashSieve of Laarhoven [24], for which the estimated time complexity was $2^{0.3289n+o(n)}$. Therefore, one might guess that also the actual asymptotic time complexity, derived after a more precise analysis, is better than that of the HashSieve.

The following proposition shows that this is indeed the case, and it describes exactly what the asymptotic time and space complexities are when the parameters are fully optimized to minimize the asymptotic time complexity. A proof of Proposition 1 and an explanation of the constant 0.2972 can be found in Appendix A.

Proposition 1. *The SphereSieve heuristically solves SVP in time and space* $2^{0.2972n+o(n)}$ *using the following parameters:*

$$k = \Theta(\sqrt{n}), \qquad t = 2^{0.0896n+o(n)}. \tag{5}$$

By varying k and t, we further obtain the trade-off between the time and space complexities indicated by the solid blue curve in Figure 1.

Note that the estimated parameters from Estimate 1 are not far off from the main result of Proposition 1. In other words, assuming that reduced vectors are always orthogonal is not entirely realistic, but it provides a reasonable first estimate of the parameters that we have to use.

Finally, note that the space complexity increases by a factor t and thus increases exponentially compared to the Nguyen-Vidick sieve. To get rid of this exponential increase in the memory, instead of storing all hash tables in memory at the same time we may choose to go through the hash tables one by one, as in [24]; we first build one hash table by adding all vectors to their corresponding hash buckets, and then we look for pairs of nearby vectors in each bucket (whose difference has norm less than γR), and add all the found vectors to our new list L_{m+1}. As the number of vectors in each hash bucket is $2^{o(n)}$, comparing all pairs of vectors in a hash bucket can be done in $2^{o(n)}$ time and the cost of processing one hash table is $2^{0.208n+o(n)}$. We then repeat this $t = 2^{0.0896n+o(n)}$ times (each time removing the previous hash table from memory) to finally achieve the following result.

Theorem 1. *The SphereSieve heuristically solves the exact shortest vector problem in time* $2^{0.2972n+o(n)}$ *and space* $2^{0.2075n+o(n)}$.

5 A practical SphereSieve variant and two-level sieving

Let us briefly consider how this algorithm can be made slightly more practical. In particular, note that each spherical hash function requires the use of $U = 2^{\Theta(n)}$ vectors $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_U$, which are (roughly) sampled from the surface of the unit hypersphere. In total, this means that the algorithm uses $t \cdot k \cdot U$ random unit vectors to define hash regions on the sphere, and all these vectors need to be stored in memory. Generating so many random vectors from the surface of the unit hypersphere seems unnecessary, especially considering that we already have a list L_m containing vectors which (almost) lie on the surface of a hypersphere as well.

The above suggests to make the following modification to the algorithm: for building a single hash function $h_{i,j}$, instead of sampling $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_U$ randomly from the surface of the sphere, we randomly sample these vectors from (a scaled version of) L_m. In other words, we use the vectors in L_m to shape the hash regions, rather than sampling and storing new vectors in memory solely for this purpose. According to Heuristic 1 these vectors are also distributed randomly on the surface of the sphere, and so using the same heuristic assumption we can justify that this modification does not drastically alter the behavior of the algorithm. Note that since we need $t \cdot k$ hash functions, we need $t \cdot k$ selections of U vectors from L_m. Fortunately $t \cdot k \cdot U \ll |L_m|$ (cf. Proposition 1), so by independently sampling U random vectors from L_m for each of the $t \cdot k$ hash functions, the hash functions $h_{i,j}$ can practically be considered independent.

Relation with two-level sieving. Now, note that for a single hash function, we first use a small set of hash region-defining vectors U (where the radius of each hash region is approximately $(\sqrt{2} - o(1))R$), and then we use the NV-sieve in each of these regions separately to make lists of centers C_{m+1} (where a vector is considered nearby if it is within a radius of approximately $(1 - o(1))R$). This very closely resembles the ideas behind Wang et al.'s two-level sieve algorithm [46], where a list $C_1 (\cong U)$ of outer centers is built (defining balls of radius $\gamma_1 \cdot R$), and each of the centers $\boldsymbol{w}$ of this outer list contains an inner list $C_2^{\boldsymbol{w}} (\cong C_{m+1})$ of center vectors (defining a ball of radius $\gamma_2 \cdot R$). In fact, for $t = k = 1$, the SphereSieve is almost identical to the two-level sieve with $\gamma_1 \approx \sqrt{2}$ and $\gamma_2 \approx 1$!

How order matters. One difference between the two methods is that the size of U in the SphereSieve is sub-exponential ($2^{\Theta(\sqrt{n})}$), compared to single exponential ($2^{\Theta(n)}$) in the two-level sieve, which means that in our case, one of these hash tables is relatively 'cheap' to build. As a result, the asymptotic exponential overhead in our case only comes from t. However, the key difference that allows us to obtain the improved performance overall seems to be that the analysis of spherical LSH [5, 6] (and the closely related analysis of the celebrated Euclidean LSH family [4]) makes crucial use of the fact that the outer list C_1 is *ordered*, and this same order is used each time a vector is assigned to a hash region. Without this observation, Lemma 2 does not hold, and as in [46, 47] one would then have to resort to computing intersections of volumes of complicated n-dimensional objects to obtain bounds on the number of points needed to make this method work. One might say that the *order* imposed on C_1 is exactly what makes spherical LSH asymptotically more efficient than the two-level sieve of Wang et al. [46] with $\gamma_1 \approx \sqrt{2}$ and $\gamma_2 \approx 1$.

6 Discussion

Theoretically, Theorem 1 shows that for sufficiently high dimensions n, spherical LSH leads to even bigger speed-ups than angular LSH [24]. With a heuristic time complexity less than $2^{0.2972n+o(n)} < 2^{3n/10+o(n)}$, the SphereSieve is the fastest heuristic algorithm to date for solving SVP in high dimensions. As a result, one might conclude that in high dimensions, to achieve $3k$ bits of security for a lattice-based cryptographic primitive relying on the hardness of exact SVP, one should use a lattice of dimension at least $10k$. As most cryptographic schemes are broken even if a short lattice vector is found (which by using BKZ [44, 45] means we can reduce

the dimension in which we need to solve SVP), and the time complexity of the SphereSieve is lower than $2^{n/3+o(n)}$, one should probably use lattices of dimension higher than $10k$ to guarantee $3k$ bits of security. So various parameter choices relying on the estimates of e.g. Chen and Nguyen [13] (solving SVP in dimension 200 takes time 2^{111}) would be too optimistic.

Although the leading term $0.2972n$ in the exponent is the best known so far and dominates the complexity in high dimensions, this does not tell the whole story. Especially for the SphereSieve presented in this paper, the $o(n)$-terms in the exponent are not negligible at all for moderate n. Experiments further indicate [16, 24, 27, 28, 32, 35, 41] that the practical time complexity of various sieving algorithms in moderate dimensions n may be higher than quadratic in the list size if we set γ close to 1, while setting $\gamma \ll 1$ makes the use of spherical LSH problematic. Moreover, while the angular LSH method of Charikar [12] considered in [24] is very efficient and hashes can be computed in linear time, with spherical LSH even the cost of computing a single hash value (before amplification) is already sub-exponential (and super-polynomial) in n. So in practice it is not clear whether the SphereSieve will outperform the angular LSH-based sieving algorithm of Laarhoven [24] for any feasible dimension n. Finding an accurate description of the practical costs of finding short(est) vectors in dimension n remains a central problem in lattice cryptography.

An important question for future work remains whether spherical LSH can be made truly efficient. While asymptotic costs are important, lower order terms matter in practice as well, and being able to compute hashes in poly(n)-time would make the SphereSieve significantly faster. As mentioned in Section 3, being able to apply spherical LSH to the faster GaussSieve [32] may also lead to a faster sieve. The recent work [9] takes a first step in this direction, showing that the same asymptotics as the SphereSieve can be achieved with efficient hashing.

References

1. Aggarwal, D., Dadush, D., Regev, O., Stephens-Davidowitz, N.: Solving the shortest vector problem in 2^n time via discrete Gaussian sampling. In: STOC (2015)
2. Ajtai, M.: The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In: STOC, pp. 10–19 (1998)
3. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: STOC, pp. 601–610 (2001)
4. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS, pp. 459–468 (2006)
5. Andoni, A., Indyk, P., Nguyen, H. L., Razenshteyn, I.: Beyond locality-sensitive hashing. In: SODA, pp. 1018–1028 (2014)

6. Andoni, A., Razenshteyn, I.: Optimal data-dependent hashing for approximate near neighbors. In: STOC (2015)
7. Becker, A., Gama, N., Joux, A.: A sieve algorithm based on overlattices. In: ANTS, pp. 49–70 (2014)
8. Becker, A., Gama, N., Joux, A.: Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522 (2015)
9. Becker, A., Laarhoven, T.: Efficient sieving on (ideal) lattices using cross-polytopic LSH. Preprint (2015)
10. Bernstein, D. J., Buchmann, J., Dahmen, E.: Post-quantum cryptography (2009)
11. Bos, J., Naehrig, M., van de Pol, J.: Sieving for shortest vectors in ideal lattices: a practical perspective. Cryptology ePrint Archive, Report 2014/880 (2014)
12. Charikar, M. S.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388 (2002)
13. Chen, Y., Nguyen, P. Q.: BKZ 2.0: Better lattice security estimates. In: ASIACRYPT, pp. 1–20 (2011)
14. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SOCG, pp. 253–262 (2004)
15. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice. Mathematics of Computation 44(170), pp. 463–471 (1985)
16. Fitzpatrick, R., Bischof, C., Buchmann, J., Dagdelen, Ö., Göpfert, F., Mariano, A., Yang, B.-Y.: Tuning GaussSieve for speed. In: LATINCRYPT, pp. 284–301 (2014)
17. Gama, N., Nguyen, P. Q., Regev, O.: Lattice enumeration using extreme pruning. In: EUROCRYPT, pp. 257–278 (2010)
18. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC, pp. 169–178 (2009)
19. Hanrot, G., Pujol, X., Stehlé, D.: Algorithms for the shortest and closest lattice vector problems. In: IWCC, pp. 159–190 (2011)
20. Hoffstein, J., Pipher, J., Silverman, J. H.: NTRU: A ring-based public key cryptosystem. In: ANTS, pp. 267–288 (1998)
21. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
22. Ishiguro, T., Kiyomoto, S., Miyake, Y., Takagi, T.: Parallel Gauss Sieve algorithm: solving the SVP challenge over a 128-dimensional ideal lattice. In: PKC, pp. 411–428 (2014)
23. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: STOC, pp. 193–206 (1983)
24. Laarhoven, T.: Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In: CRYPTO (2015)
25. Laarhoven, T., Mosca, M., van de Pol, J.: Finding shortest lattice vectors faster using quantum search. Designs, Codes and Cryptography (2015)
26. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: CT-RSA, pp. 319–339 (2011)
27. Mariano, A., Timnat, S., Bischof, C.: Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In: SBAC-PAD, pp. 278–285 (2014)
28. Mariano, A., Dagdelen, Ö, Bischof, C.: A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In: APCI&E (2014)
29. Mariano, A., Laarhoven, T., Bischof, C.: Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In: ICPP (2015)

30. Micciancio, D.: The shortest vector in a lattice is hard to approximate to within some constant. In: FOCS, pp. 92–98 (1998)
31. Micciancio, D., Voulgaris, P.: A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In: STOC, pp. 351–358 (2010)
32. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA, pp. 1468–1480 (2010)
33. Micciancio, D., Walter, M.: Fast lattice point enumeration with minimal overhead. In: SODA, pp. 276–294 (2015)
34. Milde, B., Schneider, M.: A parallel implementation of GaussSieve for the shortest vector problem in lattices. In: PaCT, pp. 452–458 (2011)
35. Nguyen, P. Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. Journal of Mathematical Cryptology 2(2), pp. 181–207 (2008)
36. Plantard, T., Schneider, M.: Ideal lattice challenge. Online at `http://latticechallenge.org/ideallattice-challenge/` (2014)
37. Pohst, M. E.: On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. ACM SIGSAM Bulletin 15(1), pp. 37–44 (1981)
38. van de Pol, J., Smart, N. P.: Estimating key sizes for high dimensional lattice-based systems. In: IMACC, pp. 290–303 (2013)
39. Pujol, X., Stehlé, D.: Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605 (2009)
40. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: STOC, pp. 84–93 (2005)
41. Schneider, M.: Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In: WALCOM, pp. 9–97 (2011)
42. Schneider, M.: Sieving for short vectors in ideal lattices. In: AFRICACRYPT, pp. 375–391 (2013)
43. Schneider, M., Gama, N., Baumann, P., Nobach, L.: SVP challenge. Online at `http://latticechallenge.org/svp-challenge` (2015)
44. Schnorr, C.-P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science, 53(2), pp. 201–224 (1987)
45. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Mathematical Programming, 66(2), pp. 181–199 (1994)
46. Wang, X., Liu, M., Tian, C., Bi, J.: Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In: ASIACCS, pp. 1–9 (2011)
47. Zhang, F., Pan, Y., Hu, G.: A three-level sieve algorithm for the shortest vector problem. In: SAC, pp. 29–47 (2013)

A Proof of Proposition 1

To prove Proposition 1, we will show how to choose a sequence of parameters $\{(k_n, t_n)\}_{n\in\mathbb{N}}$ such that for large n, the following holds:

1. The probability that a list vector $\boldsymbol{w}$ close[5] to a target vector $\boldsymbol{v}$ collides with $\boldsymbol{v}$ in at least one of the t hash tables is at least constant in n:

$$p_1^* = \mathbb{P}_{\{h_{i,j}\}\subset\mathcal{H}}(\boldsymbol{v}, \boldsymbol{w} \text{ collide} \mid \theta(\boldsymbol{v}, \boldsymbol{w}) \leq \tfrac{\pi}{3}) \geq 1 - \varepsilon. \quad (\varepsilon \neq \varepsilon(n)) \quad (6)$$

2. The average probability that a list vector $\boldsymbol{w}$ far away[5] from a target vector $\boldsymbol{v}$ collides with $\boldsymbol{v}$ is exponentially small:

$$p_2^* = \mathbb{P}_{\{h_{i,j}\}\subset\mathcal{H}}(\boldsymbol{v}, \boldsymbol{w} \text{ collide} \mid \theta(\boldsymbol{v}, \boldsymbol{w}) > \tfrac{\pi}{3}) \leq N^{-0.5681+o(1)}. \quad (7)$$

3. The number of hash tables grows as $t = N^{0.4319+o(1)}$.

This would imply that for each search, the number of candidate vectors is of the order $N \cdot N^{-0.5681} = N^{0.4319}$. Overall we search the list $\tilde{O}(N)$ times, so after substituting $N = (4/3)^{n/2+o(n)}$ this leads to the following time and space complexities:

- Time (hashing): $O(N \cdot t) = 2^{0.2972n+o(n)}$.
- Time (searching): $O(N^2 \cdot p_2^*) = 2^{0.2972n+o(n)}$.
- Space: $O(N \cdot t) = 2^{0.2972n+o(n)}$.

The next two subsections are dedicated to proving Equations (6) and (7).

A.1 Good vectors collide with constant probability

The following lemma shows how to choose k (in terms of t) to guarantee that (6) holds.

Lemma 3. *Let $\varepsilon > 0$ and let $k = 6n^{-1/2}(\ln t - \ln\ln(1/\varepsilon)) \approx (6\ln t)/\sqrt{n}$. Then the probability that reducing vectors collide in at least one of the hash tables is at least $1 - \varepsilon$.*

Proof. The probability that a reducing vector $\boldsymbol{w}$ is a candidate vector, given the angle $\Theta = \Theta(\boldsymbol{v}, \boldsymbol{w}) \in (0, \frac{\pi}{3})$, is $p_1^* = \mathbb{E}_{\Theta\in(0,\frac{\pi}{3})}[p^*(\Theta)]$, where we recall that $p^*(\theta) = 1 - (1 - p(\theta)^k)^t$ and $p(\theta) = \mathbb{P}_{h\in\mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})]$ is given in Lemma 2. Since $p^*(\Theta)$ is strictly decreasing in Θ, we can obtain a lower bound by substituting $\Theta = \frac{\pi}{3}$ above. Using the bound $1 - x \leq e^{-x}$ which holds for all x, and inserting the given expression for k, we obtain $p_1^* \geq p^*\left(\frac{\pi}{3}\right) = 1 - (1 - \exp(\ln\ln(\frac{1}{\varepsilon}) - \ln t))^t = 1 - \left(1 - \frac{\ln(1/\varepsilon)}{t}\right)^t \geq 1 - \varepsilon.$

[5] Here "close" means that $\|\boldsymbol{v} - \boldsymbol{w}\| \leq \gamma R$, which corresponds to $\theta(\boldsymbol{v}, \boldsymbol{w}) \leq 60^\circ + o(1)$. Similarly "far away" corresponds to a large angle $\theta(\boldsymbol{v}, \boldsymbol{w}) > 60^\circ + o(1)$.

A.2 Bad vectors collide with low probability

We first recall a lemma about the density of angles between random vectors. In short, the density at an angle θ is proportional to $(\sin\theta)^n$.

Lemma 4. *[24, Lemma 4] Assuming Heuristic 1 holds, the pdf $f(\theta)$ of the angle between target vectors and list vectors satisfies*

$$f(\theta) = \sqrt{\frac{2n}{\pi}}\,(\sin\theta)^{n-2}\,[1+o(1)] = 2^{n\log_2\sin\theta+o(n)}. \tag{8}$$

The following lemma relates the collision probability p_2^* of (7) to the parameters k and t. Since Lemma 3 relates k to t, this means that only t ultimately remains to be chosen.

Lemma 5. *Suppose $N = 2^{c_n\cdot n}$ with $c_n \geq \gamma_1 = \frac{1}{2}\log_2(\frac{4}{3}) \approx 0.2075$, and suppose $t = 2^{c_t\cdot n}$. Let $k = \frac{6\ln t}{\sqrt{n}}(1-o(1))$. Then, for large n, under Heuristic 1 we have*

$$p_2^* = \mathbb{P}_{\{h_{i,j}\}\subset\mathcal{H}}(\boldsymbol{v},\boldsymbol{w}\ \textit{collide} \mid \theta(\boldsymbol{v},\boldsymbol{w}) > \tfrac{\pi}{3}) \leq O(N^{-\alpha}), \tag{9}$$

where $\alpha \in (0,1)$ is defined as

$$\alpha = \frac{-1}{c_n}\left[\max_{\theta\in(\frac{\pi}{3},\frac{\pi}{2})}\left\{\log_2\sin\theta - \left(3\tan^2\left(\frac{\theta}{2}\right)-1\right)c_t\right\}\right] + o(1). \tag{10}$$

Proof. First, if we know the angle $\theta \in (\frac{\pi}{3},\frac{\pi}{2})$ between two bad vectors, then according to Lemma 2 the probability of a collision in at least one of the hash tables is equal to

$$p^*(\theta) = 1 - \left(1 - \exp\left[-\frac{k\sqrt{n}}{2}\tan^2\left(\frac{\theta}{2}\right)(1+o(1))\right]\right)^t. \tag{11}$$

Letting $f(\theta)$ denote the density of angles θ on $(\frac{\pi}{3},\frac{\pi}{2})$, we have

$$p_2^* = \mathbb{E}_{\Theta\in(\frac{\pi}{3},\frac{\pi}{2})}\left[p^*(\Theta)\right] = \int_{\pi/3}^{\pi/2} f(\theta)p^*(\theta)d\theta. \tag{12}$$

Substituting $p^*(\theta)$ and the expression of Lemma 4 for $f(\theta)$, noting that $\int_{\pi/3}^{\pi/2} f(\theta)d\theta \approx \int_0^{\pi/2} f(\theta)d\theta = 1$, we get

$$p_2^* = \int_{\pi/3}^{\pi/2}(\sin\theta)^n\left[1-\left(1-\exp\left[-3\ln t\tan^2\left(\tfrac{\theta}{2}\right)(1+o(1))\right]\right)^t\right]d\theta. \tag{13}$$

For convenience, let us write $w(\theta) = [-3 \ln t \tan^2\left(\frac{\theta}{2}\right)(1+o(1))$. Note that for $\theta \gg \frac{\pi}{3}$ we have $w(\theta) \ll -\ln t$ so that $(1-\exp w(\theta))^t \approx 1 - t \exp w(\theta)$, in which case we can simplify the expression between square brackets. However, the integration range includes $\frac{\pi}{3}$ as well, so to be careful we will split the integration interval at $\frac{\pi}{3} + \delta$, where $\delta = \Theta(n^{-1/2})$. (Note that any value δ with $\frac{1}{n} \ll \delta \ll 1$ suffices.)

$$p_2^* = \underbrace{\int_{\pi/3}^{\pi/3+\delta} f(\theta) p^*(\theta) d\theta}_{I_1} + \underbrace{\int_{\pi/3+\delta}^{\pi/2} f(\theta) p^*(\theta) d\theta}_{I_2}. \tag{14}$$

Bounding I_1. Using $f(\theta) \leq f(\frac{\pi}{3}+\delta)$, $p^*(\theta) \leq 1$, and $\sin(\frac{\pi}{3}+\delta) = \frac{1}{2}\sqrt{3}\,[1+O(\delta)]$ (which follows from a Taylor expansion of $\sin x$ around $x = \frac{\pi}{3}$), we obtain

$$I_1 \leq \mathrm{poly}(n) \sin^n(\tfrac{\pi}{3}+\delta) = \mathrm{poly}(n)(\tfrac{\sqrt{3}}{2})^n\,(1+O(\delta))^n = 2^{-\gamma_1 n + o(n)}. \tag{15}$$

Bounding I_2. For I_2, our choice of δ is sufficient to make the aforementioned approximation work[6]. Thus, for I_2 we obtain the simplified expression

$$I_2 \leq \mathrm{poly}(n) \int_{\pi/3+\delta}^{\pi/2} (\sin\theta)^n t \exp\left[-3 \ln t \tan^2\left(\frac{\theta}{2}\right)(1+o(1))\right] d\theta \tag{16}$$

$$\leq \int_{\pi/3}^{\pi/2} 2^{n \log_2 \sin\theta - (3\tan^2\left(\frac{\theta}{2}\right)-1)\log_2 t + o(n)} d\theta. \tag{17}$$

Note that the integrand is exponential in n and that the exponent $E(\theta) = n \log_2 \sin\theta + (-3\tan^2\frac{\theta}{2} - 1)\log_2 t$ is a continuous, differentiable function of θ. So the asymptotic behavior of the entire integral I_2 is the same as the asymptotic behavior of the integrand's maximum value:

$$\log_2 I_2 \leq \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} \left\{ n \log_2 \sin\theta - \left(3\tan^2\frac{\theta}{2} - 1\right)\log_2 t \right\} + o(n). \tag{18}$$

Bounding $p_2^* = I_1 + I_2$. Combining (15), (18), and $c_t = \frac{1}{n}\log_2 t$, we have

$$\frac{\log_2 p_2^*}{n} \leq \max\{-\gamma_1,\ \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} \{\log_2 \sin\theta - (3\tan^2\tfrac{\theta}{2} - 1)c_t\}\} + o(1). \tag{19}$$

The assumption $c_n \geq \gamma_1$ and the definition of $\alpha \leq 1$ now give $\log_2 p_2^* \leq -\alpha c_n n + o(n)$ which completes the proof.

[6] By choosing the order terms in k appropriately, the $o(1)$-term inside $w(\theta)$ may be cancelled out, in which case the δ-term dominates. Note that the $o(1)$-term in $w(\theta)$ can be further controlled by the choice of $\gamma = 1 - o(1)$.

A.3 Balancing the parameters

Recall that the overall time and space complexities are given by $O(N \cdot t) = 2^{(c_n+c_t)n+o(n)}$ (time for hashing), $O(N^2 \cdot p_2^*) = 2^{(c_n+(1-\alpha)c_n)n+o(n)}$ (time for comparing vectors), and $O(N \cdot t) = 2^{(c_n+c_t)n+o(n)}$ (memory requirement). For the overall time and space complexities $2^{c_{\text{time}}n}$ and $2^{c_{\text{space}}n}$ we find

$$c_{\text{time}} = c_n + \max\{c_t, (1-\alpha)c_n\} + o(1), \quad c_{\text{space}} = c_n + c_t + o(1). \quad (20)$$

Further recall that from Nguyen and Vidick's analysis, we have $N = (4/3)^{n/2+o(n)}$ or $c_n = \gamma_1$. To balance the time complexities of hashing and searching, so that the overall time complexity is minimized, we solve $(1-\alpha)\gamma_1 = c_t$ numerically[7] for c_t to obtain the following corollary. Here θ^* denotes the dominant angle θ maximizing the expression in (10). Note that the final result takes into account the density at $\theta = \theta^*$ as well, and so the result does not simply follow from Lemma 2.

Corollary 1. *Taking $c_t \approx 0.089624$ leads to:*

$$\theta^* \approx 0.42540\pi, \ \alpha \approx 0.56812, \ c_{\text{time}} \approx 0.29714, \ c_{\text{space}} \approx 0.29714. \quad (21)$$

Thus, setting $t \approx 2^{0.08962n}$ and $k = \Theta(\sqrt{n})$, the heuristic time and space complexities of the SphereSieve algorithm are balanced at $2^{0.29714n+o(n)}$.

A.4 Trade-off between the space and time complexities

Finally, note that $c_t = 0$ leads to the original Nguyen-Vidick sieve algorithm, while $c_t \approx 0.089624$ minimizes the heuristic time complexity at the cost of more space. One can obtain a continuous trade-off between these two extremes by considering values $c_t \in (0, 0.089624)$. Numerically evaluating the resulting complexities for this range of values of c_t leads to the curve shown in Figure 1.

[7] Note that α is implicitly a function of c_t as well.