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 angular locality-sensitive hashing, we get both theoretical and practical speed-ups for finding shortest vectors in lattices. Optimizing for time, we obtain a heuristic time and space complexity for solving SVP of $2^{0.3366n+o(n)}$. Preliminary experiments show that the proposed HashSieve algorithm already outperforms the GaussSieve in low dimensions, and that the practical increase in the space complexity is much smaller than the asymptotic bounds suggest. Using probing we show how we can further reduce the space complexity at the cost of slightly increasing the time complexity.

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

1 Introduction

Lattice-based cryptography. Recently lattice-based cryptography has attracted wide attention from the cryptographic community, due to e.g. its presumed resistance against quantum attacks [7], the existence of lattice-based fully homomorphic encryption schemes [16], efficient ring-based cryptographic primitives like NTRU [18, 29], and various worst-case to average-case hardness reductions [2, 15]. An important problem related to lattice-based cryptography is to estimate the hardness of the underlying hard lattice problems, such as finding short vectors; understanding their hardness is crucial for accurately choosing parameters for lattice-based cryptography [13, 27, 39].

Finding short vectors. Finding a shortest non-zero lattice vector or approximating it up to a constant factor is well-known to be NP-hard [3, 22], but various methods to quickly find reasonably short vectors are known, with the most well-known ones being the basis reduction algorithms LLL [25] and its blockwise generalization BKZ [45, 46]. The latter algorithm 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 the block-size β, the more time the algorithm takes and the shorter the vectors in the basis that the algorithm outputs. BKZ uses an algorithm for 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.

Finding shortest vectors. In the original description of BKZ, enumeration was used as the SVP subroutine [11, 21, 38, 46]. This method has a low space complexity, but its runtime is superexponential in the dimension n, which is known to be suboptimal: both sieving [4] and the Voronoi cell algorithm [32] run in single exponential time. The main drawbacks of the latter methods are that their space complexities are exponential in n as well, and that the hidden constants in the time complexities are so big that enumeration is faster than both these methods in any practical dimension n. Especially after the recent extreme pruning improvement to enumeration [14], it seems that these other methods still have a long way to go to beat enumeration.

Sieving in arbitrary lattices. On the other hand, both sieving and the Voronoi cell algorithm are relatively new and less explored than enumeration, and recent improvements have shown that at least sieving might be able to compete with enumeration in the future. Whereas the original work of Ajtai et al. [4] showed only that sieving can solve SVP in time and space $2^{O(n)}$ (with large hidden constants), it was later shown that one can provably solve SVP in arbitrary lattices in time $2^{2.47n}$ and space $2^{1.24n}$ [17,35,40,41]. Heuristic analyses of various sieving algorithms further suggest that one may actually be able to solve SVP in time $2^{0.415n}$ and space $2^{0.208n}$ [6,33,35], or optimizing for time, in time $2^{0.378n}$ and space $2^{0.293n}$ [6, 48, 49]. Various papers have also shown how to speed up sieving in practice [12, 20, 30, 31, 34, 42, 43], and sieving has recently made its way to the top 20 of the SVP challenge hall of fame [44].

Sieving in ideal lattices. Besides various attempts to make sieving practical for arbitrary lattices, some papers have also investigated how sieving can be made even faster and more space-efficient for ideal lattices [20, 33, 43]. Since lattice-based cryptographic primitives are commonly based on ideal lattices for efficiency, this class of lattices may be the most relevant for lattice-based cryptanalysis. It is not known whether there exist exponential speed-ups for sieving in ideal lattices, but several polynomial speed-ups are known which drastically improve the time and space complexities as well. Ishiguro et al. [20] used sieving to obtain the highest record in the ideal lattice challenge hall of fame [36], solving SVP in dimension 128. This is already higher than the current highest dimension for which enumeration was used to find a record in the SVP or ideal lattice challenge hall of fame.

Contributions. In this work we show how to obtain exponential speed-ups for sieving using angular locality-sensitive hashing [8, 19]. For each stored list vector $\boldsymbol{w}$ in sieving we store low-dimensional, lossy sketches (hashes) of these vectors, such that vectors that are nearby have a higher probability of having the same sketch than vectors which are far away. To search the list for nearby vectors we then do not go through the entire list, but only consider vectors with the same hash values. Choosing certain parameters appropriately, we can guarantee that the number of candidate vectors to consider decreases by an exponential factor, while the probability that nearby vectors are missed is small. Optimizing for time this leads to an algorithm with heuristic time and space complexities bounded by $2^{0.337n}$. By tuning the parameters differently, we get a continuous heuristic trade-off between the space and time complexities, as illustrated (and compared with previous heuristic results) in Figure 1. Practical experiments with our proposed algorithm seem to confirm our analysis, and show that (i) already in low dimensions, our algorithm seems to be faster than the GaussSieve algorithm of Micciancio and Voulgaris [33]; and (ii) as expected, the practical increase in the space complexity is significantly smaller than one might have guessed from only looking at the exponent of the space complexity. We also show how the space complexity can be further reduced at almost no cost by probing several hash buckets in each table and reducing the number of hash tables by a factor $\text{poly}(n)$.

Main ideas. While locality-sensitive hashing was briefly considered in the context of sieving by Nguyen and Vidick [35, Section 4.2.2], there are two main differences between their application and our techniques that explain the improvements in this paper.

- Nguyen and Vidick considered L_2-hashing based on the Euclidean norm [5, 10], while we will argue that it seems more natural to use hashing based on angular distances [8].
- Nguyen and Vidick focused on the worst-case difference between 'reducing' and 'non-reducing' vectors, while we will focus on the average-case difference.

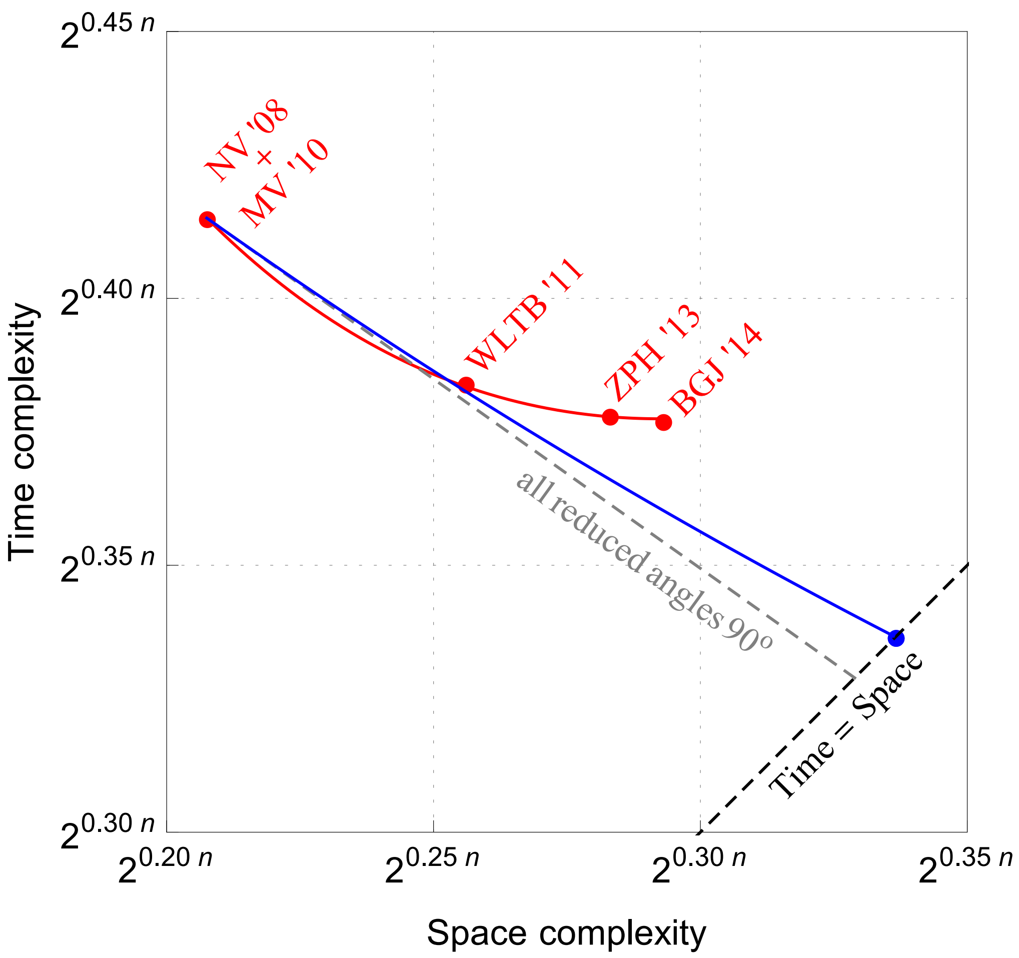


Fig. 1. The heuristic space-time trade-off of various heuristic sieving algorithms from the literature (the red points), the space-time trade-off of [6, Figure 3] (the red curve), and the heuristic trade-off between the space and time complexities obtained with our algorithm (the blue curve). The dashed, gray line shows the estimate for the space-time trade-off of our algorithm obtained by assuming that all reduced vectors are orthogonal (cf. Proposition 1).

To illustrate the second point: the worst-case 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 of two non-reducing vectors is close to 90°.

Outline. The paper is organized as follows. In Section 2 we describe the technique of locality-sensitive hashing for finding near(est) neighbors, and we describe a family of angular hash functions. Section 3 describes how to apply these techniques to e.g. the GaussSieve algorithm of Micciancio and Voulgaris to obtain the faster HashSieve algorithm. In Section 4 (and Appendix A) we show that under certain natural heuristics, sieving with locality-sensitive hashing provably solves SVP in time and space $2^{0.3366n+o(n)}$, and we show how we can choose the parameters differently to obtain a continuous trade-off between the time and space complexities (as illustrated in Figure 1). Section 5 describes preliminary experimental results using the HashSieve algorithm, aimed at verifying our heuristic analysis. Section 6 describes a useful technique to reduce the space complexity by a factor poly(n) at almost no additional cost. In Section 7 we conclude with some open problems and possible applications of (angular) locality-sensitive hashing to other lattice algorithms, such as the Voronoi cell algorithm [32].

2 Locality-sensitive hashing

2.1 Introduction

The near(est) neighbor problem is the following [19]: Given a list of n-dimensional vectors $L = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_N\} \subset \mathbb{R}^n$, preprocess L in such a way that when 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 for small dimensions n there may be ways to answer these queries in time sub-linear or even logarithmic in the list size N, for large dimensions n it generally seems hard to answer queries faster 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" [19].

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 $d(\boldsymbol{v}, \boldsymbol{w}^*) = r_1$ from $\boldsymbol{v}$, and all other points $\boldsymbol{w} \in L$ are at distance at least $d(\boldsymbol{v}, \boldsymbol{w}) \geq r_2 = (1 + \varepsilon)r_1$ 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 $\rho = (1+\varepsilon)^{-2} < 1$ [5]. For small $\varepsilon > 0$, this means that with a space complexity almost quadratic in N, one can answer queries in time sub-linear in N.

2.2 Hash families

The method of [5] described above, as well as the method we will use later, relies on using *locality-sensitive hash functions* [19]. 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, for a certain similarity measure[1] D we define $B(\boldsymbol{v}, r) = \{\boldsymbol{w} \in \mathbb{R}^n : D(\boldsymbol{v}, \boldsymbol{w}) \leq r\}$, and the set U below may be thought of as (a subset of) the natural numbers $\mathbb{N}$.

Definition 1. *[19] 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 $\boldsymbol{w} \in B(\boldsymbol{v}, r_1)$ then $\mathbb{P}_{h \in \mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] \geq p_1$.*
- *If $\boldsymbol{w} \notin B(\boldsymbol{v}, r_2)$ then $\mathbb{P}_{h \in \mathcal{H}}[h(\boldsymbol{v}) = h(\boldsymbol{w})] \leq p_2$.*

Note that if there exists a locality-sensitive 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.

2.3 Amplification

Before turning to how such hash families may actually be constructed for a similarity measure D, or how to use these to find nearest neighbors quickly, we first note that in general it is unknown whether efficiently computable (r_1, r_2, p_1, p_2)-sensitive hash families

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

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 $j = 1, 2$.

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 *some* $i \in [k]$. Clearly $h(\boldsymbol{v}) \neq h(\boldsymbol{w})$ iff $h_i(\boldsymbol{v}) \neq h_i(\boldsymbol{w})$ for all $i \in [k]$, so if $\mathbb{P}[h_i(\boldsymbol{v}) \neq h_i(\boldsymbol{w})] = 1 - p_j$ for all i, then $\mathbb{P}[h(\boldsymbol{v}) \neq h(\boldsymbol{w})] = (1 - p_j)^k$ for $j = 1, 2$.

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

2.4 Finding nearest neighbors

To use these hash families to find nearest neighbors, we can use the following method first described in [19]. 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, and search this list of candidates for the nearest neighbor.

Clearly, the quality of this algorithm for finding nearest neighbors depends on the quality of the underlying hash family and on the parameters k and t. Larger values of k and t amplify the gap between the probabilities of finding 'good' and 'bad' vectors, but larger parameters come at the cost of having to compute many hashes, both during the preprocessing phase and during the querying phase. The following lemma shows how to balance k and t such that the overall time complexity is minimized.

Lemma 1. *[19] Suppose there exists a (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$, and suppose that $\{\boldsymbol{w} \in L : r_1 < D(\boldsymbol{v}, \boldsymbol{w}) < r_2\} = \emptyset$. Then, taking*

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

with high probability we can find (if it exists) an element $\boldsymbol{w}^* \in L$ with $D(\boldsymbol{v}, \boldsymbol{w}^*) \leq r_1$, with the following costs:

(1) Time for preprocessing the list: $O(N^{1+\rho} \log_{1/p_2} N)$.
(2) Space complexity of the preprocessed data: $O(N^{1+\rho})$.
(3) Time for answering a query $\boldsymbol{v}$: $O(N^\rho)$.
  (3a) Hash evaluations of the query vector $\boldsymbol{v}$: $O(N^\rho)$.
  (3b) List vectors to compare to the query vector $\boldsymbol{v}$: $O(N^\rho)$.

Proof. Let us start with 'good' 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 $p_1^k = (p_1^{k/\rho})^\rho = N^{-\rho}$. Thus, 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. Using e.g. the Chernoff bound, with overwhelming probability the exact number of 'bad' colliding vectors will be relatively close to this average.

Summarizing, (1) the costs of preprocessing the data are computing $N \cdot t \cdot k = O(N^{1+\rho} \log_{1/p_2} N)$ hashes; (2) the space complexity of the preprocessed data (t hash tables with N vectors) is $O(N \cdot t) = O(N^{1+\rho})$; (3a) 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 (3b) 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 sense this algorithm can be seen as a generalization of the naive brute-force search method, as $k = 0$ and $t = 1$ corresponds to checking the whole list in linear time with linear space.

2.5 Angular hashing

Let us now consider actual hash families for the similarity measure D that we are interested in. As we will argue in the next section, what actually seems to be a more natural choice for D than the Euclidean distance is the *angular distance*, 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' one another if their common angle is small, and vectors are 'far apart' if their angle is large. In a sense, this is similar to the L_2-norm: if two vectors have similar Euclidean norms, then their L_2-distance is large if and only if their angular distance is large. For this similarity measure the following hash family $\mathcal{H}$ was first described in [8]:

$$\mathcal{H} = \{h_{\boldsymbol{a}} : \boldsymbol{a} \in \mathbb{R}^n, \|\boldsymbol{a}\| = 1\}, \tag{3}$$

$$h_{\boldsymbol{a}}(\boldsymbol{v}) \triangleq \begin{cases} 1 & \text{if } \boldsymbol{a}^T\boldsymbol{v} \geq 0; \\ 0 & \text{if } \boldsymbol{a}^T\boldsymbol{v} < 0. \end{cases} \tag{4}$$

Intuitively, the random unit 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 2-dimensional plane passing through the origin, and with probability 1 a random vector $\boldsymbol{a}$ does not lie on this plane

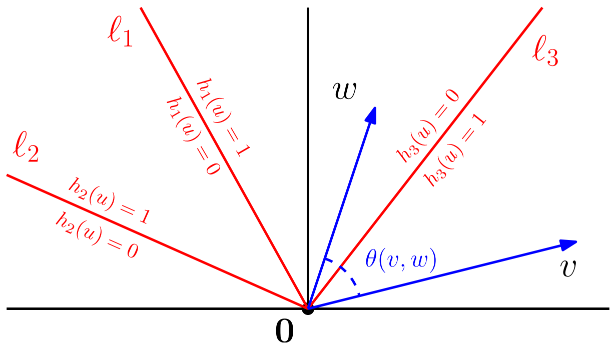


Fig. 2. A sketch of the angular hash family $\mathcal{H}$. Each line ℓ_i corresponds to a different vector $\boldsymbol{a}_i$ and a different locality-sensitive hash function $h_i \in \mathcal{H}$. Here $h_3(\boldsymbol{v}) \neq h_3(\boldsymbol{w})$ because ℓ_3 separates $\boldsymbol{v}$ and $\boldsymbol{w}$, while $h_1(\boldsymbol{v}) = h_1(\boldsymbol{w})$ and $h_2(\boldsymbol{v}) = h_2(\boldsymbol{w})$.

(for $n > 2$). This means that the hyperplane defined by $\boldsymbol{a}$ intersects this plane in some line ℓ, where the direction of ℓ is orthogonal to the projection of $\boldsymbol{a}$ onto the plane. Since $\boldsymbol{a}$ is taken uniformly at random from the hypersphere of radius 1, the line ℓ has a uniformly random 'direction' in the plane, and it maps $\boldsymbol{v}$ and $\boldsymbol{w}$ to different hash values if and only if ℓ separates $\boldsymbol{v}$ and $\boldsymbol{w}$ in the plane. Figure 2 sketches the 2-dimensional plane, possible points $\boldsymbol{v}$ and $\boldsymbol{w}$, and possible lines ℓ_i in this plane arising from different vectors $\boldsymbol{a}_i$. It is not hard to see that the probability that such a line ℓ separates $\boldsymbol{v}$ and $\boldsymbol{w}$ is directly proportional to their common angle $\theta(\boldsymbol{v}, \boldsymbol{w})$ as follows [8]:

$$\mathbb{P}_{h_{\boldsymbol{a}} \in \mathcal{H}}\left[h_{\boldsymbol{a}}(\boldsymbol{v}) = h_{\boldsymbol{a}}(\boldsymbol{w})\right] = 1 - \frac{\theta(\boldsymbol{v}, \boldsymbol{w})}{\pi}. \tag{5}$$

Thus, 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, it is $(\frac{\pi}{3}, \frac{\pi}{2}, \frac{2}{3}, \frac{1}{2})$-sensitive.

3 From GaussSieve to 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 GaussSieve of Micciancio and Voulgaris [33]. We have chosen this algorithm as our main focus since it seems to be the most practical sieving algorithm to this date, which is further motivated by the extensive attention it has received in the past few years [12, 20, 30, 31, 34, 42, 43] and by the fact that the highest sieving record in the SVP challenge database was obtained using (a modification of) the GaussSieve [24]. We note however that the same ideas can also be applied to other sieving algorithms, and in particular to the heuristic sieve algorithm of Nguyen and Vidick [35] (see Appendix B) which has provable heuristic bounds on the time complexity.

3.1 GaussSieve

A simplified version of the GaussSieve algorithm of Micciancio and Voulgaris is described in Algorithm 1. The algorithm iteratively builds a longer and longer list of vectors, occasionally reducing the lengths of list vectors in the process, until at some point L contains a shortest vector. Vectors are sampled from a discrete Gaussian over the lattice, using e.g. the sampling algorithm of Klein [23, 33]. If list vectors are modified or newly sampled vectors are reduced, they are pushed to a stack. This stack is then first emptied before new vectors are sampled.

Algorithm 1 The GaussSieve algorithm

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

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

$$\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2: \quad \text{if } \|\boldsymbol{u}_1 \pm \boldsymbol{u}_2\| < \|\boldsymbol{u}_1\| \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2. \tag{6}$$

Throughout the execution of the algorithm, the list L always satisfies the property that any two vectors $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ are pairwise reduced, i.e., $\|\boldsymbol{w}_1 \pm \boldsymbol{w}_2\| \geq \max\{\|\boldsymbol{w}_1\|, \|\boldsymbol{w}_2\|\}$. 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 reduced 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 n dimensions: 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 $(4/3)^{n/2+o(n)} = 2^{0.2075n+o(n)}$ [9].

While the space complexity of the GaussSieve is reasonably well understood, there are no heuristic bounds on the time complexity of this algorithm. One might estimate that the time complexity is 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. Thus the time complexity is at least quadratic in $|L|$. The algorithm further seems to show a similar asymptotic behavior as the sieve algorithm of Nguyen and Vidick [35], for which the asymptotic time complexity is heuristically known to be quadratic in $|L|$, i.e., of the order $2^{0.415n+o(n)}$. Thus one might conjecture that the GaussSieve also has a time complexity of $2^{0.415n+o(n)}$.

3.2 GaussSieve with angular reductions

Since the 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°, we might expect the same heuristics to apply to any reduction method that guarantees that angles in L are at least 60°. In particular, if we only reduce vectors when their angle is less than 60°, using the following rule:

$$\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2: \quad \text{if } \theta(\boldsymbol{u}_1, \pm\boldsymbol{u}_2) < 60^\circ \text{ and } \|\boldsymbol{u}_1\| \geq \|\boldsymbol{u}_2\| \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2, \tag{7}$$

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 be estimated at $2^{0.415n+o(n)}$. Basic experiments indicate that although with this weaker notion of reduction the list size increases by a certain factor, this factor appears to be sub-exponential in n.

Algorithm 2 The HashSieve algorithm

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

3.3 HashSieve with angular reductions

Replacing the stronger notion of reduction of (6) by the weaker one of (7), 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 2. Note that the setup costs of locality-sensitive hashing are spread out over the various iterations; at each iteration we only update the parts of the hash tables that were affected by updating L.

3.4 HashSieve

Finally, note that there seems to be no point to skipping potential reductions. So while for our intuition and for the theoretical motivation we may consider the case where the reductions are based on (7), in practice we will again reduce vectors (in Lines 8, 9 of Algorithm 2) based on (6).

4 Theoretical analysis of HashSieve

For analyzing the time complexity of HashSieve, we will assume that the GaussSieve has a time complexity which is quadratic in the list size, i.e. a time complexity of $2^{0.415n+o(n)}$. We will then show that using locality-sensitive hashing, we can reduce the time complexity to $2^{0.337n+o(n)}$. Note that in reality it is not known whether the time complexity of GaussSieve is provably quadratic in $|L|$ (up to sub-exponential factors), so this might not guarantee a heuristic time complexity of the order $2^{0.337n+o(n)}$. In Appendix B we therefore illustrate how the same ideas may be applied to the sieve algorithm of Nguyen and Vidick [35], for which the heuristic time complexity is known to be at most $2^{0.415n+o(n)}$. This then implies that indeed, with sieving we can heuristically solve the shortest vector problem in time and space $2^{0.337n+o(n)}$. In the main text we will focus on the GaussSieve due to its better practical performance, even though theoretically one might rather apply this analysis to the algorithm of Nguyen and Vidick due to their heuristic bounds on the time complexity.

So for now, let us assume that the GaussSieve has a time complexity quadratic in $|L|$, and that $|L| \leq 2^{0.208n+o(n)}$ due to conjectured bounds on the kissing constant. To get an estimate of the time and space complexities of the HashSieve, note that in high dimensions, angles close to 90° are much more likely to occur between random vectors than smaller angles. So one might guess that for two list vectors $\boldsymbol{w}_1, \boldsymbol{w}_2 \in L$ (which necessarily have an angle larger than 60°), with high probability their angle is close to 90°. On the other hand, vectors that can reduce one another (with respect to angular reductions) always have an angle less than 60°, and by similar arguments we expect this angle to always be close to 60°. Under the extreme assumption that all 'reduced angles' are *exactly* 90° (and non-reduced angles are at most 60°), we obtain the following estimate for the optimized time and space complexities of HashSieve.

Proposition 1. *Assuming reduced vectors are always pairwise orthogonal, sieving with locality-sensitive hashing with parameters $k = 0.2075n + o(n)$ and $t = 2^{0.1214n+o(n)}$ heuristically solves SVP in time and space $2^{0.3289n+o(n)}$. With the same assumption, we furthermore obtain the trade-off between the space and time complexities indicated by the dashed line in Figure 1.*

Proof. If all reduced 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 solve a single search for a reducing vector in time $N^{\rho} = 2^{0.1214n+o(n)}$ using $t = 2^{0.1214n+o(n)}$ hash tables. Since the time complexity of the algorithm is determined by performing such searches N times, the overall time complexity is of the order $N^{1+\rho} = 2^{0.3289n+o(n)}$. □

Of course, in practice not all reduced angles are actually 90°, and one should carefully analyze what is the real probability that a vector 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 central theorem follows from this analysis and shows how to choose the parameters asymptotically to optimize the time complexity. A proof of Theorem 1 can be found in Appendix A.

Theorem 1. *Sieving with locality-sensitive hashing with parameters $k = 0.2206n + o(n)$ and $t = 2^{0.1290n+o(n)}$ heuristically solves SVP in time and space $2^{0.3366n+o(n)}$. We furthermore obtain the trade-off between the space and time complexities 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 not very different. Thus the simple estimate based on the intuition that in high dimensions "everything is orthogonal" is not far off.

5 Practical performance of HashSieve

To experimentally verify our analysis, we implemented both the GaussSieve and the HashSieve to try to compare the asymptotic behaviors of these algorithms. For implementing the HashSieve, we note that we can use various tweaks to further speed up the algorithm. These include:

(a) With the HashSieve, the whole purpose of L is lost, and we can remove L in its entirety from the algorithm.

(b) Instead of making a big list of candidates first, we go through the hash tables one by one, checking if collisions in this table lead to reductions. If a reducing vector is already found in one of the earlier tables T_i, this may save up to $t \cdot k$ hash computations for $\boldsymbol{v}$.
(c) Since $h_i(-\boldsymbol{v}) = -h_i(\boldsymbol{v})$ and we always want to get either both or neither of the two buckets (to compare $\boldsymbol{v} \pm \boldsymbol{w}$ to $\boldsymbol{v}$), we merge each pair of hash buckets labeled $h_i(\boldsymbol{v})$ and $h_i(-\boldsymbol{v})$ into one hash bucket labeled $h_i(\pm\boldsymbol{v})$. This reduces the number of buckets per hash table and the number of hashes to compute by a factor 2.
(d) For choosing vectors $\boldsymbol{a}_{i,j}$ to use for the hash functions h_i, there is no reason to assume that drawing $\boldsymbol{a}$ from a specific, sufficiently large random subset of $\{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| = 1\}$ would lead to substantially different results. In particular, we can use sparse vectors $\boldsymbol{a}_{i,j}$ to make hash computations significantly cheaper, while retaining the same performance [1, 26]. Our experiments in dimensions 35 up to 70 indicate that even if all vectors $\boldsymbol{a}_{i,j}$ have only two non-zero entries (which are the same), the algorithm still finds the shortest vector in roughly the same number of iterations.
(e) We should not store the actual vectors in each hash table T_i, but only pointers to list vectors (which are all stored in a central list of vectors). This means that the space complexity increases from $O(N \cdot n)$ for the GaussSieve, to $O(N \cdot n + N \cdot t)$ for the HashSieve (instead of $O(N \cdot n \cdot t)$). Asymptotically the space complexity thus increases by a factor $\frac{t}{n}$ (instead of t).

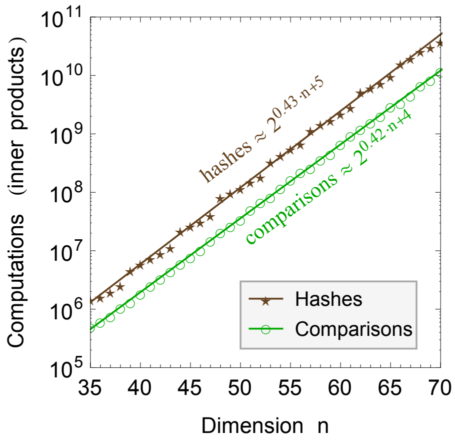
With these tweaks, we performed several experiments of finding shortest vectors using the lattices of the SVP challenge website [44]. We generated lattice bases for different seeds and different dimensions using the SVP challenge generator, then used NTL [47] to preprocess the basis (LLL reduction), and then used our implementations of the GaussSieve and the HashSieve to obtain these results. For the HashSieve we chose the parameters k and t by rounding the theoretical estimates of Theorem 1 to the nearest integer, i.e., $k = \lfloor 0.2206n \rceil$ and $t = \lfloor 2^{0.1290n} \rceil$. We performed experiments in dimensions 35 up to 70, and various statistics of these experiments are discussed below and illustrated in Figure 3.

Note that clearly there are various ways to further speed up both the GaussSieve and the HashSieve, using e.g. better preprocessing (BKZ with higher block sizes), vectorized code, parallel implementations, optimized samplers, specialized hardware etc. Furthermore, for the HashSieve our choice of the parameters k and t might be suboptimal; choosing k and t slightly differently may further increase the performance. The point of our experiments is only to try to give a somewhat fair comparison of the two algorithms and to try to estimate and compare the asymptotic behaviors of these algorithms. Further optimizing the practical performance of these algorithms is left for future work.

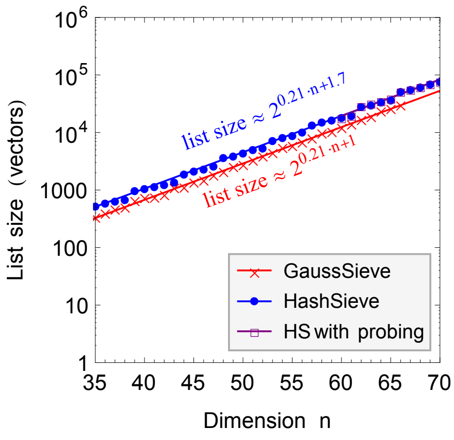
5.1 Computations of the HashSieve

Various characteristics of the HashSieve may be of interest, such as how often the algorithm computes hashes of a vector $\boldsymbol{v}$, and how often the algorithm compares a target vector $\boldsymbol{v}$ with a list vector $\boldsymbol{w}$ to see if a reduction is possible. Note that computing a hash consists of computing k inner products between $\boldsymbol{v}$ and $\boldsymbol{a}_{i,1}, \dots, \boldsymbol{a}_{i,k}$, while a comparison consists of computing only one inner product between $\boldsymbol{v}$ and $\boldsymbol{w}$. To compare these numbers, we therefore compare the total number of inner products computed for both cases, even though hash-based inner products may be significantly cheaper than comparison-based inner products.

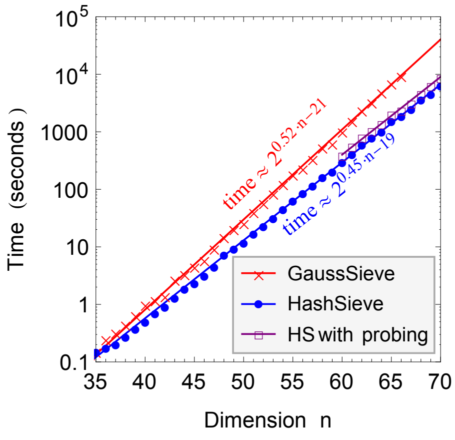
Theoretically we have chosen k and t so that the total time for each of these operations is roughly balanced, and Figure 3a confirms that this indeed seems to be the case. The total number of inner products for hashing seems to be a constant factor higher than the



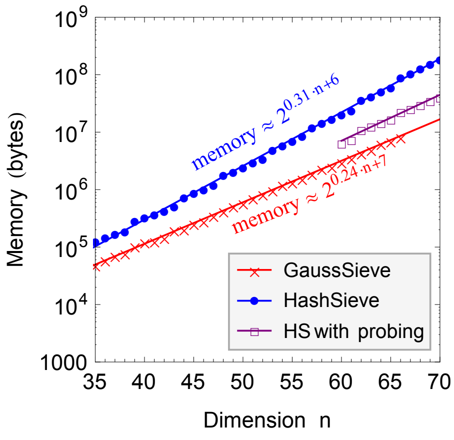
(a) HashSieve characteristics



(b) List size characteristics



(c) Time complexities



(d) Space complexities

Fig. 3. Experimental data obtained from applying the GaussSieve and the HashSieve to LLL-reduced random lattice bases in dimensions 35 to 70. Markers indicate experimental data, and lines are least-squares fits for the data. For each algorithm and dimension we performed at least 10 experiments. Note that the step-wise behavior of some curves can be explained by the fact that the parameter k of the HashSieve is small but integral, and increases by 1 only once every few dimensions.

Figure 3a shows the time spent on hashing and on comparing vectors in the HashSieve algorithm, expressed in terms of inner products. As expected, with our choice of parameters these computations are more or less balanced. Figure 3b confirms our intuition that if we miss a small fraction of the reducing pairs of vectors, the list size also increases by a small factor. Figure 3c compares the time complexities of both algorithms in our experimental setting, and seems to confirm our heuristic estimate of a speed-up of roughly $2^{0.07n}$ compared to the GaussSieve, even if the actual times are higher than the heuristics indicate. Figure 3d illustrates the space requirements for both schemes, showing that the increase in the space complexity is significantly smaller than the heuristically estimated asymptotic factor $2^{0.13n}$.

The technique of probing, leading to a better space complexity and a slightly worse time complexity than the HashSieve, is explained in Section 6.

total number of inner products computed for comparing vectors. This is also desirable, as hashing is significantly cheaper than comparing vectors. Tuning the parameters differently may change this ratio a bit, and may lead to a different overall time and space complexity.

5.2 List sizes

To heuristically prove that the algorithm succeeds using a certain amount of time and space, we assumed that if reductions are missed with a constant probability (which is the case with our choice of parameters), then the list size also increases by a constant factor. Figure 3b seems to support these heuristic assumptions, as indeed the list size in the HashSieve algorithm seems to be a (small) constant factor larger than in the GaussSieve.

5.3 Time complexities

Clearly one of the main parameters of interest is the time complexity, and Figure 3c compares the run times of our implementations of GaussSieve and HashSieve on a single core of a Dell Optiplex 780, which has a processor speed of 2.66 GHz and 4 GB of RAM. Our heuristics show that for large n we expect to achieve a speed-up of roughly $2^{0.078n}$ for each list search, and in practice we see that the asymptotic speed-up is close to $2^{0.07n}$. The experiments further indicate that the overhead of using hash tables (extra table look-ups, using more memory) is reasonably small, since the constant in the time exponent only increases by about 2. Note that the actual coefficients in the exponents are higher than heuristics suggest, which is consistent with various previous experiments [12,20,30,31,33, 42].

5.4 Space complexities

Finally, let us consider the memory requirements of both algorithms. For the GaussSieve algorithm, the total space complexity is clearly dominated by the memory required to store all list vectors. 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. Figure 3d illustrates the experimental space complexities of both algorithms for various dimensions. As expected, for large n the factor t dominates, but the polynomial factor n is not insignificant either. Even in dimension 70, we have $t \approx 523$ and $n = 70$, so that $\frac{t}{n} \approx 7.5$ is significantly smaller than the estimated increase of a factor $t \approx 523$ based only on looking at the exponents.

6 Reducing the space complexity with probing

Using locality-sensitive hashing, we showed how to obtain an exponential speed-up at the cost of exponentially more storage. To amplify the gap between the collision probabilities p_1 and p_2, we used many independent hash tables t to see if a vector $\boldsymbol{w}$ collides with $\boldsymbol{v}$ in at least one of these tables. Since t grows exponentially with n, and each table contains all list vectors, for large parameters the space complexity is dominated by the term $t \cdot N$.

In 2006, Panigraphy [37] proposed that instead of using many different hash tables and checking only one bucket in each table for candidates, one could also go through several hash buckets in each hash table and use fewer hash tables to get a similar quality for the list of candidates. So instead of selecting a new bucket by moving to a new and independent hash table, we select a new bucket by moving to a different bucket in the same hash table.

Construction. To illustrate how this method might work, consider the following modification to the HashSieve algorithm. In each hash table i, instead of only checking the bucket labeled $h_i(\pm \boldsymbol{v}) \in \{0,1\}^k$ for candidates, we also check buckets labeled $h_i(\pm \boldsymbol{v}) \oplus \boldsymbol{e}_j$ for all $j \in [k]$, where $\boldsymbol{e}_j$ is the jth unit vector in k dimensions and $\oplus$ represents 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.

Heuristics. To analyze the effect of this modification on the algorithm, for now let us again make the simplifying assumption that reducing vectors have an angle of at most 60° with $\boldsymbol{v}$, and non-reducing vectors have an angle of exactly 90° with $\boldsymbol{v}$. For reducing 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 $(\frac{2}{3})^k + k(\frac{2}{3})^{k-1}(\frac{1}{3})^1 = (\frac{2}{3})^k \left[1 + \frac{k}{2}\right]$. For non-reducing vectors, the probability that two vectors land in the exact same bucket is $(\frac{1}{2})^k$, and the probability that two vectors land in buckets differing in at most one bit is $(\frac{1}{2})^k \left[1 + k\right]$. 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 good vector increases by a factor $1 + \frac{k}{2}$.
– The probability of finding a bad vector increases by a factor $1 + k$.

If we use probing and reduce t by a factor $1 + \frac{k}{2}$, we thus expect the probability of finding good vectors to be the same as when we do not use probing and use the original value of t. For bad vectors the probability of becoming a candidate vector in this new setting increases by a factor $\frac{1+k}{1+k/2} < 2$. Overall we therefore expect that without further modifications the number of comparisons between list vectors increases by a factor less than 2 (thus also increasing the time complexity by a factor less than 2) while the space complexity decreases by a factor $1 + \frac{k}{2} = O(n)$.[2]

Results. Although what we described above is just a basic, naive technique for probing the hash tables, we can already see significant gains in the space complexity. For instance, in dimension 70 without probing we would have $k = 15$ and $t = 523$ if we round the values of Theorem 1 to the nearest integer, while with probing we would have $k = 15$ and $t \approx 62$, i.e., a factor 8.5 fewer hash tables. Experimentally the maximum list size in this setting is approximately 77 000, and assuming that pointers to list vectors consume 4 bytes of memory, this leads to a space complexity of at least 161 MB to store all hash tables without probing, and 19 MB to store all hash tables with probing. The number of inner products computed between list vectors increases (experimentally) by less than 60%, and the overall time complexity increases by less than 40%. Note that storing the list vectors requires slightly more than 21 MB in both cases, so with one level of probing the space complexity of the hash tables is already less than that of the list of vectors. Figure 3 further illustrates how the time, space, and list sizes change by using probing in dimensions 60 up to 70.

[2] While we assumed that reduced vectors always have an angle of 90° to estimate that the time complexity increases by at most a factor 2, without this assumption (only using the fact that the angle is always at most 90°) this factor would be even smaller.

Extensions. This procedure of probing adjacent buckets (buckets at Hamming distance 1) can trivially be generalized to considering all hash buckets which differ from the hash value $h(\pm \boldsymbol{v}) \in \{0,1\}^k$ in at most $0 \leq \ell \leq k$ 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 $1 + \frac{1}{2}k + \frac{1}{4}\binom{k}{2} + \dots + \frac{1}{2^\ell}\binom{k}{\ell} = O(n^\ell)$ and an increase in the time complexity by less than a factor $2^\ell = O(1)$. For instance, in dimension 100 without probing we would have $k = 22$ and $t = 7643$, with one level of probing ($\ell = 1$) we would have $t = 637$, and with two levels of probing ($\ell = 2$) we would have $t = 110$. For $\ell = 2$ the space complexity of the hash tables is thus reduced by a factor almost 70 at the cost of a factor of at most 4 in the 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 go through. If $\boldsymbol{a}_{i,j}^T \boldsymbol{v} \approx 0$ then it makes more sense to consider the bucket labeled $h_i(\pm \boldsymbol{v}) \oplus \boldsymbol{e}_j$ than if $\boldsymbol{a}_{i,j}^T \boldsymbol{v} \gg 0$ or $\boldsymbol{a}_{i,j}^T \boldsymbol{v} \ll 0$. More generally, given $\boldsymbol{v}$ and any bucket $\boldsymbol{b} \in \{0,1\}^k$, we can heuristically compute the probability that a reducing vector is in $\boldsymbol{b}$ and only check those buckets with the highest probabilities. For more details, see e.g. [28].

7 Locality-sensitive hashing and lattice algorithms

Let us finally mention some open problems related to applying locality-sensitive hashing to sieving and to other lattice algorithms.

7.1 L_2-based hashing

Instead of using angular hashes, one could also try using locality-sensitive hash families designed for the Euclidean norm. If 'good' vectors in a list L of size N are at distance at most R from $\boldsymbol{v}$, and 'bad' vectors are at distance at least $c \cdot R$ from $\boldsymbol{v}$ for $c > 1$, then one can use e.g. the method of Datar et al. [10] to find nearest neighbors in time $N^{1/c}$, or use the asymptotically near-optimal algorithm of Andoni and Indyk [5] to solve nearest neighbor in time $N^{1/c^2 + o(1)}$. Under the assumption that reducing vectors have an angle $60°$ with $\boldsymbol{v}$ and non-reducing vectors are orthogonal to $\boldsymbol{v}$, and assuming that all vectors have the same Euclidean norms, we have that for reducing vectors $\boldsymbol{w}$, $\|\boldsymbol{v} - \boldsymbol{w}\| \approx \|\boldsymbol{v}\|$ while for non-reducing vectors $\boldsymbol{w}$, $\|\boldsymbol{v} - \boldsymbol{w}\| \approx \sqrt{2} \cdot \|\boldsymbol{v}\|$. One might thus conjecture that we could apply these results with $c \approx \sqrt{2}$ in the terminology of approximate nearest neighbor search. This would imply that the method of Datar et al. does not lead to a bigger speed-up than angular hashing, but that the method of Andoni and Indyk might find close vectors in the list in time $\sqrt{N}$.

Besides the question whether we can actually solve nearest neighbor in time $\sqrt{N}$ when not all reduced vectors are actually orthogonal and have the same length, some drawbacks of the L_2-hashing method of Andoni and Indyk are:

(i) The vectors in the list L shrink over time, and thus the target distance at which we want to find the closest neighbor decreases over time. Note that for the angular reductions, we are always looking for vectors with angle $60°$ and this conveniently does not change as vectors get shorter.

(ii) Computing hashes with the algorithm of Andoni and Indyk is much more involved than computing angular hashes, which only involves computing (sparse) inner products. Therefore the hidden constants may be significantly larger than those of the angular hash-based approach.

Regardless of these drawbacks, if the constant in the exponent can somehow heuristically be reduced, then even if the hidden constants are big, this might be of theoretical interest.

7.2 Other sieving algorithms

Besides the GaussSieve of Micciancio and Voulgaris (Section 3) and the sieve algorithm of Nguyen and Vidick (Appendix B), there are various other sieve algorithms for which hashing may lead to speed-ups. One could try to apply the same techniques to the various other heuristic sieves [6, 48, 49] and provable sieves [4, 33, 35, 40] in the literature. For the sieve algorithms with provable bounds on the time and space complexities, note that we used some heuristic assumptions to obtain our results, which might make obtaining non-heuristic, provable bounds with hashing somewhat challenging.

7.3 The Voronoi cell algorithm

The Voronoi cell algorithm of Micciancio and Voulgaris [32] has a provable and practical time complexity of $2^{2n+o(n)}$ for solving SVP and CVP, using $2^{n+o(n)}$ space to store a representation of the Voronoi cell of the lattice. Using locality-sensitive hashing may speed up some parts of this algorithm as well, but this does not seem to decrease the leading constant in the time exponent, as various other parts of their algorithm cannot be sped up with hashing. Note however that for solving the closest vector problem with preprocessing (CVPP), these other routines are not needed anymore, and their CVPP algorithm of complexity $2^{2n+o(n)}$ basically comes down to searching the list of relevant vectors (of which there are $2^{n+o(n)}$) at most $2^{n+o(n)}$ times sequentially. With hashing one may be able to reduce the time complexity of one search from $O(N)$ to $O(N^{1-\alpha}) = O(N^{0.6218})$ (where the value of α can be found in Corollary 1 in Appendix A) at the cost of $N^{1.6218}$ space. This leads to the following proposition:

Proposition 2. *Combining the Voronoi cell algorithm [32] with locality-sensitive hashing, we can heuristically solve CVPP in time and space* $2^{1.6218n+o(n)}$.

Although this slightly improves the theoretical time complexity of the Voronoi algorithm for CVPP, due to the huge space requirements and the high practical time complexity this still does not seem to make the Voronoi algorithm competitive with enumeration or sieving.

Acknowledgments

The author is grateful to Meilof Veeningen and Niels de Vreede for their help and advice with implementation issues and debugging the code used for experiments. The author further thanks Daniel J. Bernstein, Marleen Kooiman, Tanja Lange, Joop van de Pol, and Benne de Weger for various insightful suggestions and comments. The author thanks Michele Mosca for funding a research visit to Waterloo to collaborate on lattices and quantum algorithms, which indirectly inspired work on this topic. The author also thanks Stacey Jeffery and John M. Schanck for inspiring discussions there.

References

1. Achlioptas, D.: Database-Friendly Random Projections. In: PODS, pp. 274–281 (2001)
2. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: STOC, pp. 99–108, (1996)
3. Ajtai, M.: The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In: STOC, pp. 10–19 (1998)
4. Ajtai, M., Kumar, R., Sivakumar, D.: A Sieve Algorithm for the Shortest Lattice Vector Problem. In: STOC, pp. 601–610 (2001)

5. Andoni, A., Indyk, P.: Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In: FOCS, pp. 459–468 (2006)
6. Becker, A., Gama, N., Joux, A.: A sieve algorithm based on overlattices. In: ANTS, pp. 49–70 (2014)
7. Bernstein, D. J., Buchmann, J., Dahmen, E.: Post-Quantum Cryptography (2009)
8. Charikar, M. S.: Similarity Estimation Techniques from Rounding Algorithms. In: STOC, pp. 380–388 (2002)
9. Conway, J. H., Sloane, N. J. A.: Sphere Packings, Lattices and Groups (1999)
10. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In: SCG, pp. 253–262 (2004)
11. Fincke, U., Pohst, M.: Improved Methods for Calculating Vectors of Short Length in a Lattice. Mathematics of Computation 44(170), pp. 463–471 (1985)
12. 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)
13. Gama, N., Nguyen, P. Q.: Predicting Lattice Reduction. In: EUROCRYPT, pp. 31–51 (2008)
14. Gama, N., Nguyen, P. Q., Regev, O.: Lattice Enumeration Using Extreme Pruning. In: EUROCRYPT, pp. 257–278 (2010)
15. Gama, N., Izabachene, M., Nguyen, P. Q., Xie, X.: Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions. Cryptology ePrint Archive (2014)
16. Gentry, C.: Fully Homomorphic Encryption using Ideal Lattices. In: STOC, pp. 169–178 (2009)
17. Hanrot, G., Pujol, X., Stehlé, D.: Algorithms for the Shortest and Closest Lattice Vector Problems. In: IWCC, pp. 159–190 (2011)
18. Hoffstein, J., Pipher, J., Silverman, J. H.: NTRU: A Ring-Based Public Key Cryptosystem. In: ANTS, pp. 267–288 (1998)
19. Indyk, P., Motwani, R.: Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In: STOC, pp. 604–613 (1998)
20. Ishiguro, T., Kiyomoto, S., Miyake, Y., Takagi, T.: Parallel Gauss Sieve Algorithm: Solving the SVP Challenge over a 128-Dimensional Ideal Lattice. In: PKC, pp. 411–428 (2014)
21. Kannan, R.: Improved Algorithms for Integer Programming and Related Lattice Problems. In: STOC, pp. 193–206 (1983)
22. Khot, S.: Hardness of approximating the shortest vector problem in lattices. In: FOCS, pp. 126–135 (2004)
23. Klein, P.: Finding the closest lattice vector when it's unusually close. In: SODA, pp. 937–941 (2000)
24. Kleinjung, T.: Private communication (2014)
25. Lenstra, A. K., Lenstra, H. W., Lovász, L.: Factoring Polynomials with Rational Coefficients. Mathematische Annalen 261(4), pp. 515–534 (1982)
26. Li, P., Hastie, T. J., Church, K. W.: Very Sparse Random Projections. In: KDD, pp. 287–296 (2006)
27. Lindner, R., Peikert, C.: Better Key Sizes (and Attacks) for LWE-Based Encryption. In: CT-RSA, pp. 319–339 (2011)
28. 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)
29. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: EUROCRYPT, pp. 1–23 (2010)
30. Mariano, A., Timnat, S., Bischof, C.: Lock-free GaussSieve for Linear Speedups in Parallel High Performance SVP Calculation. In: SBAC-PAD (2014)
31. Mariano, A., Dagdelen, Ö., Bischof, C.: A Comprehensive Empirical Comparison of Parallel ListSieve and GaussSieve. In: APCI&E (2014)
32. Micciancio, D., Voulgaris, P.: A Deterministic Single Exponential Time Algorithm for Most Lattice Problems based on Voronoi Cell Computations. In: STOC, pp. 351–358 (2010)
33. Micciancio, D., Voulgaris, P.: Faster Exponential Time Algorithms for the Shortest Vector Problem. In: SODA, pp. 1468–1480 (2010)
34. Milde, B., Schneider, M.: A Parallel Implementation of GaussSieve for the Shortest Vector Problem in Lattices. In: PaCT, pp. 452–458 (2011)
35. Nguyen, P. Q., Vidick, T.: Sieve Algorithms for the Shortest Vector Problem are Practical. J. Math. Crypt. 2(2), pp. 181–207 (2008)
36. Plantard, T., Schneider, M.: Ideal Lattice Challenge. Online at `http://latticechallenge.org/ideallattice-challenge/` (2014)
37. Panigraphy, R.: Entropy based nearest neighbor search in high dimensions. In: SODA, pp. 1186–1195 (2006)
38. Pohst, M. E.: On the Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications. ACM SIGSAM Bulletin 15(1), pp. 37–44 (1981)
39. van de Pol, J., Smart, N. P.: Estimating Key Sizes for High Dimensional Lattice-Based Systems. In: IMACC, pp. 290–303 (2013)

40. Pujol, X., Stehlé, D.: Solving the Shortest Lattice Vector Problem in Time $2^{2.465n}$. Cryptology ePrint Archive (2009)
41. Regev, O.: Lecture Notes on Lattices in Computer Science (2004)
42. Schneider, M.: Analysis of Gauss-Sieve for Solving the Shortest Vector Problem in Lattices. In: WALCOM, pp. 89–97 (2011)
43. Schneider, M.: Sieving for Short Vectors in Ideal Lattices. In: AFRICACRYPT, pp. 375–391 (2013)
44. Schneider, M., Gama, N., Baumann, P., Nobach, L.: SVP Challenge. Online at `http://latticechallenge.org/svp-challenge` (2014)
45. Schnorr, C.-P.: A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms. Theoretical Computer Science 53(2), pp. 201–224 (1987)
46. Schnorr, C.-P., Euchner, M.: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. Mathematical Programming 66(2), pp. 181–199 (1994)
47. Shoup, V.: Number Theory Library (NTL), v6.2. Online at `http://www.shoup.net/ntl/` (2014)
48. Wang, X., Liu, M., Tian, C., Bi, J.: Improved Nguyen-Vidick Heuristic Sieve Algorithm for Shortest Vector Problem. In: ASIACCS, pp. 1–9 (2011)
49. 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 reducing vector $\boldsymbol{w}$ collides with $\boldsymbol{v}$ in at least one of the t hash tables is at least constant in n:
$$p_1^* = \mathbb{P}(\text{good vectors collide}) \geq 1 - \varepsilon. \qquad (\varepsilon \in (0,1), \varepsilon \neq \varepsilon_n) \tag{8}$$
2. The average probability that a non-reducing vector $\boldsymbol{w}$ collides with $\boldsymbol{v}$ in at least one of the t hash tables is exponentially small:
$$p_2^* = \mathbb{P}(\text{bad vectors collide}) \leq O(N^{-0.3782}). \tag{9}$$
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 $\text{poly}(n) \cdot N$ times, so after substituting $N = 2^{0.2075n+o(n)}$ this leads to the following asymptotic time and space complexities:

- Time (hashing): $\tilde{O}(t \cdot N) = 2^{0.337n+o(n)}$.
- Time (searching): $O(N^2 \cdot p_2^*) = 2^{0.337n+o(n)}$.
- Space: $O(t \cdot N) = 2^{0.337n+o(n)}$.

The remaining part of this section is dedicated to proving Equations (8) and (9). We first prove that reducing vectors often collide in at least one of the hash tables, given that k is a suitable function of t (Section A.1). We then show how p_2^* scales as a function of k and t (Section A.2) and how to choose k and t to minimize the overall time complexity (Section A.3). Finally we describe how to obtain the trade-off between the space and time complexities of Figure 1 by choosing k and t slightly differently (Section A.4).

A.1 Reducing vectors collide with constant probability

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

$$p_1^* = \mathbb{E}_{\Theta \in (0, \frac{\pi}{3})} \left[p^*(\Theta)\right] = \mathbb{E}_{\Theta \in (0, \frac{\pi}{3})} \left[1 - \left(1 - \left(1 - \frac{\Theta}{\pi}\right)^k\right)^t\right], \tag{10}$$

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:

$$p_1^* \geq 1 - \left(1 - \frac{\ln(1/\varepsilon)}{t}\right)^t \geq 1 - \exp(-\ln(1/\varepsilon)) = 1 - \varepsilon. \tag{11}$$

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 the following natural heuristic instead.

Heuristic 1 *The angle $\Theta(\boldsymbol{v}, \boldsymbol{w})$ between two non-reducing vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ follows the same distribution as the distribution of angles $\Theta(\boldsymbol{v}, \boldsymbol{w})$ obtained by fixing $\boldsymbol{v}$ and drawing $\boldsymbol{w}$ uniformly at random from Ω, where*

$$\Omega = \left\{\boldsymbol{w} \in \mathbb{R}^n : \|\boldsymbol{w}\| = 1 \text{ and } \theta(\boldsymbol{v}, \boldsymbol{w}) \in \left(\tfrac{\pi}{3}, \tfrac{\pi}{2}\right)\right\}. \tag{12}$$

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 various literature, e.g. in [9, 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*

$$A_n(R) = \frac{2\pi^{n/2}}{\Gamma(n/2)} R^{n-1}. \tag{13}$$

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, the probability density function $f(\theta)$ of angles between non-reducing vectors satisfies*

$$f(\theta) = \sqrt{\frac{2n}{\pi}}\,(\sin\theta)^{n-2}\left[1 + o(1)\right] = 2^{\log_2(\sin\theta)n + o(n)}. \tag{14}$$

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

$$w_2^2 + \cdots + w_n^2 = 1 - \cos^2\theta = \sin^2\theta. \tag{15}$$

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

$$f(\theta) = \frac{1}{M} A_{n-1}(\sin\theta), \qquad M = \int_{\pi/3}^{\pi/2} A_{n-1}(\sin\phi) d\phi. \tag{16}$$

For the normalizing constant M, note that integrating the given expression from 0 to $\frac{\pi}{2}$ would give us exactly half the hypersurface area of the n-dimensional hypersphere of radius 1, and the contribution of angles less than $\frac{\pi}{3}$ is negligible for large n. Writing out the expressions for $A_{n-1}(\sin\theta)$ and $M \sim A_n(1)$, we therefore obtain

$$f(\theta) = \frac{A_{n-1}(\sin\theta)}{(1-o(1))\frac{1}{2}A_n(1)} = \frac{2}{\sqrt{\pi}} \cdot \frac{\Gamma(\frac{n}{2})}{\Gamma(\frac{n-1}{2})} \cdot (\sin\theta)^{n-2}\left[1 + o(1)\right]. \tag{17}$$

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

We are now ready to prove the main result, showing that bad collisions 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*

$$U(\theta) = \log_2(\sin\theta) + \frac{c_t}{\gamma_2}\log_2\left(1 - \frac{\theta}{\pi}\right). \tag{18}$$

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

$$p_2^* = \mathbb{P}(\textit{bad vectors collide}) \leq O(N^{-\alpha}), \tag{19}$$

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

$$\alpha = \frac{1}{c_n}\left[-c_t - \max_{\theta \in (\frac{\pi}{3}, \frac{\pi}{2})} U(\theta)\right] + o(1). \tag{20}$$

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

$$p_2^* = \mathbb{E}_{\Theta \in (\frac{\pi}{3}, \frac{\pi}{2})}\left[p^*(\Theta)\right] = \int_{\pi/3}^{\pi/2} f(\theta) p^*(\theta) d\theta. \tag{21}$$

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

$$p_2^* = \sqrt{\frac{2n}{\pi}} \int_{\pi/3}^{\pi/2} (\sin\theta)^{n-2}\left[1 + o(1)\right]\left[1 - \left(1 - \left(1 - \frac{\theta}{\pi}\right)^k\right)^t\right] d\theta. \tag{22}$$

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 $\delta = O(n^{-1/2})$:

$$p_2^* = \underbrace{\int_{\pi/3}^{\pi/3+\delta} f(\theta)p^*(\theta)d\theta}_{I_1} + \underbrace{\int_{\pi/3+\delta}^{\pi/2} f(\theta)p^*(\theta)d\theta}_{I_2}. \tag{23}$$

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

$$I_1 \leq \mathrm{poly}(n) \cdot \delta(1-\varepsilon)\sin^{n-2}\left(\frac{\pi}{3}+u\right). \tag{24}$$

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

$$I_1 \leq 2^{-\gamma_1 n+o(n)}\,(1+O(\delta))^n = 2^{-\gamma_1 n+o(n)}. \tag{25}$$

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

$$I_2 \leq \mathrm{poly}(n) \cdot t \int_{\pi/3+u}^{\pi/2} (\sin\theta)^{n-2}\left(1-\frac{\theta}{\pi}\right)^k d\theta \tag{26}$$

$$\leq \int_{\pi/3}^{\pi/2} 2^{n\log_2(\sin\theta)+k\log_2(1-\frac{\theta}{\pi})+c_t n+o(n)} d\theta. \tag{27}$$

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:

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

Bounding $p_2^* = I_1 + I_2$. Combining the results from (25) and (28), we have

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

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

$$-\gamma_1 \leq -\alpha c_n + o(1). \tag{30}$$

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

The first inequality is automatically satisfied if $\alpha \leq 1$ and $N = 2^{\gamma_1 n+o(n)}$. For the second inequality, we isolate α to obtain

$$\alpha \leq \frac{1}{c_n}\left[-c_t - \max_{\theta\in(\frac{\pi}{3},\frac{\pi}{2})} U(\theta)\right] + o(1). \tag{32}$$

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

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): $O(t \cdot N) = 2^{(c_n + c_t)n + o(n)}$.
- Time (searching): $O(N^2 \cdot \hat{p}_2) = 2^{(c_n + (1-\alpha)c_n)n + o(n)}$.
- Time (overall): $2^{(c_n + \max\{c_t, (1-\alpha)c_n\})n + o(n)}$.
- Space: $O(t \cdot N) = 2^{(c_n + c_t)n + o(n)}$.

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

$$c_{\text{time}} = c_n + \max\{c_t, (1-\alpha)c_n\}, \tag{33}$$

$$c_{\text{space}} = c_n + c_t. \tag{34}$$

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[3] to obtain the following corollary.

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

$$\theta^* \approx 0.458921\pi, \qquad \alpha \approx 0.378163, \tag{35}$$

$$c_{\text{time}} \approx 0.336562, \qquad c_{\text{space}} \approx 0.336562. \tag{36}$$

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 Locality-sensitive hashing applied to Nguyen and Vidick's sieve

Although in the main text we focused applying locality-sensitive hashing to the GaussSieve algorithm of Micciancio and Voulgaris, the same ideas can be applied to other sieve algorithms such as the one of Nguyen and Vidick [35, Algorithms 4 and 5]. Since for the GaussSieve there are no heuristic bounds on the time complexity, and since such bounds do exist for the sieve algorithm of Nguyen and Vidick, we highlight how the same techniques can be applied to their algorithm to obtain similar speedups with "provable heuristic bounds."

First, we recall that the sieve algorithm of Nguyen and Vidick [35] starts with a long list L_0 of reasonably long, randomly sampled lattice vectors $\boldsymbol{v}$, 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

Algorithm 3 Nguyen and Vidick's lattice sieve

```
1: Compute the maximum norm $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
2: Initialize an empty list $L_{m+1}$ and an empty list of centers $C_{m+1}$
3: for each $\boldsymbol{v} \in L_m$ do
4:     if $\|\boldsymbol{v}\| \leq \gamma R$ then
5:         Add $\boldsymbol{v}$ to the list $L_{m+1}$
6:         Continue the loop
7:     for each $\boldsymbol{w} \in C_{m+1}$ do
8:         Reduce $\boldsymbol{v}$ with $\boldsymbol{w}$
9:         if $\boldsymbol{v}$ has changed then
10:            Add $\boldsymbol{v}$ to the list $L_{m+1}$
11:            Continue the outermost loop
12:    Add $\boldsymbol{v}$ to the centers $C_{m+1}$
```

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}. This algorithm is described in a somewhat simplified form in Algorithm 3. The reduction step in Line 8 is implemented as:

$$\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2 : \quad \text{if } \|\boldsymbol{u}_1 \pm \boldsymbol{u}_2\| \leq \gamma R \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2. \tag{37}$$

To obtain the heuristic estimate $2^{0.415n+o(n)}$ for the time complexity (and $2^{0.208n+o(n)}$ for the space complexity), Nguyen and Vidick let γ approach 1 in their analysis. This means that all vectors with a length significantly shorter than R are automatically added to L_{m+1}, and the bottleneck on the time complexity comes from those vectors $\boldsymbol{v}$ with $\gamma R < \|\boldsymbol{v}\| \leq R$, i.e., the vectors $\boldsymbol{v}$ 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 so the reduction method described in (37) is almost equivalent to the reduction step of the GaussSieve in (6). Note that in the limiting case of $\gamma \to 1$, the sieve of Nguyen and Vidick searches exactly[4] for vectors $\boldsymbol{w} \in C_{m+1}$ whose angle with $\boldsymbol{v}$ is less than $60°$, and we can apply the same techniques to this algorithm to obtain Algorithm 4, achieving the asymptotic speed-ups described in Theorem 1 and Figure 1.

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

[4] For the extreme case of $\gamma = 1$, one would have $\|\boldsymbol{v}\| = \|\boldsymbol{w}\| = R = \gamma R$ and thus the condition $\|\boldsymbol{v} - \boldsymbol{w}\| \leq \gamma R$ is exactly equivalent to $\theta(\boldsymbol{v}, \boldsymbol{w}) \leq 60°$.

Algorithm 4 Nguyen and Vidick's lattice sieve with hashing

```
 1: Compute the maximum norm $R = \max_{\boldsymbol{v} \in L_m} \|\boldsymbol{v}\|$
 2: Initialize an empty list $L_{m+1}$ and an empty list of centers $C_{m+1}$
 3: Initialize $k$ empty hash tables $T_i$
 4: Sample $k \cdot t$ random hash vectors $\boldsymbol{a}_{i,j}$
 5: for each $\boldsymbol{v} \in L_m$ do
 6:     if $\|\boldsymbol{v}\| \leq \gamma R$ then
 7:         Add $\boldsymbol{v}$ to the list $L_{m+1}$
 8:         Continue the loop
 9:     Obtain the set of candidates $C = \bigcup_{i=1}^{t} T_i[h_i(\pm \boldsymbol{v})]$
10:     for each $\boldsymbol{w} \in C$ do
11:         Reduce $\boldsymbol{v}$ with $\boldsymbol{w}$
12:         if $\boldsymbol{v}$ has changed then
13:             Add $\boldsymbol{v}$ to the list $L_{m+1}$
14:             Continue the outermost loop
15:     Add $\boldsymbol{v}$ to the centers $C_{m+1}$
16:     Add $\boldsymbol{v}$ to all $k$ hash tables $T_i$
```