Sieving for shortest vectors in lattices using angular locality-sensitive hashing

Thijs Laarhoven

Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
mail@thijs.com

Abstract. By replacing the brute-force list search in sieving algorithms with Charikar's angular locality-sensitive hashing (LSH) method, we get both theoretical and practical speedups for solving the shortest vector problem (SVP) on lattices. Combining angular LSH with a variant of Nguyen and Vidick's heuristic sieve algorithm, we obtain heuristic time and space complexities for solving SVP of $2^{0.3366n+o(n)}$ and $2^{0.2075n+o(n)}$ respectively, while combining the same hash family with Micciancio and Voulgaris' GaussSieve algorithm leads to a practical algorithm with (conjectured) time and space complexities bounded by $2^{0.3366n+o(n)}$, leading to the best complexities for solving SVP in high dimensions to date.
Experiments show that in moderate dimensions the GaussSieve-based HashSieve algorithm already outperforms the GaussSieve, and the practical increase in the space complexity is smaller than the asymptotic bounds suggest, and can be further reduced with probing. Extrapolating to higher dimensions, we estimate that a fully optimized and parallelized implementation of the GaussSieve-based HashSieve algorithm might need a few core years to solve SVP in dimension 130 or even 140.

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

1 Introduction

Lattice cryptography. Over the past few decades, lattice-based cryptography has attracted wide attention from the cryptographic community, due to e.g. its presumed resistance against quantum attacks [14], average-case hardness guarantees [3], the existence of lattice-based fully homomorphic encryption schemes [21], and efficient cryptographic primitives like NTRU [23]. An important problem related to lattice cryptography is to estimate the hardness of the underlying hard lattice problems, such as finding short vectors; a good understanding is critical for accurately choosing parameters in lattice cryptography [33, 47].

Finding short vectors. Given a basis $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\} \subset \mathbb{R}^n$ of an n-dimensional lattice $\mathcal{L} = \sum_{i=1}^{n} \mathbb{Z}\boldsymbol{b}_i$, finding a shortest non-zero lattice vector (with respect to the Euclidean norm) or approximating it up to a constant factor is well-known to be NP-hard under randomized reductions [4, 27]. For large approximation factors, various fast algorithms for finding short vectors are known, such as the lattice basis reduction algorithms LLL [31] and BKZ [52, 53]. The latter has a block-size parameter β which can be tuned to obtain a trade-off between the time complexity and the quality of the output; the higher β, the longer the algorithm takes and the shorter the vectors in the output basis. BKZ uses an algorithm for solving the exact shortest vector problem (SVP) in lattices of dimension β as a subroutine, and the runtime of BKZ largely depends on the runtime of this subroutine. Estimating the complexity of solving exact SVP therefore has direct consequences for the estimated hardness of solving approximate SVP with BKZ.

Finding shortest vectors. In the original description of BKZ, enumeration was used as the SVP subroutine [18, 26, 46, 53]. This method has a low (polynomial) space complexity, but its runtime is super-exponential ($2^{\Omega(n \log n)}$), which is known to be suboptimal: sieving [5], the Voronoi cell algorithm [39], and the recent discrete Gaussian sampling approach [2] all run in single exponential time ($2^{O(n)}$). The main drawbacks of the latter methods are that their space complexities are exponential in n as well, and due to larger hidden constants in the exponents enumeration is commonly still considered more practical than other methods in moderate dimensions n [20, 41].

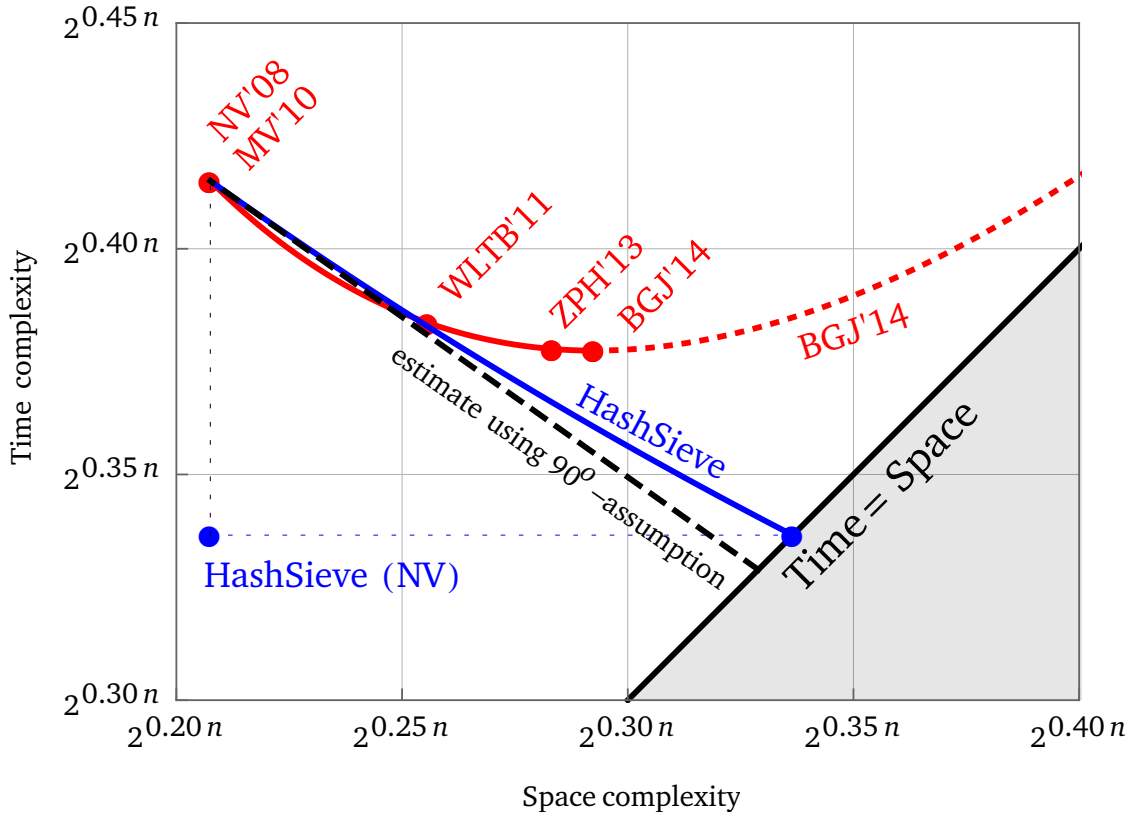


Fig. 1. The heuristic space-time trade-off of various heuristic sieves from the literature (red), and the heuristic trade-off between the space and time complexities obtained with the HashSieve (blue curve). For the NV-sieve, we can further process the hash tables sequentially to obtain a speedup rather than a trade-off (blue point). The dashed, gray line shows the estimate for the space-time trade-off of the HashSieve obtained by assuming that all reduced vectors are orthogonal (cf. Proposition 1). The referenced works are: NV'08 [43]; MV'10 [40]; WLTB'11 [55]; ZPH'13 [56]; BGJ'14 [10].

Sieving in arbitrary lattices. On the other hand, these other SVP algorithms are relatively new, and recent improvements have shown that at least sieving may be able to compete with enumeration in the future. While the original work of Ajtai et al. [5] showed only that sieving solves SVP in time and space $2^{O(n)}$, later work showed that one can provably solve SVP in arbitrary lattices in time $2^{2.47n+o(n)}$ and space $2^{1.24n+o(n)}$ [22,43,48]. Heuristic analyses of sieving algorithms further suggest that one may be able to solve SVP in time $2^{0.42n+o(n)}$ and space $2^{0.21n+o(n)}$ [10,40,43], or optimizing for time, in time $2^{0.38n+o(n)}$ and space $2^{0.29n+o(n)}$ [10,55,56]. Other works have shown how to speed up sieving in practice [15,19,25,35,36,42,49,50], and sieving recently made its way to the top 25 of the SVP challenge hall of fame [51], using the GaussSieve algorithm [29,40].

Sieving in ideal lattices. The potential of sieving is further illustrated by recent results in ideal lattices [15,25]; while it is not known how to use the additional structure in ideal lattices (commonly used in lattice cryptography) for enumeration or other SVP algorithms, sieving does admit significant polynomial speedups for ideal lattices, and the GaussSieve was recently used to solve SVP on an ideal lattice in dimension 128 [15,25,45]. This is higher than the highest dimension for which enumeration was used to find a record in either lattice challenge [45,51], which further illustrates the potential of sieving and the possible impact of further improvements to sieving and, in particular, the GaussSieve algorithm.

Contributions. In this work we show how to obtain exponential trade-offs and speedups for sieving using angular locality-sensitive hashing [16,24], a technique from the field of nearest neighbor searching. In short, for each list vector $\boldsymbol{w}$ we store low-dimensional, lossy *sketches*, such that vectors that are nearby have a higher probability of having the same sketch (hash) than vectors which are far apart. To search the list for nearby vectors we then do not go through the entire list of lattice vectors, but only consider those vectors that have at least one matching hash value in one of the hash tables.

Trading space for time. Storing all list vectors in exponentially many hash tables requires exponentially more space, but searching for nearby vectors can then be done exponentially faster as well, as many distant vectors are not considered for reductions. Optimizing for time, the resulting HashSieve algorithm has heuristic time and space complexities bounded by $2^{0.3366n+o(n)}$, while tuning the parameters differently we get a trade-off as illustrated by the solid blue curve in Figure 1.

From a tradeoff to a speedup. Applying angular LSH to a variant of the Nguyen-Vidick sieve [43], we further obtain an algorithm with heuristic time and space complexities of $2^{0.3366n+o(n)}$ and $2^{0.2075n+o(n)}$ respectively, as illustrated by the blue point in Figure 1. The key observation is that the hash tables of the HashSieve can be processed sequentially, and we only need to store and use one hash table at a time. The resulting algorithm achieves the same heuristic speed-up, but the asymptotic space complexity remains the same as in the original NV-sieve algorithm. This improvement is explained in detail in the full version. Note that this speedup does not appear to be compatible with the GaussSieve and only works with the NV-sieve, which may make the resulting algorithm slower in moderate dimensions, even though the memory used is much smaller.

Experimental results. Practical experiments with the (GaussSieve-based) HashSieve algorithm validate our heuristic analysis, and show that (i) already in low dimensions, the HashSieve outperforms the GaussSieve; and (ii) the increase in the space complexity is smaller than one might guess from only looking at the leading exponent of the space complexity. We also show how to further reduce the space complexity at almost no cost by a technique called probing, which reduces the required number of hash tables by a factor $\mathrm{poly}(n)$. In the end, these results will be an important guide for estimating the hardness of exact SVP in moderate dimensions, and for the hardness of approximate SVP in high dimensions using BKZ with sieving as the SVP subroutine.

Main ideas. While the use of LSH was briefly considered in the context of sieving by Nguyen and Vidick [43, Section 4.2.2], there are two main differences between their approach and ours:

- Nguyen and Vidick considered LSH families based on the *Euclidean distance* [6], while we argue it seems more natural to consider LSH families based on the *angular distance* or *cosine similarity* [16] between pairs of vectors[1].
- Nguyen and Vidick focused on the *worst-case* difference between nearby and faraway vectors, while we will focus on the *average-case* difference.

To illustrate the second point: the *smallest* angle between pairwise reduced vectors may be only slightly bigger than 60° (i.e. hardly any bigger than angles of non-reduced vectors), while in high dimensions the *average* angle between two pairwise reduced vectors is actually close to 90°. This precise average-case analysis is crucial in obtaining the complexity $2^{0.3366n+o(n)}$ rather than $2^{0.4150n+o(n)}$.

Outlook. Although this work focuses on applying angular LSH to sieving, more generally this work could be considered the first to succeed in applying LSH to lattice algorithms. Various recent follow-up works have already further investigated the use of different LSH methods [7, 8] and other nearest neighbor search methods [9, 11, 38] in the context of lattice sieving [11–13, 30, 37], and an open problem is whether other lattice algorithms (e.g. provable sieving algorithms, the Voronoi cell algorithm [39]) can benefit from related techniques as well.

Roadmap. In Section 2 we describe the technique of (angular) LSH for finding near(est) neighbors. Section 3 describes how to apply these techniques to the Nguyen-Vidick sieve, and Section 4 states the main result regarding the optimal time and space complexities of sieving using angular LSH. In Section 5 we finally describe how similar ideas can trivially be applied to the GaussSieve, and Section 6 describes experiments performed using the GaussSieve-based HashSieve, and possible consequences for the estimated complexity of SVP in high dimensions by extrapolating to higher dimensions.

[1] In the analysis for the Nguyen-Vidick sieve on thin spherical shells, as described later in this paper, Euclidean distances directly translate to angular distances and vice versa. In that sense we could have obtained an identical exponent if we had described everything in terms of Euclidean distances, although in that case we may not have considered using Charikar's hash family as it is described in terms of the cosine similarity measure.

2 Locality-sensitive hashing

2.1 Introduction

The near(est) neighbor problem is the following [24]: Given a list of n-dimensional vectors $L = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_N\} \subset \mathbb{R}^n$, preprocess L in such a way that, when later given a target vector $\boldsymbol{v} \notin L$, one can efficiently find an element $\boldsymbol{w} \in L$ which is close(st) to $\boldsymbol{v}$. While in low (fixed) dimensions n there may be ways to answer these queries in time sub-linear or even logarithmic in the list size N, in high dimensions it 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 objects is sometimes referred to as the "curse of dimensionality" [24].

Fortunately, if we know that the list of objects L has a certain structure, or if we know that 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. For instance, for the Euclidean norm, if it is known that the closest point $\boldsymbol{w}^* \in L$ lies at distance $\|\boldsymbol{v} - \boldsymbol{w}^*\| = r_1$, and all other points $\boldsymbol{w} \in L$ are at distance at least $\|\boldsymbol{v} - \boldsymbol{w}\| \geq r_2 = (1+\varepsilon)r_1$ from $\boldsymbol{v}$, then it is possible to preprocess L using time and space $O(N^{1+\rho})$, and answer queries in time $O(N^\rho)$, where $\rho = (1+\varepsilon)^{-2} < 1$ [6]. For $\varepsilon > 0$, this corresponds to a sub-linear time and sub-quadratic (super-linear) space complexity in N.

2.2 Hash families

The method of [6] described above, as well as the method we will use later, relies on using *locality-sensitive hash functions* [24]. These are functions h which map an n-dimensional vector $\boldsymbol{v}$ to a low-dimensional *sketch* of $\boldsymbol{v}$, such that vectors which are nearby in $\mathbb{R}^n$ have a high probability of having the same sketch, while vectors which are far away 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, we assume D is a certain similarity measure[2], and the set U below may be thought of as (a subset of) the natural numbers $\mathbb{N}$.

Definition 1. *[24] A family $\mathcal{H} = \{h : \mathbb{R}^n \to U\}$ is called (r_1, r_2, p_1, p_2)-sensitive for a similarity measure D if for any $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ we have*

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

Note that if we are given a hash family $\mathcal{H}$ which is (r_1, r_2, p_1, p_2)-sensitive with $p_1 \gg p_2$, then 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, by only looking at their hash values.

2.3 Amplification

Before turning to how such hash families may actually be constructed or used to find nearest neighbors, note that in general it is unknown 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 $p_1' > p_1$ and $p_2' < p_2$, thereby amplifying the gap between p_1 and p_2.

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 k different, pairwise independent functions $h_1, \ldots, h_k \in \mathcal{H}$ and a one-to-one mapping $f : U^k \to U$, and defining $h \in \mathcal{H}'$ as $h(\boldsymbol{v}) = f(h_1(\boldsymbol{v}), \ldots, h_k(\boldsymbol{v}))$. Clearly $h(\boldsymbol{v}) = h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) = h_i(\boldsymbol{w})$ for all $i \in [k]$, so if $\mathbb{P}[h_i(\boldsymbol{v}) = h_i(\boldsymbol{w})] = p_j$ for all i, then $\mathbb{P}[h(\boldsymbol{v}) = h(\boldsymbol{w})] = p_j^k$ for $j = 1, 2$.

[2] A similarity measure D may informally be thought of as a "slightly relaxed" (distance) metric, which may not satisfy all properties associated to metrics.

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 t different, pairwise independent functions $h_1, \ldots, h_t \in \mathcal{H}$, 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 *at least one* $i \in [t]$. Clearly $h(\boldsymbol{v}) \neq h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) \neq h_i(\boldsymbol{w})$ for all $i \in [t]$, so if $\mathbb{P}[h_i(\boldsymbol{v}) \neq h_i(\boldsymbol{w})] = 1 - p_j$ for all i, then $\mathbb{P}[h(\boldsymbol{v}) \neq h(\boldsymbol{w})] = (1-p_j)^t$ for $j = 1, 2$.[3]

Combining a k-wise AND-composition 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 $(r_1, r_2, 1-(1-p_1^k)^t, 1-(1-p_2^k)^t)$-sensitive hash family $\mathcal{H}'$ as follows:

$$(r_1, r_2, p_1, p_2) \quad \xrightarrow{k-\text{AND}} \quad (r_1, r_2, p_1^k, p_2^k) \quad \xrightarrow{t-\text{OR}} \quad (r_1, r_2, (1-p_1^k)^t, (1-p_2^k)^t).$$

As long as $p_1 > p_2$, we can always find values k and t such that $p_1^* = 1-(1-p_1^k)^t \approx 1$ is close to 1 and $p_2^* = 1-(1-p_2^k)^t \approx 0$ is very small.

2.4 Finding nearest neighbors

To use these hash families to find nearest neighbors, we may use the following method first described in [24]. First, we choose $t \cdot k$ random hash functions $h_{i,j} \in \mathcal{H}$, and we 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, we 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 vector $\boldsymbol{v}$, we 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) in a list of candidates, and search this set of candidates for a nearest neighbor.

Clearly, the quality of this algorithm for finding nearest neighbors depends on the quality of the underlying hash family $\mathcal{H}$ and on the parameters k and t. Larger values of k and t amplify the gap between the probabilities of finding 'good' (nearby) and 'bad' (faraway) vectors, which makes the list of candidates shorter, but larger parameters come at the cost of having to compute many hashes (both during the preprocessing and querying phases) and having to store many hash tables in memory. The following lemma shows how to balance k and t so that the overall time complexity is minimized.

Lemma 1. *[24] Suppose there exists an (r_1, r_2, p_1, p_2)-sensitive family $\mathcal{H}$. For a list L of size N, let*

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

Then with high probability we can either (a) find an element $\boldsymbol{w}^* \in L$ that satisfies $D(\boldsymbol{v}, \boldsymbol{w}^*) \leq r_2$, or (b) conclude that with high probability, no elements $\boldsymbol{w} \in L$ with $D(\boldsymbol{v}, \boldsymbol{w}) > r_1$ exist, with the following costs:

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

Proof. Let us start with 'good' or nearby vectors, i.e., vectors $\boldsymbol{w} \in L$ at distance at most r_1 from $\boldsymbol{v}$. Taking k-way ANDs, we obtain a collision probability of $p_1^k = (p_1^{k/\rho})^\rho = N^{-\rho}$. Then, taking a t-way OR-composition, the overall collision probability is $p_1^* = 1-(1-p_1^k)^t = 1-(1-N^{-\rho})^{O(N^\rho)} = O(1)$. So the probability that good vectors are found is constant, where the constant depends only on the constant hidden inside $t = O(N^\rho)$.

For 'bad' vectors $\boldsymbol{w}$ at distance at least r_2 from $\boldsymbol{v}$, the k-way AND-composition gives us a collision probability of $p_2^k = N^{-1}$. Taking a t-way OR, we get an overall probability of a collision of $p_2^* = 1-(1-p_2^k)^t = 1-(1-N^{-1})^{O(N^\rho)} = O(N^{\rho-1})$. Since the total length of the list is N, we expect that roughly $N \cdot O(N^{\rho-1}) = O(N^\rho)$ bad vectors $\boldsymbol{w} \in L$ collide with $\boldsymbol{v}$ in at least one of the hash tables.

[3] Note that a mapping h resulting from an OR-composition is strictly not a function and only defines a relation.

Using e.g. the Chernoff bound, with overwhelming probability the exact number of 'bad' colliding vectors will be relatively close to this average.

Summarizing, the costs of preprocessing the data are computing $N \cdot t \cdot k = O(N^{1+\rho} \log_{1/p_2} N)$ hashes; the space complexity of the preprocessed data (t hash tables with N vectors) is $O(N \cdot t) = O(N^{1+\rho})$; the costs of obtaining a list of candidate nearest vectors for $\boldsymbol{v}$ are computing $t = O(N^\rho)$ hashes of $\boldsymbol{v}$ and performing as many table look-ups; and the cost of searching the resulting list of candidates for the right vector is equal to the length of this list, which is at most $O(N^\rho)$. This algorithm succeeds with constant probability, and by changing the constant for t the probability of failure can be made arbitrarily small. □

Although Lemma 1 only shows how to choose k and t to minimize the time complexity, we can also tune k and t so that we use more time and less space. In a way this algorithm can be seen as a generalization of the naive brute-force search solution for finding nearest neighbors, as $k = 0$ and $t = 1$ corresponds to checking the whole list for nearby vectors in linear time and linear space, and increasing both k and t leads to fewer comparisons but a higher cost of computing hashes and checking buckets.

2.5 Angular hashing

Let us now consider actual hash families for the similarity measure D that we are interested in. As argued in the next section, what seems a more natural choice for D than the Euclidean distance is the *angular distance* or *cosine similarity*, defined on $\mathbb{R}^n$ as

$$D(\boldsymbol{v}, \boldsymbol{w}) = \theta(\boldsymbol{v}, \boldsymbol{w}) = \arccos\left(\frac{\boldsymbol{v}^T \boldsymbol{w}}{\|\boldsymbol{v}\| \cdot \|\boldsymbol{w}\|}\right). \tag{2}$$

With this similarity measure, two vectors are 'nearby' if their common angle is small, and 'far apart' if their angle is large. In a sense, this is similar to the Euclidean norm: if two vectors have similar Euclidean norms, their distance is large iff their angular distance is large. For this similarity measure D, the following hash family $\mathcal{H}$ was first described in [16]:

$$\mathcal{H} = \{h_{\boldsymbol{a}} : \boldsymbol{a} \in \mathbb{R}^n, \|\boldsymbol{a}\| = 1\}, \qquad h_{\boldsymbol{a}}(\boldsymbol{v}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \boldsymbol{a}^T \boldsymbol{v} \geq 0; \\ 0 & \text{if } \boldsymbol{a}^T \boldsymbol{v} < 0. \end{cases} \tag{3}$$

Intuitively, the vector $\boldsymbol{a}$ defines a hyperplane (for which $\boldsymbol{a}$ is a normal vector), and $h_{\boldsymbol{a}}$ maps the two regions separated by this hyperplane to different bits.

To see why this is a non-trivial locality-sensitive hash family for the angular distance, consider two vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$. These two vectors lie on a two-dimensional plane passing through the origin, and with probability 1 a hash vector $\boldsymbol{a}$ does not lie on this plane (for $n > 2$). This means that the hyperplane defined by $\boldsymbol{a}$ intersects this plane in some line ℓ. Since $\boldsymbol{a}$ is taken uniformly at random from the unit sphere, the line ℓ has a uniformly random 'direction' in the plane, and maps $\boldsymbol{v}$ and $\boldsymbol{w}$ to different hash values iff ℓ separates $\boldsymbol{v}$ and $\boldsymbol{w}$ in the plane. Therefore the probability that $h(\boldsymbol{v}) \neq h(\boldsymbol{w})$ is directly proportional to their common angle $\theta(\boldsymbol{v}, \boldsymbol{w})$ as follows [16]:

$$\mathbb{P}_{h_{\boldsymbol{a}} \in \mathcal{H}}\big[h_{\boldsymbol{a}}(\boldsymbol{v}) = h_{\boldsymbol{a}}(\boldsymbol{w})\big] = 1 - \frac{\theta(\boldsymbol{v}, \boldsymbol{w})}{\pi}. \tag{4}$$

So for any two angles $\theta_1 < \theta_2$, the family $\mathcal{H}$ is $(\theta_1, \theta_2, 1 - \frac{\theta_1}{\pi}, 1 - \frac{\theta_2}{\pi})$-sensitive. In particular, Charikar's hyperplane hash family is $(\frac{\pi}{3}, \frac{\pi}{2}, \frac{2}{3}, \frac{1}{2})$-sensitive.

To illustrate LSH and in particular the angular LSH method described above, Figure 2 shows how hyperplane hashing might work in a 2-dimensional setting where we have a list $L = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{10}\}$ and a query vector $\boldsymbol{v}$, and we used $k = 2$ hyperplanes in each of t hash tables to find nearby vectors. Preprocessing would consist of computing and storing each of the list vectors in their corresponding hash buckets, which involves k inner product computations for each vector for each hash table. Answering a query can be done by computing a target vector's hash buckets in each hash table and searching the vectors in these hash buckets for a nearest neighbor.

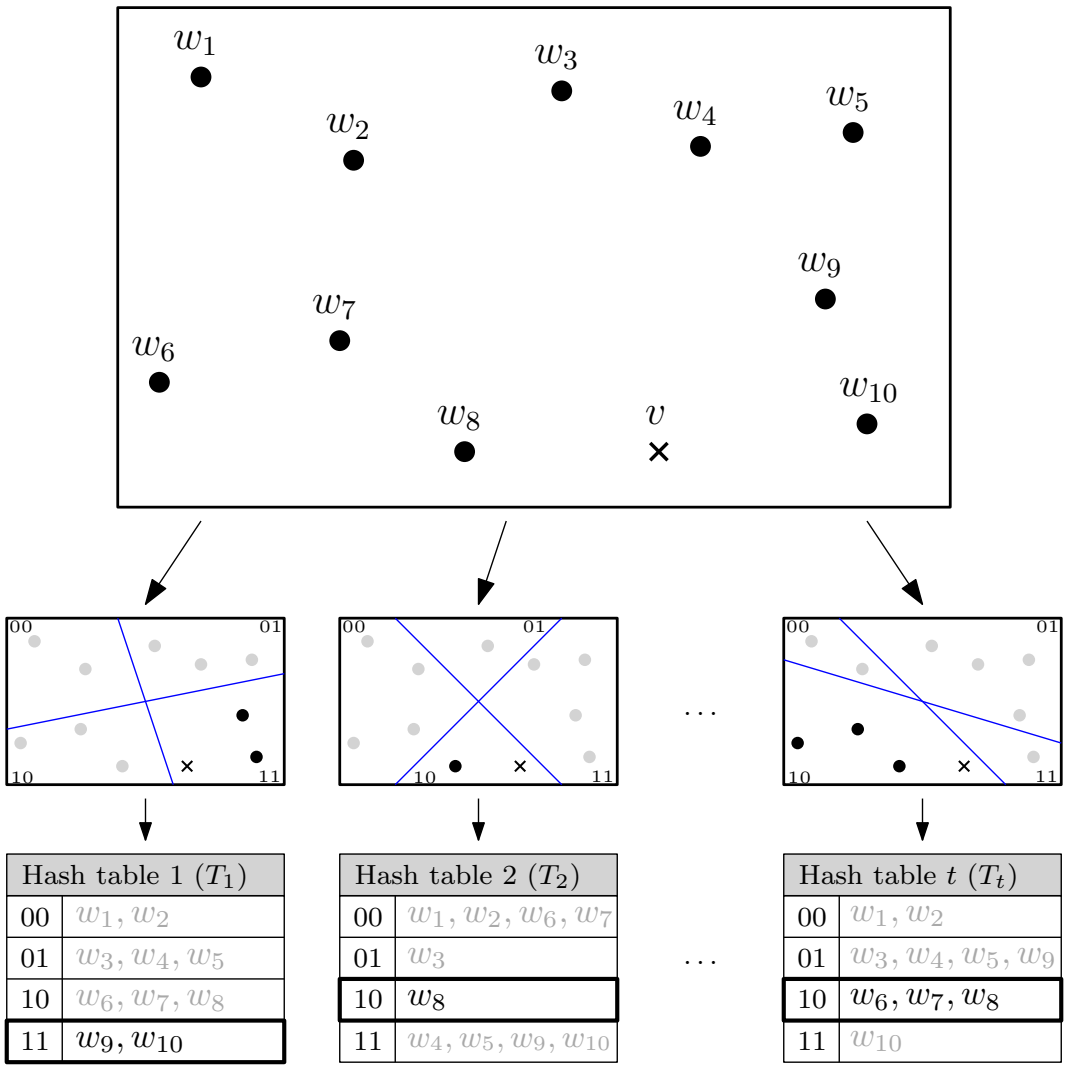


Fig. 2. An example of angular LSH, using $k = 2$ hyperplanes and $2^k = 4$ buckets in each hash table. Given 10 list vectors $L = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{10}\}$ and a target vector $\boldsymbol{v}$, for each of the t hash tables we first compute $\boldsymbol{v}$'s hash value (i.e. compute the region in which it lies), look up vectors which have the same hash value, and compare $\boldsymbol{v}$ to those vectors. Based on these three hash tables, we will find $C = \{\boldsymbol{w}_6, \boldsymbol{w}_7, \boldsymbol{w}_8, \boldsymbol{w}_9, \boldsymbol{w}_{10}\}$ as the set of candidate near neighbors for $\boldsymbol{v}$.

3 From the Nguyen-Vidick sieve to the (NV-)HashSieve

Let us now describe how locality-sensitive hashing can be used to speed up sieving algorithms, and in particular how we can speed up the NV-sieve of Nguyen and Vidick [43] using angular LSH. The same ideas can also be applied to the GaussSieve [40], as illustrated in Section 5.

3.1 The Nguyen-Vidick sieve algorithm

First, recall that the sieving algorithm of Nguyen and Vidick [43] starts with a long list L_0 of reasonably long, randomly sampled lattice vectors $\boldsymbol{v}$ sampled from a discrete Gaussian over the lattice, using e.g. Klein's algorithm [28], and then repeatedly applies a sieve to it to split each list L_m into a list C_{m+1} of centers and a new list L_{m+1} of vectors whose norms are at least a geometric factor $\gamma < 1$ smaller than the maximum norm of the vectors in L_m. After repeatedly applying this sieve, we eventually hope to be left with a short list of very short vectors, which contains the shortest vector.

At the core of Nguyen and Vidick's algorithm lies the sieve that maps L_m onto two sets L_{m+1} and C_{m+1}. A slightly modified and simplified version of this algorithm is described in Algorithm 2. Here we have replaced the on-the-fly generation of the set of centers of the original algorithm description by a predetermined random selection of list points to be used as centers. Note that Nguyen and Vidick's analysis is essentially based on this modified algorithm rather than the on-the-fly generation described

Algorithm 1 Nguyen and Vidick's lattice sieve (without angular LSH)

Require: An input list L_m of $(4/3)^{n/2+o(n)}$ lattice vectors of norm at most $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Ensure: The output list L_{m+1} has size $(4/3)^{n/2+o(n)}$ and only contains lattice vectors of norm at most $\gamma \cdot R$
1: Initialize an empty list L_{m+1}
2: Sample $C_{m+1} \subset L_m \cap \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| \geq \gamma \cdot R\}$ of size $\mathrm{poly}(n) \cdot (4/3)^{n/2}$
3: **for each** $\boldsymbol{v} \in L_m \setminus C_{m+1}$ **do**
4: **if** $\|\boldsymbol{v}\| \leq \gamma R$ **then**
5: Add $\boldsymbol{v}$ to the list L_{m+1}
6: **else**
7: **for each** $\boldsymbol{w} \in C_{m+1}$ **do**
8: **if** $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ **then**
9: Add $\boldsymbol{v} \pm \boldsymbol{w}$ to the list L_{m+1}
10: Continue the loop over "$\boldsymbol{v} \in L_m \setminus C_{m+1}$"

in [43, Algorithm 5], and for reducing the space complexity later on in this paper it is essential that we can select the set of centers in advance. An intuitively simpler variant of this algorithm, which achieves the same exponent and achieves similar speedups with LSH is described in Appendix B.

In Lines 7–10 of Algorithm 1, the Nguyen-Vidick sieve essentially solves the following search problem through a brute-force linear search:

$$\text{Find an element } \boldsymbol{w} \in C_{m+1} \text{ such that } \|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma R. \tag{5}$$

To obtain the estimate $2^{0.415n+o(n)}$ for the time complexity and $2^{0.208n+o(n)}$ for the space complexity, Nguyen and Vidick further let γ approach 1 in their analysis. This means that all vectors with a length significantly shorter than $\gamma \cdot R \approx R$ are automatically added to L_{m+1} in Line 5, and the bottleneck of the time complexity comes from those vectors $\boldsymbol{v}$ with $\gamma R < \|\boldsymbol{v}\| \leq R$, i.e., the vectors $\boldsymbol{v}$ lying in a thin spherical shell of thickness $(1-\gamma)R$. For those vectors $\boldsymbol{v}$ we have $\gamma R \approx \|\boldsymbol{v}\| \approx R$, and for vectors $\boldsymbol{w} \in C_{m+1}$ we also know that $\gamma R \approx \|\boldsymbol{v}\| \approx R$. This implies that the reduction method described in (5) for $\gamma \to 1$ is essentially equivalent to the following angular reduction step:

$$\text{Find an element } \boldsymbol{w} \in C_{m+1} \text{ such that } \theta(\boldsymbol{v}, \pm\boldsymbol{w}) < 60^\circ. \tag{6}$$

In the limiting case of $\gamma \to 1$, the problems (5) and (10) are essentially equivalent.

3.2 The (NV-)HashSieve algorithm

Considering the angular notion of reduction of (10), we can clearly see the connection with nearest neighbor searching for the angular distance or cosine similarity, and how we can fit in angular or hyperplane LSH. Replacing the brute-force list search in the original algorithm with the technique of angular locality-sensitive hashing, we obtain Algorithm 2. Blue lines in Algorithm 2 indicate modifications to the original algorithm. Note that the setup costs of locality-sensitive hashing (building the hash tables) are only paid once, rather than once for each search.

3.3 Relation with leveled sieving

Overall, the crucial modification we introduce is that by using hash tables and looking up vectors to reduce the target vector with in these hash tables, we make the search space smaller; instead of comparing a new vector to *all* vectors in C_{m+1}, we only compare a vector to a much smaller subset of candidates $C \subset C_{m+1}$, which mostly contains good, nearby candidates for reduction, and does not contain many of the 'bad' vectors in C_{m+1} which are not nearby in space.

In a way, this idea is very similar to the technique previously used in two- and three-level sieving [55, 56]. There, the search space of candidate nearby vectors was reduced by partitioning the space into regions, and for each vector storing in which region it lies. In those algorithms, two nearby vectors in adjacent regions are not considered for reductions, which means one needs more vectors to saturate the space (a higher space complexity) but less time to search the list of candidates for nearby vectors (a lower time complexity).

Algorithm 2 Nguyen and Vidick's lattice sieve (with angular LSH)

Require: An input list L_m of $(4/3)^{n/2+o(n)}$ lattice vectors of norm at most $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Ensure: The output list L_{m+1} has size $(4/3)^{n/2+o(n)}$ and only contains lattice vectors of norm at most $\gamma \cdot R$
Initialize an empty list L_{m+1}
Sample $C_{m+1} \subset L_m \cap \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| \geq \gamma \cdot R\}$ of size $\mathrm{poly}(n) \cdot (4/3)^{n/2}$
Initialize k empty hash tables T_i and sample $k \cdot t$ random hash functions $h_{i,j} \in \mathcal{H}$
for each $\boldsymbol{w} \in C_{m+1}$ **do**
    Add $\boldsymbol{w}$ to all k hash tables T_i, to the buckets $T_i[h_i(\boldsymbol{w})]$
for each $\boldsymbol{v} \in L_m \setminus C_{m+1}$ **do**
    if $\|\boldsymbol{v}\| \leq \gamma R$ **then**
        Add $\boldsymbol{v}$ to the list L_{m+1}
    else
        Obtain the set of candidates $C = \bigcup_{i=1}^{t} T_i[h_i(\pm \boldsymbol{v})]$
        for each $\boldsymbol{w} \in C$ **do**
            if $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ **then**
                Add $\boldsymbol{v} \pm \boldsymbol{w}$ to the list L_{m+1}
                Continue the loop over "$\boldsymbol{v} \in L_m \setminus C_{m+1}$"

Two key differences between leveled sieving and our method are (i) the way the partitions of $\mathbb{R}^n$ are chosen (using giant balls in leveled sieving, similar to the Euclidean LSH method of [6], versus using random hyperplanes here); and (ii) the idea of LSH is to guarantee that nearby vectors are still found with high (constant) probability by using many hash tables. The increased space complexity of sieving with LSH comes purely from using many "'rerandomized" hash tables, and not from an increased list size as a result of missing many nearby vectors.

4 Theoretical results

4.1 High-dimensional intuition

To estimate the complexity of the angular LSH-based lattice sieve, we will make use of the following heuristic assumption previously described in [43]:

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

Note that under this assumption, in high dimensions angles close to 90° are much more likely to occur between pairs of vectors than smaller angles. So one might guess that for two vectors $\boldsymbol{v}, \boldsymbol{w} \in L_m$, with high probability their angle is very close to 90°. On the other hand, nearby vectors $\boldsymbol{w} \in L$ that can reduce our target vector $\boldsymbol{v}$ always have an angle less than 60° with $\boldsymbol{w}$, and by similar arguments we expect this angle to always be close to 60° and not much less than this. Under the extreme assumption that all 'random angles' between vectors $\boldsymbol{v}, \boldsymbol{w}$ that do not satisfy the condition $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ are *exactly* 90° (and angles of nearby pairs of vectors are at most 60°), we obtain the following preliminary estimate for the costs of the algorithm.

Proposition 1. *Assuming that non-reducing vectors are always pairwise orthogonal, the NV-sieve with angular LSH with parameters $k = 0.2075n + o(n)$ and $t = 2^{0.1214n+o(n)}$ heuristically solves SVP in time and space $2^{0.3289n+o(n)}$. By varying the values k and t, we further obtain the trade-off between the space and time complexities indicated by the dashed line in Figure 1.*

Proof. If all 'random angles' are 90°, then we can simply let $\theta_1 = \frac{\pi}{3}$ and $\theta_2 = \frac{\pi}{2}$ and use the hash family described in Section 2.5 with $p_1 = \frac{2}{3}$ and $p_2 = \frac{1}{2}$. Applying Lemma 1, we can perform a single search in time $N^\rho = 2^{0.1214n+o(n)}$ using $t = 2^{0.1214n+o(n)}$ hash tables, where $\rho = \frac{\log(1/p_1)}{\log(1/p_2)} = \log_2(\frac{3}{2}) \approx 0.585$. Since we need to perform these searches $\tilde{O}(N)$ times, and we need to repeat the whole sieving procedure $\mathrm{poly}(n)$ times, the time complexity is of the order $\tilde{O}(N^{1+\rho}) = 2^{0.3289n+o(n)}$. The space complexity is dominated by having to store all N vectors in $t = O(N^\rho)$ hash tables, leading to a space complexity of $\tilde{O}(N^{1+\rho}) = 2^{0.3289n+o(n)}$ as well. □

Algorithm 3 Nguyen and Vidick's lattice sieve (with angular LSH, space-efficient)

Require: An input list L_m of $(4/3)^{n/2+o(n)}$ lattice vectors of norm at most $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Ensure: The output list L_{m+1} has size $(4/3)^{n/2+o(n)}$ and only contains lattice vectors of norm at most $\gamma \cdot R$

```
Initialize an empty list L_{m+1}
Sample C_{m+1} ⊂ L_m ∩ {x ∈ R^n : ‖x‖ ≥ γ·R} of size poly(n)·(4/3)^{n/2}
for each i ∈ {1,...,t} do
    Initialize an empty hash table T and sample k random hash functions h_{i,j} ∈ H
    for each w ∈ C_{m+1} do
        Add w to the hash table T, to bucket T[h_i(w)]
    for each v ∈ L_m \ C_{m+1} do
        if ‖v‖ ≤ γR then
            Add v to the list L_{m+1}
        else
            Obtain the (partial) set of candidates C = T[h_i(±v)]
            for each w ∈ C do
                Reduce v with w
                if v has changed then
                    Add v to the list L_{m+1}
                    Continue the loop over "v ∈ L_m \ C_{m+1}"
```

4.2 Heuristically solving SVP in time and space $2^{0.3366n+o(n)}$

Of course, in practice not all random 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. The following theorem follows from this analysis and shows how to choose the parameters to optimize the asymptotic time complexity. A proof of Theorem 1 as well as details on the resulting time/memory trade-off can be found in Appendix A.

Theorem 1. *The Nguyen-Vidick sieve with angular LSH with parameters*

$$k = 0.2206n + o(n), \qquad t = 2^{0.1290n+o(n)}, \tag{7}$$

and $\gamma \to 1$ heuristically solves SVP in time and space $2^{0.3366n+o(n)}$. Tuning k and t differently, we further obtain the trade-off indicated by the solid blue line in Figure 1.

Note that the optimized values in Theorem 1 and Proposition 1, and the associated curves in Figure 1 are very similar. The simple estimate based on the intuition that in high dimensions "everything is orthogonal" is not far off. Also note that as there are $2^k \gg N$ hash buckets in each table, the space complexity would be slightly higher ($2^{0.3496n+o(n)}$) if hash tables are simply stored as arrays. To achieve the minimum asymptotic time and space complexities, one should only store non-empty buckets in memory as asymptotically most buckets are empty.

4.3 Heuristically solving SVP in time $2^{0.3366n+o(n)}$ and space $2^{0.2075n+o(n)}$

For the Nguyen-Vidick sieve [43], we can in fact process the hash tables sequentially and eliminate the need of storing exponentially many hash tables in memory at the same time. The simple but crucial modification that we can make to this algorithm, in similar fashion to the filtering idea described in [12], is that we process the hash tables one by one; we first construct the first hash table, add all vectors in C_{m+1} to this hash table, and look for short difference vectors to add to L_{m+1}. By then deleting this hash table from memory and building a new hash table (repeating this $t = 2^{0.13n+o(n)}$ times) we keep adding more and more vectors to L_{m+1} until finally we will again have found the exact same set of short vectors for the next iteration. In this case however we never stored all hash tables in memory at the same time, and the memory increase compared to the NV-sieve is asymptotically negligible. This modification leads to the algorithm described in Algorithm 3, and the previous discussion also immediately leads to the following result.

Algorithm 4 The GaussSieve algorithm (without angular LSH)

```
1: Initialize an empty list L and an empty stack S
2: repeat
3:     Get a vector v from the stack (or sample a new one if S = ∅)
4:     for each w ∈ L do
5:         Reduce v with w
6:         Reduce w with v
7:         if w has changed then
8:             Remove w from the list L
9:             Add w to the stack S (unless w = 0)
10:    if v has changed then
11:        Add v to the stack S (unless v = 0)
12:    else
13:        Add v to the list L
14: until v is a shortest vector
```

Theorem 2. *The space-efficient Nguyen-Vidick sieve with angular LSH with parameters*

$$k = 0.2206n + o(n), \qquad t = 2^{0.1290n+o(n)}, \tag{8}$$

and $\gamma \to 1$ heuristically solves SVP in time $2^{0.3366n+o(n)}$ and space $2^{0.2075n+o(n)}$. These complexities are indicated by the left-most blue point in Figure 1.

Note that this choice of parameters k, t still balances the costs of computing hashes and comparing vectors; the fact that the blue point in Figure 1 now does not lie on the "Time = Space"-diagonal does not mean we can further reduce the time complexity by increasing t.

5 From the GaussSieve to the HashSieve

Let us next describe how locality-sensitive hashing can also be used to speed up the GaussSieve of Micciancio and Voulgaris [40]. This algorithm is our main focus for practical applications, since it seems to be the fastest and most space-efficient sieving algorithm to date, which is further motivated by the extensive attention it has received in recent years [15, 19, 25, 29, 35, 36, 42, 49, 50] and by the fact that the current highest sieving records in the SVP challenge database were obtained using (a modification of) the GaussSieve [29, 51].

5.1 The GaussSieve algorithm

A simplified version of the GaussSieve algorithm of Micciancio and Voulgaris is described in Algorithm 4. The algorithm iteratively builds a longer and longer list L of lattice vectors, occasionally reducing the lengths of list vectors in the process, until at some point this list L contains a shortest vector. Vectors are again sampled from a discrete Gaussian over the lattice, using e.g. the sampling algorithm of Klein [28, 40], or popped from the stack if the stack is non-empty. If list vectors are modified or newly sampled vectors are reduced, they are pushed to the stack.

In the GaussSieve, the reductions in Lines 5 and 6 follow the rule:

$$\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2 : \quad \text{if } \|\boldsymbol{u}_1 \pm \boldsymbol{u}_2\| < \|\boldsymbol{u}_1\| \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2. \tag{9}$$

Throughout the execution of the algorithm, the list L is always pairwise reduced w.r.t. (9), i.e., $\|\boldsymbol{w}_1 \pm \boldsymbol{w}_2\| \geq \max\{\|\boldsymbol{w}_1\|, \|\boldsymbol{w}_2\|\}$ for all $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$. This implies that two list vectors $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ always have an angle of at least 60°; otherwise one of them would have been used to reduce the other before being added to the list. Since all angles between list vectors are always at least 60°, the size of L is bounded by the *kissing constant* in dimension n: the maximum number of vectors in $\mathbb{R}^n$ one can find such that any two vectors have an angle of at least 60°. Bounds and conjectures on the kissing constant in high dimensions lead us to believe that the size of the list L will not exceed $2^{0.2075n+o(n)}$ [17].

While the space complexity of the GaussSieve is reasonably well understood, there are no proven bounds on the time complexity of this algorithm. One might estimate that the time complexity is

Algorithm 5 The GaussSieve algorithm (with angular LSH) – The HashSieve algorithm

1: Initialize an empty list L and an empty stack S
2: Initialize t empty hash tables T_i and sample $k \cdot t$ random hash functions $h_{i,j} \in \mathcal{H}$
3: **repeat**
4: Get a vector $\boldsymbol{v}$ from the stack (or sample a new one if $S = \emptyset$)
5: Obtain the set of candidates $C = \bigcup_{i=1}^{t} T_i[h_i(\boldsymbol{v})]$
6: **for each** $\boldsymbol{w} \in C$ **do**
7: Reduce $\boldsymbol{v}$ with $\boldsymbol{w}$
8: Reduce $\boldsymbol{w}$ with $\boldsymbol{v}$
9: **if** $\boldsymbol{w}$ has changed **then**
10: Remove $\boldsymbol{w}$ from the list L
11: Remove $\boldsymbol{w}$ from all t hash tables T_i
12: Add $\boldsymbol{w}$ to the stack S (unless $\boldsymbol{w} = \boldsymbol{0}$)
13: **if** $\boldsymbol{v}$ has changed **then**
14: Add $\boldsymbol{v}$ to the stack S (unless $\boldsymbol{v} = \boldsymbol{0}$)
15: **else**
16: Add $\boldsymbol{v}$ to the list L
17: Add $\boldsymbol{v}$ to all t hash tables T_i
18: **until** $\boldsymbol{v}$ is a shortest vector

determined by the double loop over L: at any time each pair of vectors $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ was compared at least once to see if one could reduce the other, so the time complexity is at least quadratic in $|L|$. The algorithm further seems to display a similar asymptotic behavior as the NV-sieve in experiments [40, 43], for which the asymptotic time complexity is heuristically known to be quadratic in $|L|$, i.e., of the order $2^{0.415n+o(n)}$. One might therefore conjecture that the GaussSieve also has a time complexity of $2^{0.415n+o(n)}$, which closely matches previous experiments with the GaussSieve in high dimensions [29].

Since these heuristic bounds on the space and time complexities are only based on the fact that each pair of vectors $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ has an angle of at least $60°$, the same heuristics apply to any reduction method that guarantees that angles between vectors in L are at least $60°$. In particular, if we reduce vectors only if their angle is at most $60°$ using the following rule:

$$\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2 : \quad \text{if } \theta(\boldsymbol{u}_1, \pm\boldsymbol{u}_2) < 60° \text{ and } \|\boldsymbol{u}_1\| \geq \|\boldsymbol{u}_2\| \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2, \tag{10}$$

then we expect the same heuristic bounds on the time and space complexities to apply. More precisely, the list size would again be bounded by $2^{0.208n+o(n)}$, and the time complexity may again be estimated to be of the order $2^{0.415n+o(n)}$. Basic experiments show that, although with this notion of reduction the list size increases, this factor indeed appears to be sub-exponential in n.

5.2 The HashSieve algorithm

Replacing the stronger notion of reduction of (9) by the weaker one of (10), we can clearly see the connection with angular hashing. Considering the GaussSieve with angular reductions, we are repeatedly sampling new target vectors $\boldsymbol{v}$ (with each time almost the same list L), and each time we are looking for vectors $\boldsymbol{w} \in L$ whose angle with $\boldsymbol{v}$ is at most $60°$. Replacing the brute-force list search in the original algorithm with the technique of angular locality-sensitive hashing, we obtain Algorithm 5. Blue lines in Algorithm 5 indicate modifications to the GaussSieve. Note that the setup costs of locality-sensitive hashing are again spread out over the various iterations; at each iteration we only update the parts of the hash tables that were affected by updating L. This means that we only pay the setup costs of locality-sensitive hashing once, rather than once for each search.

Finally, there is no point in skipping potential reductions in Lines 7 and 8. So while for our intuition and for the theoretical motivation we may consider the case where the reductions are based on (10), in practice we will again reduce vectors based on (9). This algorithm is illustrated in Figure 2.

5.3 Reducing the space complexity with probing

For the Nguyen-Vidick sieve, recall that we could process hash tables sequentially rather than in parallel, to prevent getting an increased space complexity. For the GaussSieve the same trick does not

seem to apply, and we only obtain a space/time trade-off similar to Theorem 1. As the practicability of sieving seems bounded both by the required amounts of time and memory, this trade-off may only move the problem from one point to another. Being able to handle the increased memory complexity is crucial to making this method practical in higher dimensions.

To decrease the memory requirement of the hash tables, Panigraphy [44] suggested that instead of using many hash tables and checking only one hash bucket in each table for candidate nearby vectors, one could also check several hash buckets in each hash table for nearby vectors, and use fewer hash tables overall to get a similar quality for the list of candidates, using significantly less memory.

Construction. To illustrate how this method might work, consider the following modification to the HashSieve algorithm. In each hash table T_i, instead of only checking the bucket labeled $h_i(\boldsymbol{v}) \in \{0,1\}^k$ for candidates, we also check buckets labeled $h_i(\boldsymbol{v}) \oplus \boldsymbol{e}_j$ for all $j \in [k]$, where $\boldsymbol{e}_j$ is the jth unit vector in k dimensions and $\oplus$ represents component-wise addition in $\mathbb{Z}_2$ (bitwise XOR). In other words, we now consider a vector $\boldsymbol{w} \in L$ a candidate iff it is separated from $\boldsymbol{v}$ by at most one of the random hyperplanes defined by the hash vectors $\boldsymbol{a}_{i,j}$. This means that in each table we now check $k+1$ hash buckets, instead of only one bucket.

Example. To make this construction even more explicit, consider the following example. Suppose we have a vector $\boldsymbol{v}$ and $h_i(\boldsymbol{v}) = (h_1(\boldsymbol{v}), h_2(\boldsymbol{v}), h_3(\boldsymbol{v}), h_4(\boldsymbol{v}), h_5(\boldsymbol{v})) = (0,0,1,0,1)$. Now the bucket most likely to contain nearby vectors is the bucket labeled $(0,0,1,0,1)$, but also those buckets labeled $(1,0,1,0,1), (0,1,1,0,1), (0,0,0,0,1), (0,0,1,1,1), (0,0,1,0,0)$ are quite likely to contain nearby vectors; these buckets contain vectors in adjacent hash regions with four equal hash values and only one different hash value, and one hyperplane separating these vectors from our target vector $\boldsymbol{v}$. Also checking these buckets may lead to finding more nearby vectors, while we will still only check a small fraction of all vectors spread out over the table.

Analysis. To analyze the effect of this modification on the algorithm, let us again make the simplifying assumption that non-reduced vectors have an angle of at most 60° with $\boldsymbol{v}$, and reduced vectors have an angle of exactly 90° with $\boldsymbol{v}$. For non-reduced vectors, the probability that none of the hyperplanes separate $\boldsymbol{v}$ and $\boldsymbol{w}$ is $(\frac{2}{3})^k$, and the probability that *at most one* of the hyperplanes separates these vectors is $(\frac{2}{3})^k + k(\frac{2}{3})^{k-1}(\frac{1}{3})^1 = (\frac{2}{3})^k\left[1+\frac{k}{2}\right]$. Similarly, for reduced vectors the probability of two vectors landing in the same bucket is $(\frac{1}{2})^k$, and the probability of landing in buckets differing in at most one bit is $(\frac{1}{2})^k[1+k]$. Comparing the probabilities of finding given vectors with and without this technique of *probing* multiple buckets, we see that with probing:

- The probability of finding a nearby vector increases by a factor $1+\frac{k}{2}$.
- The probability of finding a distant vector increases by a factor $1+k$.

Note that an increase in the probability of finding nearby vectors in a hash table by a factor α roughly translates to a decrease of t by a factor α, which is motivated by the approximations $1-(1-\alpha p^k)^{t/\alpha} \approx \frac{t}{\alpha}\cdot\alpha p^k = t\cdot p^k \approx 1-(1-p^k)^t$. Therefore, using this technique of probing, we can use the same value k as before, but a smaller value $t_1 = \frac{t}{1+k/2}$ now suffices to still find nearby vectors with high probability. With these values of k and t_1, the probability of finding faraway vectors increases by roughly a factor $\frac{1+k}{1+k/2} < 2$, thus leading to up to 2 times more comparisons overall, and an increase in the time complexity by a factor less than 2. But more importantly, the memory requirement of the hash tables decreases by a factor $1+\frac{k}{2} = O(n)$. To illustrate the possible impact of probing: in dimension 80 we have $k \approx 18$, and so probing might lead to a loss of a factor 2 in the time complexity, and a gain a factor 10 in the memory requirement.

Multiprobe. The procedure of probing adjacent buckets (buckets at Hamming distance 1) can trivially be generalized to considering all buckets with labels that differ from the hash value $h_i(\boldsymbol{v}) \in \{0,1\}^k$ in at most $0 \le \ell \le k$ bits. For $\ell = O(1)$ and large n, this implies a reduction in the number of hash tables (and the space complexity) by a factor $1+\frac{1}{2}k+\frac{1}{4}\binom{k}{2}+\cdots+\frac{1}{2^\ell}\binom{k}{\ell} = O(n^\ell)$ and an increase in the time complexity by less than a factor $2^\ell = O(1)$. For instance, in dimension 120, without probing we have $(k,t) \approx (26, 45700)$; with one level of probing we have $(k,t_1) \approx (26, 3200)$; and with two levels of

probing we have $(k, t_2) \approx (26, 450)$. Using two levels of probing, in dimension 120 the space complexity of the hash tables is reduced by a factor more than 100, at the cost of a factor less than 4 increase of the overall time complexity.

Besides using multiple levels of probing and brute-forcing all buckets at each level, one could also consider more sophisticated ways of choosing which buckets to check for candidates. If $\boldsymbol{a}_{i,j}^T \boldsymbol{w} \approx 0$ then it makes more sense to consider the bucket labeled $h_i(\boldsymbol{v}) \oplus \boldsymbol{e}_j$ than if $\boldsymbol{a}_{i,j}^T \boldsymbol{v} \gg 0$ or $\boldsymbol{a}_{i,j}^T \boldsymbol{v} \ll 0$. In other words, if a vector $\boldsymbol{v}$ lies close to the hyperplane defined by $\boldsymbol{a}_{i,j}$, it makes more sense to consider vectors on the other side of the hyperplane as well, than if $\boldsymbol{v}$ lies far away from the hyperplane. Algorithmically speaking, given $\boldsymbol{v}$ and any bucket $\boldsymbol{b} \in \{0,1\}^k$, one could for instance compute the probability that a reducing vector is in this bucket labeled $\boldsymbol{b}$, and only check those buckets with the highest probabilities of containing nearby vectors. For more details, see e.g. [34].

6 Practical results

6.1 Experimental results in moderate dimensions

To verify the claimed speedups, we implemented both the GaussSieve and the GaussSieve-based HashSieve to try to compare the asymptotic trends of these algorithms. For implementing the HashSieve, we note that we can use various simple tweaks to further improve the algorithm's performance. These include:

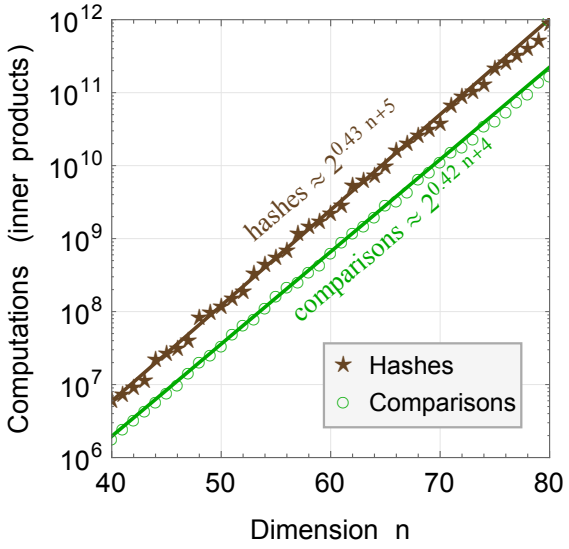
(a) With the HashSieve, maintaining a list L is no longer needed.
(b) Instead of making a list of candidates, we go through the hash tables one by one, checking if collisions in this table lead to reductions. If a nearby vector is found early on, this may save up to $t \cdot k$ hash computations.
(c) As $h_i(-\boldsymbol{v}) = -h_i(\boldsymbol{v})$ the hash of $-\boldsymbol{v}$ can be computed for free from $h_i(\boldsymbol{v})$.
(d) Instead of comparing $\pm\boldsymbol{v}$ to all candidate vectors $\boldsymbol{w}$, we only compare $+\boldsymbol{v}$ to the vectors in the bucket $h_i(\boldsymbol{v})$ and $-\boldsymbol{v}$ to the vectors in the bucket labeled $-h_i(\boldsymbol{v})$. This further reduces the number of comparisons by a factor 2 compared to the GaussSieve, where both comparisons are done for each potential reduction.
(e) For choosing vectors $\boldsymbol{a}_{i,j}$ to use for the hash functions h_i, there is no reason to assume that drawing these vectors from a specific, sufficiently large random subset of the unit sphere would lead to substantially different results. In particular, using sparse vectors $\boldsymbol{a}_{i,j}$ makes computing hash values significantly cheaper, while retaining the same performance [1, 32]. Our experiments indicated that even if all vectors $\boldsymbol{a}_{i,j}$ have only two equal non-zero entries, the algorithm still finds the shortest vector in (roughly) the same number of iterations as with random vectors $\boldsymbol{a}_{i,j}$.
(f) We should not store the actual vectors, but only pointers to vectors in each hash table T_i. This means that compared to the GaussSieve, the space complexity roughly increases from $O(N \cdot n)$ to $O(N \cdot n + N \cdot t)$ instead of $O(N \cdot n \cdot t)$, i.e., an asymptotic increase of a factor t/n rather than t.

With these tweaks, we performed several experiments of finding shortest vectors using the lattices of the SVP challenge [51]. We generated lattice bases for different seeds and different dimensions using the SVP challenge generator, used NTL [54] to preprocess the bases (LLL reduction with $\delta = 0.99$), and we then used our implementations of the GaussSieve and HashSieve to obtain these results. For the HashSieve we chose k and t by rounding the theoretical estimates of Theorem 1 to the nearest integers, i.e., $k = \lfloor 0.2206n \rceil$ and $t = \lfloor 2^{0.1290n} \rceil$ (see Figure 3a). Note that clearly there are ways to further speed up both the GaussSieve and the HashSieve, using e.g. better preprocessing, vectorized code, parallel implementations, optimized samplers, etc. The purpose of our experiments is only to obtain a fair comparison of the two algorithms and to estimate and compare the asymptotic behaviors of these algorithms. Details on a more optimized implementation of the HashSieve are given in [37].

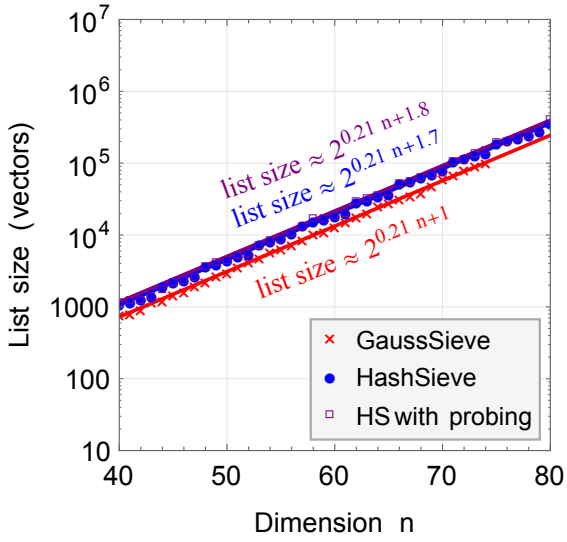
Computations. Figure 3b shows the number of inner products computed by the HashSieve for comparing vectors and for computing hashes. We have chosen k and t so that the total time for each of these operations is roughly balanced, and indeed this seems to be the case. The total number of inner products for hashing seems to be a constant factor higher than the total number of inner products

Dimension (n)	40	45	50	55	60	65	70	75	80	85	90	95	100
Hash length (k)	9	10	11	12	13	14	15	17	18	19	20	21	22
Hash tables...													
...without probing (t)	36	56	87	137	214	334	523	817	1278	1999	3126	4888	7643
...with 1-level probing (t_1)	7	9	13	20	29	42	62	86	128	190	284	425	637
...with 2-level probing (t_2)	2	3	4	6	8	11	15	19	26	38	53	76	110

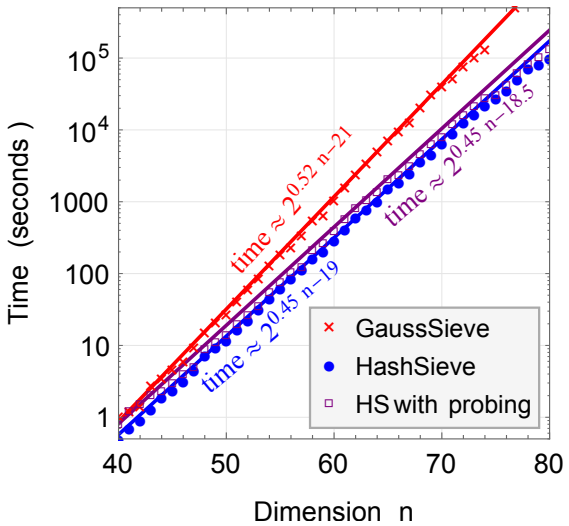
(a) Parameters in the HashSieve (the theoretical leading terms, rounded to the nearest integer), without probing (k, t), with one level of probing (k, t_1), and with two levels of probing (k, t_2), for various dimensions n.



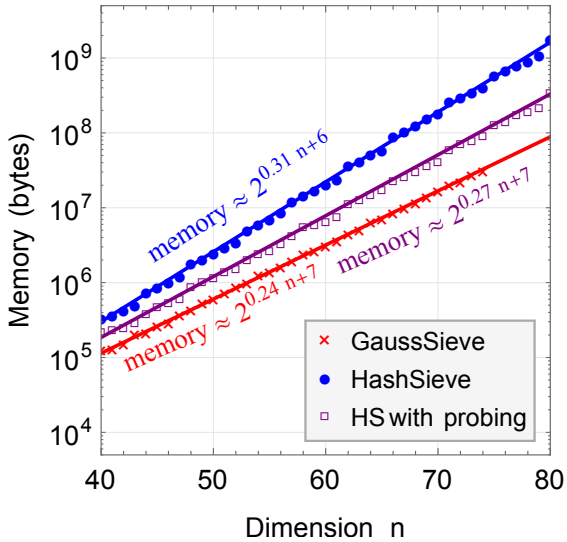
(b) HashSieve computations (without probing)



(c) Maximum list sizes during execution



(d) Time complexities



(e) Space complexities

Fig. 3. Experimental data for the GaussSieve and the HashSieve (with/without probing). Markers indicate experimental data, while lines and labels represent least-squares fits based on this data. The experimental data considers the GaussSieve (red), the HashSieve (blue), and the HashSieve with one level of probing (purple).
Figure 3a details the parameters k and t used in the experiments, and what they might be in higher dimensions. Figure 3b shows the time spent on hashing and comparing vectors in the HashSieve. Figure 3c confirms our intuition that if we miss a small fraction of the reducing vectors, the list size increases by a small factor as well. Figure 3d compares the time complexities of the algorithms, confirming our theoretical analysis of a speedup of roughly $2^{0.07n}$ over the GaussSieve. Figure 3e illustrates the space requirements of each algorithm. Note that probing decreases the required memory at the cost of a small increase in the time. Also note that the step-wise behavior of some curves is explained by the fact that k is small but integral, and increases by 1 only once every four/five dimensions.

computed for comparing vectors, which may also be desirable, as hashing is significantly cheaper than comparing vectors using sparse hash vectors. Tuning the parameters differently may slightly change this ratio.

List sizes. In the analysis, we assumed that if reductions are missed with a constant probability, then the list size also increases by a constant factor. Figure 3c seems to support this intuition, as indeed the list sizes in the HashSieve seem to be a (small) constant factor larger than in the GaussSieve.

Time complexities. Figure 3d compares the timings of the GaussSieve and HashSieve on a single core of a Dell Optiplex 780, which has a processor speed of 2.66 GHz. Theoretically, we expect to achieve a speedup of roughly $2^{0.078n}$ for each list search, and in practice we see that the asymptotic speedup of the HashSieve over the GaussSieve is close to $2^{0.07n}$ using a least-squares fit.

Note that the coefficients in the least-squares fits for the time complexities of the GaussSieve and HashSieve are higher than theory suggests, which is in fact consistent with previous experiments in low dimensions [19, 25, 35, 36, 40]. This phenomenon seems to be caused purely by the low dimensionality of our experiments. Figure 3d shows that in higher dimensions, the points start to deviate from the straight line, with a better scaling of the time complexity in higher dimensions. High-dimensional experiments of the GaussSieve ($80 \leq n \leq 100$) and the HashSieve ($86 \leq n \leq 96$) demonstrated that these algorithms start following the expected trends of $2^{0.42n+o(n)}$ (GaussSieve) and $2^{0.34n+o(n)}$ (HashSieve) as n gets larger [29, 37]. In high dimensions we therefore expect the coefficient 0.3366 to be accurate. For more details, see [37].

Space complexities. Figure 3e illustrates the experimental space complexities of the tested algorithms for various dimensions. For the GaussSieve, the total space complexity is dominated by the memory required to store the list L. In our experiments we stored each vector coordinate in a register of 4 bytes, and since each vector has n entries, this leads to a total space complexity for the GaussSieve of roughly $4nN$ bytes. For the HashSieve the asymptotic space complexity is significantly higher, but recall that in our hash tables we only store pointers to vectors, which may also be only 4 bytes each. For the HashSieve, we estimate the total space complexity as $4nN + 4tN \sim 4tN$ bytes, i.e., roughly a factor $\frac{t}{n} \approx 2^{0.1290n}/n$ higher than the space complexity of the GaussSieve. Using probing, the memory requirement is further reduced by a significant amount, at the cost of a small increase in the time complexity (Figure 3d).

6.2 High-dimensional extrapolations

As explained at the start of this section, the experiments in Section 6.1 are aimed at verifying the heuristic analysis and at establishing trends which hold regardless of the amount of optimization of the code, the quality of preprocessing of the input basis, the amount of parallelization etc. However, the linear estimates in Figure 3 may not be accurate. For instance, the time complexities of the GaussSieve and HashSieve seem to scale better in higher dimensions; the time complexities may well be $2^{0.415n+o(n)}$ and $2^{0.337n+o(n)}$ respectively, but the contribution of the $o(n)$ only starts to fade away for large n. To get a better feeling of the actual time complexities in high dimensions, one would have to run these algorithms in higher dimensions. In recent work, Mariano et al. [37] showed that the HashSieve can be parallelized in a similar fashion as the GaussSieve [35]. With better preprocessing and optimized code (but without probing), Mariano et al. were able to solve SVP in dimensions up to 96 in less than one day on one machine using the HashSieve[4]. Based on experiments in dimensions 86 up to 96, they further estimated the time complexity to lie between $2^{0.32n-15}$ and $2^{0.33n-16}$, which is close to the theoretical estimate $2^{0.3366n+o(n)}$. So although the points in Figure 3d almost seem to lie on a line with a different leading constant, these leading constants should not be taken for granted for high-dimensional extrapolations; the theoretical estimate $2^{0.3366n+o(n)}$ seems more accurate.

[4] At the time of writing, Mariano et al.'s highest SVP challenge records obtained using the HashSieve are in dimension 107, using five days on one multi-core machine.

Finally, let us try to estimate the highest practical dimension n in which the HashSieve may be able to solve SVP right now. The current highest dimension that was attacked using the GaussSieve is $n = 116$, for which 32GB RAM and about 2 core years were needed [29]. Assuming the theoretical estimates for the GaussSieve ($2^{0.4150n+o(n)}$) and HashSieve ($2^{0.3366n+o(n)}$) are accurate, and assuming there is a constant overhead of approximately 2^2 of the HashSieve compared to the GaussSieve (based on the exponents in Figure 3d), we might estimate the time complexities of the GaussSieve and HashSieve to be $G(n) = 2^{0.4150n+C}$ and $H(n) = 2^{0.3366n+C+2}$ respectively. To solve SVP in the same dimension $n = 116$, we therefore expect to use a factor $G(116)/H(116) \approx 137$ less time using the HashSieve, or five core days on the same machine. With approximately two core years, we may further be able to solve SVP in dimension 138 using the HashSieve, which would place sieving near the very top of the SVP hall of fame [51]. This does not take into account the space complexity though, which at this point may have increased to several TBs. Several levels of probing may significantly reduce the required amount of RAM, but further experiments have to be conducted to see how practical the HashSieve is in high dimensions. As in high dimensions the space requirement also becomes an issue, studying the memory-efficient NV-sieve-based HashSieve (with space complexity $2^{0.2075n+o(n)}$) may be an interesting topic for future work.

Acknowledgments

The author is grateful to Meilof Veeningen and Niels de Vreede for their help and advice with implementing the algorithm. The author further thanks the anonymous reviewers, Daniel J. Bernstein, Marleen Kooiman, Tanja Lange, Artur Mariano, Joop van de Pol, and Benne de Weger for their valuable suggestions and comments. The author thanks Michele Mosca for funding a research visit to Waterloo to collaborate on lattices and quantum algorithms, which later inspired work on this topic, and the author thanks Stacey Jeffery, Michele Mosca, Joop van de Pol, and John M. Schanck for valuable discussions there.

References

1. Achlioptas, D.: Database-friendly random projections. In: PODS (2001)
2. 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)
3. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: STOC, pp. 99–108, (1996)
4. Ajtai, M.: The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In: STOC, pp. 10–19 (1998)
5. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: STOC, pp. 601–610 (2001)
6. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS, pp. 459–468 (2006)
7. Andoni, A., Indyk, P., Nguyen, H. L., Razenshteyn, I.: Beyond locality-sensitive hashing. In: SODA, pp. 1018–1028 (2014)
8. Andoni, A., Razenshteyn, I.: Optimal data-dependent hashing for approximate near neighbors. In: STOC, pp. 793–801 (2015)
9. Andoni, A., Indyk, P., Kapralov, M., Laarhoven, T., Razenshteyn, I., Schmidt, L.: Practical and optimal LSH for angular distance. Preprint (2015)
10. Becker, A., Gama, N., Joux, A.: A sieve algorithm based on overlattices. In: ANTS, pp. 49–70 (2014)
11. Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. Preprint (2015)
12. 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)
13. Becker, A., Laarhoven, T.: Efficient sieving in (ideal) lattices using cross-polytopic LSH. Preprint (2015)
14. Bernstein, D. J., Buchmann, J., Dahmen, E.: Post-quantum cryptography (2009)
15. Bos, J. W., Naehrig, M., van de Pol, J.: Sieving for shortest vectors in ideal lattices: a practical perspective. Cryptology ePrint Archive, Report 2014/880 (2014)
16. Charikar, M. S.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388 (2002)
17. Conway, J. H., Sloane, N. J. A.: Sphere packings, lattices and groups (1999)
18. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice. Mathematics of Computation 44(170), pp. 463–471 (1985)
19. 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)

20. Gama, N., Nguyen, P. Q., Regev, O.: Lattice enumeration using extreme pruning. In: EUROCRYPT, pp. 257–278 (2010)
21. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC (2009)
22. Hanrot, G., Pujol, X., Stehlé, D.: Algorithms for the shortest and closest lattice vector problems. In: IWCC, pp. 159–190 (2011)
23. Hoffstein, J., Pipher, J., Silverman, J. H.: NTRU: A ring-based public key cryptosystem. In: ANTS, pp. 267–288 (1998)
24. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
25. Ishiguro, T., Kiyomoto, S., Miyake, Y., Takagi, T.: Parallel Gauss Sieve algorithm: solving the SVP challenge over a 128-dimensional ideal lattice. In: PKC (2014)
26. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: STOC, pp. 193–206 (1983)
27. Khot, S.: Hardness of approximating the shortest vector problem in lattices. In: FOCS, pp. 126–135 (2004)
28. Klein, P.: Finding the closest lattice vector when it's unusually close. In: SODA, pp. 937–941 (2000)
29. Kleinjung, T.: Private communication (2014)
30. Laarhoven, T., de Weger, B.: Sieving for shortest vectors in lattices using Euclidean and spherical locality-sensitive hashing. In: LATINCRYPT (2015)
31. Lenstra, A. K., Lenstra, H. W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), pp. 515–534 (1982)
32. Li, P., Hastie, T. J., Church, K. W.: Very sparse random projections. In: KDD, pp. 287–296 (2006)
33. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: CT-RSA, pp. 319–339 (2011)
34. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: VLDB, pp. 950–961 (2007)
35. Mariano, A., Timnat, S., Bischof, C.: Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In: SBAC-PAD (2014)
36. Mariano, A., Dagdelen, Ö., Bischof, C.: A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In: APCI&E (2014)
37. Mariano, A., Laarhoven, T., Bischof, C.: Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In: ICPP (2015)
38. May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: EUROCRYPT, pp. 203–228 (2015)
39. Micciancio, D., Voulgaris, P.: A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In: STOC (2010)
40. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA, pp. 1468–1480 (2010)
41. Micciancio, D., Walter, M.: Fast lattice point enumeration with minimal overhead. In: SODA, pp. 276–294 (2015)
42. Milde, B., Schneider, M.: A parallel implementation of GaussSieve for the shortest vector problem in lattices. In: PACT, pp. 452–458 (2011)
43. Nguyen, P. Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. Math. Crypt. 2(2), pp. 181–207 (2008)
44. Panigraphy, R.: Entropy based nearest neighbor search in high dimensions. In: SODA, pp. 1186–1195 (2006)
45. Plantard, T., Schneider, M.: Ideal lattice challenge. Online at `http://latticechallenge.org/ideallattice-challenge/` (2014)
46. Pohst, M. E.: On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. ACM Bulletin 15(1), pp. 37–44 (1981)
47. Van de Pol, J., Smart, N. P.: Estimating key sizes for high dimensional lattice-based systems. In: IMACC, pp. 290–303 (2013)
48. Pujol, X., Stehlé, D.: Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605 (2009)
49. Schneider, M.: Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In: WALCOM, pp. 89–97 (2011)
50. Schneider, M.: Sieving for short vectors in ideal lattices. In: AFRICACRYPT, pp. 375–391 (2013)
51. Schneider, M., Gama, N., Baumann, P., Nobach, L.: SVP challenge. Online at `http://latticechallenge.org/svp-challenge` (2014)
52. Schnorr, C.-P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science 53(2), pp. 201–224 (1987)
53. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Math. Programming 66(2), pp. 181–199 (1994)
54. Shoup, V.: Number Theory Library (NTL), v6.2. Online at `http://www.shoup.net/ntl/` (2014)
55. Wang, X., Liu, M., Tian, C., Bi, J.: Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In: ASIACCS, pp. 1–9 (2011)
56. 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 Theorem 1

To prove the claim in Theorem 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 average probability that a nearby (reducing) vector $\boldsymbol{w}$ collides with $\boldsymbol{v}$ in at least one of the t hash tables is at least constant in n:
$$p_1^* = \mathbb{P}_{h_{i,j}\in\mathcal{H}}[\boldsymbol{v}, \boldsymbol{w} \text{ collide} \mid \theta(\boldsymbol{v}, \boldsymbol{w}) \leq \tfrac{\pi}{3}] \geq 1 - \varepsilon. \qquad (0 < \varepsilon \neq \varepsilon(n)) \tag{11}$$
2. The average probability that a distant (non-reducing) vector $\boldsymbol{w}$ collides with $\boldsymbol{v}$ in at least one of the t hash tables is exponentially small, under Heuristic 1:
$$p_2^* = \mathbb{P}_{h_{i,j}\in\mathcal{H}}[\boldsymbol{v}, \boldsymbol{w} \text{ collide} \mid \theta(\boldsymbol{v}, \boldsymbol{w}) > \tfrac{\pi}{3}] \leq O(N^{-0.3782}). \tag{12}$$
3. The number of hash tables grows as $t = N^{0.6218}$.

This would imply that for each search, the number of candidate vectors is of the order $N \cdot N^{-0.3782} = N^{0.6218}$. Overall, we heuristically expect to iterate searching the list $\mathrm{poly}(n) \cdot N$ times, so after substituting $N = 2^{0.2075n+o(n)}$ this leads to the following asymptotic time and space complexities:

- Time (hashing): $O(N \cdot t) = 2^{0.3366n+o(n)}$.
- Time (searching): $O(N^2 \cdot p_2^*) = 2^{0.3366n+o(n)}$.
- Space: $O(N \cdot t) = 2^{0.3366n+o(n)}$.

The remaining part of this subsection is dedicated to proving (11) and (12). We first prove that nearby vectors often collide in at least one of the hash tables, given that k is a suitable function of t. We then show how p_2^* scales as a function of k and t and how to choose k and t to minimize the asymptotic time complexity. Finally we describe how to obtain the trade-off between the space and time complexities as indicated in Figure 1 by choosing k and t slightly differently.

A.1 Reducing vectors collide with constant probability

To guarantee that the list size does not increase by more than a constant factor, we need to make sure that nearby vectors are actually found with constant probability $1 - \varepsilon$ with $\varepsilon \neq \varepsilon(n)$ not depending on the dimension. The condition that the probability of finding nearby vectors is constant will impose a first condition on (the relation between) k and t.

Lemma 2. *Let $k = \log_{3/2}(t) - \log_{3/2}(\ln 1/\varepsilon)$. 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

$$p_1^* = \mathbb{E}_{\Theta\in(0,\frac{\pi}{3})}\left[p^*(\Theta)\right] = \mathbb{E}_{\Theta\in(0,\frac{\pi}{3})}\left[1 - \left(1 - \left(1 - \frac{\Theta}{\pi}\right)^k\right)^t\right], \tag{13}$$

where the angle Θ is a random variable with a certain distribution on $(0, \frac{\pi}{3})$. Since the argument on the right hand side is strictly decreasing in Θ, we can obtain a lower bound by substituting $\Theta = \frac{\pi}{3}$. Using the bound $1 - x < e^{-x}$ which holds for all x, we obtain:

$$p_1^* \geq 1 - \left(1 - \frac{\ln(1/\varepsilon)}{t}\right)^t \geq 1 - \exp(-\ln(1/\varepsilon)) = 1 - \varepsilon. \tag{14}$$

This completes the proof. □

A.2 Non-reducing vectors collide with low probability

The proof that non-reducing vectors do not often lead to hash collisions is somewhat more involved. We need to average the probability of a collision over all possible angles between $\boldsymbol{v}$ and $\boldsymbol{w}$, given that $\boldsymbol{v}$ and $\boldsymbol{w}$ cannot reduce one another, where we thus have to take the density of angles Θ into account. Since it is not so easy to compute the exact distribution of angles that may occur between list vectors throughout the algorithm, we will use Heuristic 1 defined in the main text. To obtain a tight bound on the average probability of a "useless hash collision" we will further use the following lemma about the surface area of hyperspheres. This formula can be found in e.g. [17, p. 10, Eq. (19)].

Lemma 3. *The hypersurface area of the n-dimensional hypersphere of radius R, defined as $S_n(R) = \{\boldsymbol{v} \in \mathbb{R}^n : \|\boldsymbol{v}\| = R\}$, is equal to*

$$A_n(R) = \frac{2\pi^{n/2}}{\Gamma(n/2)} R^{n-1}. \tag{15}$$

The previous heuristic and lemma allow us to derive the density of angles $f(\theta)$ between non-reducing vectors explicitly as follows.

Lemma 4. *Assuming Heuristic 1 holds, the probability density function $f(\theta)$ of angles between pairs of vectors from L_m satisfies*

$$f(\theta) = \sqrt{\frac{2n}{\pi}} \, (\sin\theta)^{n-2} \, [1 + o(1)] = 2^{\log_2(\sin\theta)n + o(n)}. \tag{16}$$

Proof. Suppose without loss of generality that $\boldsymbol{v} = (1, 0, \ldots, 0)$ is fixed. To derive the density at a given angle θ, we basically need to know the fraction of points in Ω that have this angle with $\boldsymbol{v}$. Note that if the angle is fixed at θ, then the first coordinate of $\boldsymbol{w}$ is $\cos\theta$ and so the remaining coordinates of $\boldsymbol{w}$ must satisfy

$$w_2^2 + \cdots + w_n^2 = 1 - \cos^2\theta = \sin^2\theta. \tag{17}$$

This equation defines an $(n-1)$-dimensional hypersphere with radius $\sin\theta$, whose volume follows from Lemma 3. Dividing by the total volume of Ω, the density function f thus satisfies

$$f(\theta) = \frac{1}{M} A_{n-1}(\sin\theta), \qquad M = \int_0^{2\pi} A_{n-1}(\sin\phi)\, d\phi. \tag{18}$$

Writing out the expressions for $A_{n-1}(\sin\theta)$ and $M \sim A_n(1)$, we therefore obtain

$$f(\theta) = \frac{A_{n-1}(\sin\theta)}{(1-o(1))\frac{1}{2}A_n(1)} = \frac{2}{\sqrt{\pi}} \cdot \frac{\Gamma(\frac{n}{2})}{\Gamma(\frac{n-1}{2})} \cdot (\sin\theta)^{n-2} \, [1 + o(1)] \, . \tag{19}$$

Noting that $\Gamma(n + \frac{1}{2}) \sim \sqrt{n} \cdot \Gamma(n)$ for large n, the result follows. □

We are now ready to prove the main result, showing that collisions between faraway vectors occur with exponentially small probability. We first prove a general result relating the probability of a bad collision to the parameter t, and then show how to choose t to balance the time and space complexities.

Lemma 5. *Let $\gamma_1 = \frac{1}{2}\log_2(\frac{4}{3}) \approx 0.2075$, let $\gamma_2 = \log_2(\frac{3}{2}) \approx 0.5850$, and suppose $N = 2^{c_n \cdot n}$ and $t = 2^{c_t \cdot n}$ with $c_n \geq \gamma_1$. For $\theta \in [0, \pi]$, let $U(\theta) < 0$ be defined as*

$$U(\theta) = \log_2(\sin\theta) + \frac{c_t}{\gamma_2} \log_2\left(1 - \frac{\theta}{\pi}\right). \tag{20}$$

Then, for large n the probability of bad collisions is bounded by

$$p_2^* = \mathbb{P}(\textit{bad vectors collide}) \leq O(N^{-\alpha}), \qquad \textit{where } \alpha = \frac{1}{c_n}\left[-c_t - \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} U(\theta)\right] + o(1). \tag{21}$$

Proof. First, if we know the angle $\theta \in (\frac{\pi}{3}, \frac{\pi}{2})$ between two non-reducing vectors, then the probability of a collision is $p^*(\theta) = 1 - (1 - (1 - \frac{\theta}{\pi})^k)^t$. Letting $f(\theta)$ denote the density of angles θ on Ω, we have

$$p_2^* = \mathbb{E}_{\Theta \in (\frac{\pi}{3}, \frac{\pi}{2})} [p^*(\Theta)] = \int_{\pi/3}^{\pi/2} f(\theta) p^*(\theta) d\theta. \tag{22}$$

Substituting $p^*(\theta) = 1 - (1 - (1 - \frac{\theta}{\pi})^k)^t$ and the expression of Lemma 4 for $f(\theta)$, we get

$$p_2^* = \sqrt{\frac{2n}{\pi}} \int_{\pi/3}^{\pi/2} (\sin\theta)^{n-2} [1 + o(1)] \left[1 - \left(1 - \left(1 - \frac{\theta}{\pi}\right)^k\right)^t\right] d\theta. \tag{23}$$

Next, note that for $\theta \gg \frac{\pi}{3}$ we have $t \ll (1 - \frac{\theta}{\pi})^{-k}$ and so $(1 - \left(1 - \frac{\theta}{\pi}\right)^k)^t \approx 1 - t\left(1 - \frac{\theta}{\pi}\right)^k$. In that case, we can simplify the expression between square brackets to $t \cdot \left(1 - \frac{\theta}{\pi}\right)^k$. However, the integration range includes $\frac{\pi}{3}$ as well, so to be careful we will split the integral in two disjoint parts, where we let $\delta = O(n^{-1/2})$:

$$p_2^* = \underbrace{\int_{\pi/3}^{\pi/3+\delta} f(\theta) p^*(\theta) d\theta}_{I_1} + \underbrace{\int_{\pi/3+\delta}^{\pi/2} f(\theta) p^*(\theta) d\theta}_{I_2}. \tag{24}$$

Bounding I_1. Using $f(\theta) \le f(\frac{\pi}{3} + \delta)$ and $p^*(\theta) \le p^*(\frac{\pi}{3})$, we obtain

$$I_1 \le \mathrm{poly}(n) \cdot \delta(1 - \varepsilon) \sin^{n-2}\left(\frac{\pi}{3} + u\right). \tag{25}$$

From a Taylor expansion around $\theta = \frac{\pi}{3}$ of $\sin\theta$ we derive that $\sin(\frac{\pi}{3} + u) \sim \frac{1}{2}\sqrt{3}\,[1 + O(\delta)]$, which leads to

$$I_1 \le 2^{-\gamma_1 n + o(n)} (1 + O(\delta))^n = 2^{-\gamma_1 n + o(n)}. \tag{26}$$

Bounding I_2. For I_2, this choice of δ is sufficient to make the approximation $(1 - \left(1 - \frac{\theta}{\pi}\right)^k)^t \approx 1 - t\left(1 - \frac{\theta}{\pi}\right)^k$ work. Thus, for I_2 we obtain the simplified expression

$$I_2 \le \mathrm{poly}(n) \cdot t \int_{\pi/3+u}^{\pi/2} (\sin\theta)^{n-2} \left(1 - \frac{\theta}{\pi}\right)^k d\theta \le \int_{\pi/3}^{\pi/2} 2^{n \log_2(\sin\theta) + k \log_2(1 - \frac{\theta}{\pi}) + c_t n + o(n)} d\theta. \tag{27}$$

Note that the integrand is exponential in n (assuming k is at most linear n) and the exponent is a continuous, differentiable function of θ. So the asymptotic behavior of the integral is the same as the asymptotic behavior of its maximum value:

$$\log_2 I_2 \le c_t n + \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} \left\{ n \log_2(\sin\theta) + k \log_2\left(1 - \frac{\theta}{\pi}\right) \right\} + o(n). \tag{28}$$

Bounding $p_2^* = I_1 + I_2$. Combining the results from (26) and (28), we have

$$\frac{\log_2 p_2^*}{n} \le \max\left\{ -\gamma_1,\ c_t + \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} U(\theta) \right\} + o(1). \tag{29}$$

In the end, we would like to prove that $p_2^* \le N^{-\alpha}$, or equivalently $\frac{1}{n} \log_2 p_2^* \le -\alpha c_n$. To complete the proof, it therefore suffices to prove the following two inequalities:

$$\max\{-\gamma_1, c_t + \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} U(\theta)\} \le -\alpha c_n + o(1). \tag{30}$$

Note that the inequality corresponding to the first term in the maximum is automatically satisfied if $\alpha \le 1$ and $N = 2^{\gamma_1 n + o(n)}$. For the second inequality, we isolate α to obtain

$$\alpha \le \frac{1}{c_n} \left[-c_t - \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} U(\theta) \right] + o(1). \tag{31}$$

Since we want α to be as large as possible, we set α equal to its lower bound, leading to the result. □

A.3 Balancing the parameters

To return to actual consequences for the complexity of the scheme, recall that the overall time and space complexities are heuristically given by:

- Time (hashing): $O(N \cdot t) = 2^{(c_n+c_t)n+o(n)}$.
- Time (searching): $O(N^2 \cdot p_2^*) = 2^{(c_n+(1-\alpha)c_n)n+o(n)}$.
- Time (overall): $2^{(c_n+\max\{c_t,(1-\alpha)c_n\})n+o(n)}$.
- Space: $O(N \cdot t) = 2^{(c_n+c_t)n+o(n)}$.

Writing the overall time complexity as $2^{c_{\text{time}}n+o(n)}$ and the space complexity as $2^{c_{\text{time}}n+o(n)}$, this means

$$c_{\text{time}} = c_n + \max\{c_t, (1-\alpha)c_n\}, \qquad c_{\text{space}} = c_n + c_t. \tag{32}$$

Also recall that from bounds on the kissing constant in high dimensions, we expect that $N = 2^{0.2075n}$ or $c_n = \gamma_1$. To balance the asymptotic time complexities for hashing and searching, so that the time and space complexities are the same and the time complexity is minimized, we solve $(1-\alpha)\gamma_1 = c_t$ numerically for c_t[5] to obtain the following corollary.

Corollary 1. *Taking $c_t \approx 0.129043$ leads to:*

$$\theta^* \approx 0.458921\pi, \qquad \alpha \approx 0.378163, \qquad c_{\text{time}} \approx 0.336562, \qquad c_{\text{space}} \approx 0.336562.$$

In other words, using $t \approx 2^{0.129043n}$ hash tables and a hash length of $k \approx 0.220600n$, the heuristic time and space complexities of the algorithm are balanced at $2^{0.336562n+o(n)}$.

A.4 Trade-offs between space and time

Finally, note that $c_t = 0$ leads to the original GaussSieve algorithm, while $c_t \approx 0.129043$ minimizes the heuristic time complexity at the cost of more space. One can also obtain a continuous time-memory trade-off between the GaussSieve and the HashSieve algorithm by considering values $c_t \in (0, 0.129043)$. Numerically evaluating the resulting time and space complexities for this range of values of c_t leads to the graph shown in Figure 1.

B The quadratic sieve and angular LSH trade-offs and speed-ups

Let us finally describe a variant of the Nguyen-Vidick sieve which achieves the same performance, but may be slightly easier to think about. Whereas in the Nguyen-Vidick sieve a random subset of centers C_{m+1} is chosen and only some of the pairs of vectors $(\boldsymbol{v}, \boldsymbol{w}) \in L_m \times C_{m+1}$ are considered for reduction, this algorithm simply considers all combinations $(\boldsymbol{v}, \boldsymbol{w}) \in L_m \times L_m$ and sees if their sum or difference is short enough to be added to L_{m+1}. This means that significantly more comparisons are done than in the NV-sieve, but clearly the runtime is (at most) quadratic in the size of the input list, hence bounded by $2^{0.415n+o(n)}$. Moreover, as strictly more pairs of vectors are considered for reduction, the output list L_{m+1} contains all vectors that would have passes through an iteration of the Nguyen-Vidick sieve. So this algorithm achieves the same asymptotic time and space complexities as the Nguyen-Vidick sieve, with the same heuristic guarantee of correctness as the Nguyen-Vidick sieve.

The resulting algorithm, without using angular LSH, is outlined in Algorithm 6. For this algorithm, LSH methods can be applied in a similar fashion as for the NV-sieve, resulting in a space-time trade-off or a clean speed-up as illustrated in Algorithms 7 and 8 respectively.

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

Algorithm 6 The QuadraticSieve algorithm (without angular LSH)

Require: An input list L_m of $(4/3)^{n/2+o(n)}$ lattice vectors of norm at most $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Ensure: The output list L_{m+1} has size $(4/3)^{n/2+o(n)}$ and only contains lattice vectors of norm at most $\gamma \cdot R$

Initialize an empty list L_{m+1}
for each $\boldsymbol{v} \in L_m$ **do**
  for each $\boldsymbol{w} \in L_m \cup \{\boldsymbol{0}\}$ **do**
    if $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ **then**
      Add $\boldsymbol{v} \pm w$ to the list L_{m+1}

Algorithm 7 The QuadraticSieve algorithm (with angular LSH)

Require: An input list L_m of $(4/3)^{n/2+o(n)}$ lattice vectors of norm at most $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Ensure: The output list L_{m+1} has size $(4/3)^{n/2+o(n)}$ and only contains lattice vectors of norm at most $\gamma \cdot R$

Initialize an empty list L_{m+1}
Initialize k empty hash tables T_i and sample $k \cdot t$ random hash functions $h_{i,j} \in \mathcal{H}$
for each $\boldsymbol{w} \in L_m$ **do**
  Add $\boldsymbol{w}$ to all k hash tables T_i, to the buckets $T_i[h_i(\boldsymbol{w})]$
for each $\boldsymbol{v} \in L_m$ **do**
  for each $\boldsymbol{w} \in \bigcup_{i=1}^{t} T_i[h_i(\pm\boldsymbol{v})] \cup \{\boldsymbol{0}\}$ **do**
    if $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ **then**
      Add $\boldsymbol{v} \pm w$ to the list L_{m+1}

Algorithm 8 The QuadraticSieve algorithm (with angular LSH, space-efficient)

Require: An input list L_m of $(4/3)^{n/2+o(n)}$ lattice vectors of norm at most $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
Ensure: The output list L_{m+1} has size $(4/3)^{n/2+o(n)}$ and only contains lattice vectors of norm at most $\gamma \cdot R$

Initialize an empty list L_{m+1}
for each $i \in \{1, \ldots, t\}$ **do**
  Initialize an empty hash table T and sample k random hash functions $h_{i,j} \in \mathcal{H}$
  for each $\boldsymbol{w} \in L_m$ **do**
    Add $\boldsymbol{w}$ to the hash table T, to bucket $T[h_i(\boldsymbol{w})]$
  for each $\boldsymbol{v} \in L_m$ **do**
    for each $\boldsymbol{w} \in T[h_i(\pm\boldsymbol{v})] \cup \{\boldsymbol{0}\}$ **do**
      if $\|\boldsymbol{v} \pm \boldsymbol{w}\| \leq \gamma \cdot R$ **then**
        Add $\boldsymbol{v} \pm w$ to the list L_{m+1}