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 an algorithm with (conjectured) heuristic time and space complexities of $2^{0.3366n+o(n)}$. Experiments with the GaussSieve-variant show that in moderate dimensions the proposed HashSieve algorithm already outperforms the GaussSieve, and the practical increase in the space complexity is much 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 [10], average-case hardness guarantees [3], the existence of lattice-based fully homomorphic encryption schemes [16], and efficient cryptographic primitives like NTRU [17]. 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 [28, 39].

Finding short vectors. Given a basis $\{\boldsymbol{b}_1, \dots, \boldsymbol{b}_n\} \subset \mathbb{R}^n$ 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, 21]. For large approximation

factors, various fast algorithms for finding short vectors are known, such as the lattice basis reduction algorithms LLL [26] and BKZ [43, 44]. 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 [14, 20, 38, 44]. This method has a low (polynomial) space complexity, but its runtime is superexponential ($2^{\Omega(n \log n)}$), which is known to be suboptimal: sieving [5], the Voronoi cell algorithm [32], 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 these other methods in moderate dimensions n [34].

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 $2^{1.24n+o(n)}$ [35, 40]. Heuristic analyses of sieving algorithms further suggest that one may be able to solve SVP in time $2^{0.42n+o(n)}$ and space $2^{0.21n+o(n)}$ [7, 33, 35], or optimizing for time, in time $2^{0.38n+o(n)}$ and space $2^{0.29n+o(n)}$ [7, 45, 46]. Other works have shown how to speed up sieving in practice [11, 15, 19, 29, 30, 41], and sieving recently made its way to the top 25 of the SVP challenge hall of fame [42], using the GaussSieve algorithm [23, 33].

Sieving in ideal lattices. The potential of sieving is further illustrated by recent results in ideal lattices [11, 19]; 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 [11, 19, 37]. This is higher than the highest dimension for which enumeration was used to find a record in either lattice challenge [37, 42], 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 [12, 18], a technique from the field of nearest neighbor searching. In short, for each list vector $\boldsymbol{w}$ we store low-dimensional, lossy *sketches* (hashes), such that vectors that are

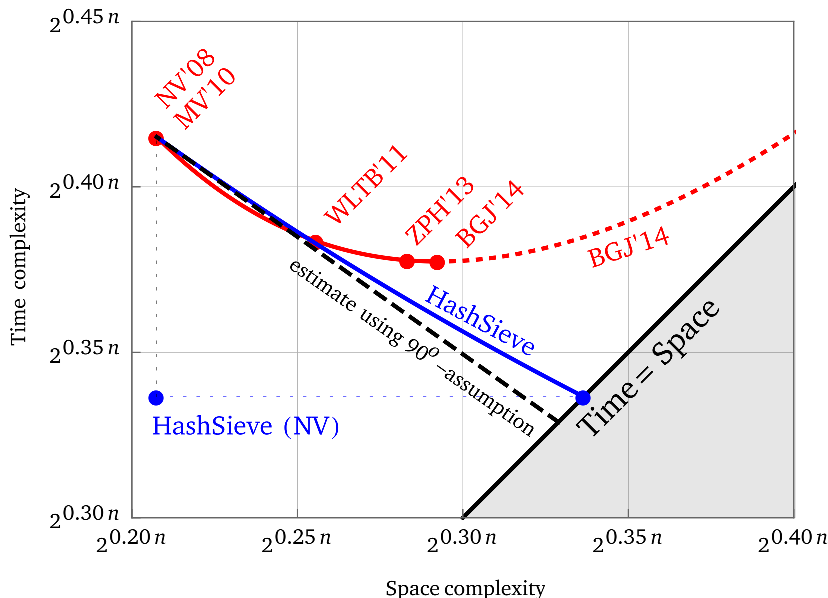


Fig. 1. The heuristic space-time trade-off of various heuristic sieves from the literature (red), and the heuristic trade-off between the space and time complexities obtained with the HashSieve (blue curve). For the NV-sieve, we can further process the hash tables sequentially to obtain a speedup rather 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 [35]; MV'10 [33]; WLTB'11 [45]; ZPH'13 [46]; BGJ'14 [7].

nearby have a higher probability of having the same sketch (hash value) 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 sketch (hash value) in one of the hash tables. 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 both bounded by $2^{0.3366n+o(n)}$, while tuning the parameters differently, we get a continuous heuristic trade-off between the space and time complexities 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 [35], 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 Hash-Sieve 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 significantly 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 [35, Section 4.2.2], there are two main differences:

- Nguyen and Vidick considered LSH families based on *Euclidean distances* [6], while we will argue that it seems more natural to consider hash families based on *angular distances* or *cosine similarities* [12].
- 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 in the GaussSieve 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°.

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 and other nearest neighbor search methods in the context of lattice sieving [8, 9, 25, 31], and an open problem is whether other lattice algorithms (e.g. provable sieving algorithms, the Voronoi cell algorithm) may benefit from related techniques as well.

Roadmap. In Section 2 we describe the technique of (angular) LSH for finding near(est) neighbors, and Section 3 describes how to apply these techniques to the GaussSieve. Section 4 states the main result regarding the time and space complexities of sieving using angular LSH, and describes the technique of probing. In Section 5 we finally describe experiments performed using the GaussSieve-based HashSieve, and possible consequences for the estimated complexity of SVP in high dimensions. The full version [24] contains details on how angular LSH may be combined with the NV-sieve, and how the memory can be reduced to obtain a memory-wise asymptotically superior NV-sieve-based HashSieve.

2 Locality-sensitive hashing

2.1 Introduction

The near(est) neighbor problem is the following [18]: 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 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" [18].

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* [18]. 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[1], and the set U below may be thought of as (a subset of) the natural numbers $\mathbb{N}$.

Definition 1. *[18] 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 (and that of $\boldsymbol{v}$).

[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.

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 $h_1, \dots, h_k \in \mathcal{H}$ and a one-to-one mapping $f : U^k \to U$, and defining $h \in \mathcal{H}'$ as $h(\boldsymbol{v}) = f(h_1(\boldsymbol{v}), \dots, h_k(\boldsymbol{v}))$. 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 $h_1, \dots, h_t \in \mathcal{H}$, 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 $j = 1, 2$.[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 follows:

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

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 [18]. 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 $h_1, \dots, h_t$. Then, given the list L, we build t different hash tables $T_1, \dots, T_t$, 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

[2] Note that h is strictly not a function and only defines a relation.

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. *[18] Suppose there exists a (r_1, r_2, p_1, p_2)-sensitive hash family $\mathcal{H}$. Then, for a list L of size N, 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 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 $\boldsymbol{v}$: $\tilde{O}(N^\rho)$.
 (3a) Hash evaluations of the query vector $\boldsymbol{v}$: $O(N^\rho)$.
 (3b) List vectors to compare to the query vector $\boldsymbol{v}$: $O(N^\rho)$.

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.

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*, 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 [12]:

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

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 [12]:

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

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.

3 From the GaussSieve to the 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 date, which is further motivated by the extensive attention it has received in recent years [15, 19, 23, 29, 30, 41] and by the fact that the highest sieving record in the SVP challenge database was obtained using (a modification of) the GaussSieve [23, 42]. Note that the same ideas can also be applied to the Nguyen-Vidick sieve [35], which has proven complexity bounds. Details on this combination are in the full version.

3.1 The GaussSieve algorithm

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 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 sampled from a discrete Gaussian over the lattice, using e.g. the sampling algorithm of Klein [22, 33], or popped from the stack. 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:

$$\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2 : \quad \text{if } \|\boldsymbol{u}_1 \pm \boldsymbol{u}_2\| < \|\boldsymbol{u}_1\| \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2. \tag{5}$$

Throughout the execution of the algorithm, the list L is always pairwise reduced w.r.t. (5), 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

Algorithm 1 The GaussSieve algorithm (simplified)

```
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
```

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 therefore not exceed $2^{0.2075n+o(n)}$ [13].

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 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 show a similar asymptotic behavior as the NV-sieve [35], for which the asymptotic time complexity is heuristically known to be quadratic in $|L|$, i.e., of the order $2^{0.415n+o(n)}$. 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 [23].

3.2 The 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 $60°$, 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:

$$\begin{aligned}&\text{Reduce } \boldsymbol{u}_1 \text{ with } \boldsymbol{u}_2 : \\ &\quad \text{if } \theta(\boldsymbol{u}_1, \pm\boldsymbol{u}_2) < 60° \text{ and } \|\boldsymbol{u}_1\| \geq \|\boldsymbol{u}_2\| \text{ then } \boldsymbol{u}_1 \leftarrow \boldsymbol{u}_1 \pm \boldsymbol{u}_2,\end{aligned} \tag{6}$$

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 $2^{0.208n+o(n)}$,

Algorithm 2 The GaussSieve-based 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 if $S = \emptyset$)
6: Obtain the set of candidates $C = \bigcup_{i=1}^{t} T_i[h_i(\boldsymbol{v})]$
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 (unless $\boldsymbol{w} = \mathbf{0}$)
14: **if** $\boldsymbol{v}$ has changed **then**
15: Add $\boldsymbol{v}$ to the stack S (unless $\boldsymbol{v} = \mathbf{0}$)
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

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.

3.3 The HashSieve with angular reductions

Replacing the stronger notion of reduction of (5) by the weaker one of (6), 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. Blue lines in Algorithm 2 indicate modifications to the GaussSieve. 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. This means that we only pay the setup costs of locality-sensitive hashing once, rather than once for each search.

3.4 The (GaussSieve-based) HashSieve algorithm

Finally, note that there seems to be no point in skipping potential reductions in Lines 8 and 9. So while for our intuition and for the theoretical motivation we may consider the case where the reductions are based on (6), in practice we will again reduce vectors based on (5). The algorithm is illustrated in Figure 2.

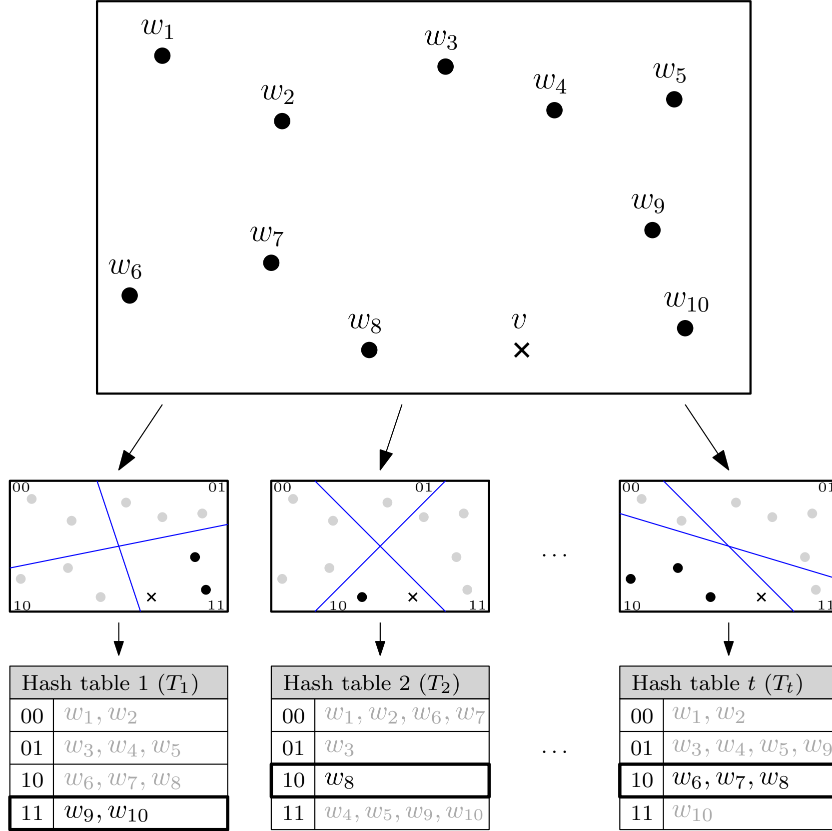


Fig. 2. An example of the HashSieve, using $k = 2$ hyperplanes and $2^k = 4$ buckets in each hash table. Given 10 list vectors $L = \{\boldsymbol{w}_1, \dots, \boldsymbol{w}_{10}\}$ and a target vector $\boldsymbol{v}$, for each of the t hash tables we first compute $\boldsymbol{v}$'s hash value (i.e. compute the region in which it lies), look up vectors with the same hash value, and compare $\boldsymbol{v}$ with those vectors. Here we will try to reduce $\boldsymbol{v}$ with $C = \{\boldsymbol{w}_6, \boldsymbol{w}_7, \boldsymbol{w}_8, \boldsymbol{w}_9, \boldsymbol{w}_{10}\}$ and vice versa.

3.5 Relation with leveled sieving

Overall, the crucial modification going from the GaussSieve to the HashSieve 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 L, we only compare the vector to a much smaller subset of candidates $C \subset L$, which mostly contains good candidates for reduction, and does not contain many of the 'bad' vectors in L which are not useful for reductions anyway.

In a way, the idea of the HashSieve is similar to the technique previously used in two- and three-level sieving [45,46]. 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). The key difference between leveled sieving and our method is in the way the partitions of $\mathbb{R}^n$ are chosen: using giant balls in leveled sieving (similar to the Euclidean LSH method of [6]), and using intersections of half-spaces in the HashSieve.

4 Theoretical results

For analyzing the time complexity of sieving with angular LSH, for clarity of exposition we will analyze the GaussSieve-based HashSieve and 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 angular LSH, we can reduce the time complexity to $2^{0.337n+o(n)}$. Note that although practical experiments in high dimensions seem to verify this assumption [23], in reality it is not known whether the time complexity of the GaussSieve is quadratic in $|L|$. At first sight this therefore may not guarantee a heuristic time complexity of the order $2^{0.337n+o(n)}$. In the full version we illustrate how the same techniques can be applied to the sieve of Nguyen and Vidick [35], for which the heuristic time complexity is in fact known to be at most $2^{0.415n+o(n)}$, and for which we get the same speedup. This implies that indeed, with sieving we can provably solve SVP in time and space $2^{0.337n+o(n)}$ under the same heuristic assumptions of Nguyen and Vidick [35]. For clarity of exposition, in the main text we will continue focusing 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 and space complexities.

4.1 High-dimensional intuition

So for now, suppose that the GaussSieve has a time complexity quadratic in $|L|$ and that $|L| \leq 2^{0.208n+o(n)}$. To estimate the complexities of the HashSieve, we will use the following assumption previously described in [35]:

Heuristic 1 *The angle $\Theta(\boldsymbol{v}, \boldsymbol{w})$ between random sampled/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 list vectors than smaller angles. So one might guess that for two 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 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' between vectors that are unable to reduce each other are *exactly* 90° (and non-reduced angles are at most 60°), we obtain the following estimate for the costs of the HashSieve algorithm.

Proposition 1. *Assuming that reduced vectors are always pairwise orthogonal, the HashSieve with parameters* $k = 0.2075n+o(n)$ *and* $t = 2^{0.1214n+o(n)}$ *heuristically solves SVP in time and space* $2^{0.3289n+o(n)}$. *We further 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 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 $\tilde{O}(|L|) = \tilde{O}(N)$ times, the time complexity is of the order $\tilde{O}(N^{1+\rho}) = 2^{0.3289n+o(n)}$. □

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

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 to optimize the asymptotic time complexity. A rigorous proof of Theorem 1 based on the NV-sieve can be found in the full version.

Theorem 1. *Sieving with angular locality-sensitive hashing with parameters*

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

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. So the simple estimate based on the intuition that in high dimensions "everything is orthogonal" is not far off.

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

For completeness let us briefly explain how for the NV-sieve [35], we can in fact process the hash tables sequentially and eliminate the need of storing exponentially many hash tables in memory, for which full details are given in the full version. To illustrate the idea, recall that in the Nguyen-Vidick sieve we are given a list L of size $2^{0.21n+o(n)}$ of vectors of norm at most R, and we want to build a new list L' of similar size $2^{0.21n+o(n)}$ of vectors of norm at most γR with $\gamma < 1$. To do this, we look at (almost) all pairs of vectors in L, and see if their difference (sum) is short; if so, we add it to L'. As the probability of finding a short vector is roughly $2^{-0.21n+o(n)}$ and we have $2^{0.42n+o(n)}$ pairs of vectors, this will result in enough vectors to continue in the next iterations.

The natural way to apply angular LSH to this algorithm would be to add all vectors in L to t independent hash tables, and to find short vectors to add to

L' we then compute a new vector $\boldsymbol{v}$'s hash value for each of these t hash tables, look for potential short vectors $\boldsymbol{v} \pm \boldsymbol{w}$ by comparing $\boldsymbol{v}$ with the colliding vectors $\boldsymbol{w} \in \bigcup_{i=1}^{t} T_i[h_i(\boldsymbol{v})]$, and process all vectors one by one. This results in similar asymptotic time and space complexities as illustrated above.

The simple but crucial modification that we can make to this algorithm is that we process the tables one by one; we first construct the first hash table, add all vectors in L to this hash table, and look for short difference vectors inside each of the buckets of L to add to L'. The cost of building and processing one hash table is of the order $2^{0.21n+o(n)}$, and the number of vectors found that can be added to L' is of the order $2^{0.08n+o(n)}$. By then deleting the hash table from memory and building new hash tables over and over ($t = 2^{0.13n+o(n)}$ times) we keep building a longer list L' until finally we will again have found $2^{0.21n+o(n)}$ 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 leads to the following result.

Theorem 2. *Sieving with angular locality-sensitive hashing with parameters*

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

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 balances the costs of computing hashes and comparing vectors; the fact that the blue point in Figure 1 does not lie on the "Time = Space"-line does not mean we can further reduce the time complexity.

4.4 Reducing the space complexity with probing

Finally, as the above modification only seems to work with the less practical NV-sieve (and not with the GaussSieve), and since for the GaussSieve-based HashSieve the memory requirement increases exponentially, let us briefly sketch how we can reduce the required amount of memory in practice for the (GaussSieve-based) HashSieve using *probing*. The key observation here is that, as illustrated in Figure 2, we only check one bucket in each hash table for nearby vectors, leading to t hash buckets in total that are checked for candidate reductions. This seems wasteful, as the hash tables contain more information: we also know for instance which hash buckets are next-most likely to contain nearby vectors, which are buckets with very similar hash values. By also probing these buckets in a clever way and checking multiple hash buckets per hash table, we can significantly reduce the number of hash tables t in practice such that in the end we still find as many good vectors. Using ℓ levels of probing (checking all buckets with hash value at Hamming distance at most ℓ to $h(\boldsymbol{v})$) we can reduce t by a factor $O(n^\ell)$ at the cost of increasing the time complexity by a factor at most 2^ℓ. This does not constitute an exponential improvement, but the polynomial reduction in memory may be worthwhile in practice. More details on probing can be found in the full version.

5 Practical results

5.1 Experimental results in moderate dimensions

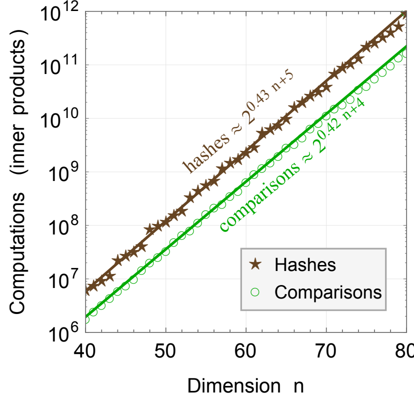
To verify our theoretical analysis, we implemented both the GaussSieve and the GaussSieve-based HashSieve to try to compare the asymptotic trends of these algorithms. For implementing the HashSieve, we note that we can use various simple tweaks to further improve the algorithm's performance. These include:

(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 reducing 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 $\boldsymbol{a}$ 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 hash computations significantly cheaper, while retaining the same performance [1,27]. 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.
(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.

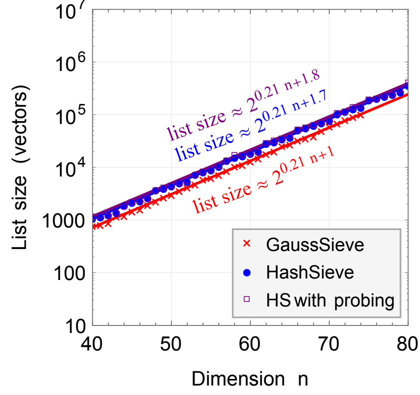
With these tweaks, we performed several experiments of finding shortest vectors using the lattices of the SVP challenge [42]. We generated lattice bases for different seeds and different dimensions using the SVP challenge generator, used NTL (Number Theory Library) to preprocess the bases (LLL reduction), 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 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 try to estimate and compare the asymptotic behaviors of these algorithms. Details on a more optimized implementation of the HashSieve are given in [31].

Dimension (n)	40	45	50	55	60	65	70	75	80
Hash length (k)	9	10	11	12	13	14	15	17	18
Hash tables (t)	36	56	87	137	214	334	523	817	1278
. . . with probing (t_1)	7	9	13	19	28	41	60	88	130

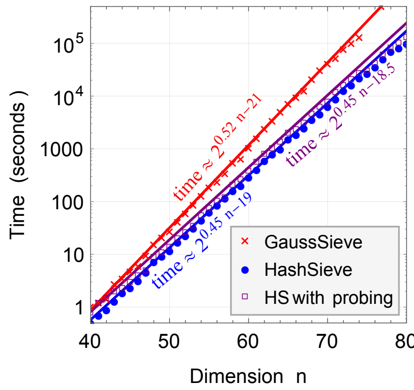
(a) Parameters used in HashSieve experiments, without (t) and with (t_1) probing



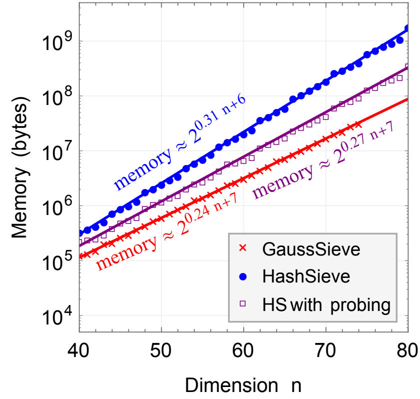
(b) HashSieve computations (no prob.)



(c) List sizes



(d) Time complexities



(e) Space complexities

Fig. 3. Experimental data for the GaussSieve and the HashSieve (with/without probing). Markers indicate experiments, lines and labels represent least-squares fits.

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. 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 in Figure 3 is explained by the fact that k is small but integral, and increases by 1 only once every four/five dimensions.

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 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 [15, 19, 29, 30, 33]. 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 [23, 31]. In high dimensions we therefore expect the coefficient 0.3366 to be accurate. For more details, see [31].

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).

5.2 High-dimensional extrapolations

As explained at the start of this section, the experiments in Section 5.1 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. [31] showed that the HashSieve can be parallelized in a similar fashion as the GaussSieve [29]. 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[3]. 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.

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 [23]. 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 [42]. 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.

[3] 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.

Acknowledgments

The author is grateful to Meilof Veeningen and Niels de Vreede for their help and advice with implementations. The author 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 further thanks Michele Mosca for funding a research visit to Waterloo to collaborate on lattices and quantum algorithms, and the author thanks Stacey Jeffery, Michele Mosca, Joop van de Pol, and John M. Schanck for valuable discussions there. The author also thanks Memphis Depay for his inspiration.

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. Becker, A., Gama, N., Joux, A.: A sieve algorithm based on overlattices. In: ANTS, pp. 49–70 (2014)
8. Becker, A., Gama, N., Joux, A.: Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Preprint (2015)
9. Becker, A., Laarhoven, T.: Efficient sieving in (ideal) lattices using cross-polytopic LSH. Preprint (2015)
10. Bernstein, D. J., Buchmann, J., Dahmen, E.: Post-quantum cryptography (2009)
11. 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)
12. Charikar, M. S.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388 (2002)
13. Conway, J. H., Sloane, N. J. A.: Sphere packings, lattices and groups (1999)
14. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice. Mathematics of Computation 44(170), pp. 463–471 (1985)
15. 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)
16. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC (2009)
17. Hoffstein, J., Pipher, J., Silverman, J. H.: NTRU: A ring-based public key cryptosystem. In: ANTS, pp. 267–288 (1998)
18. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
19. 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)
20. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: STOC, pp. 193–206 (1983)

21. Khot, S.: Hardness of approximating the shortest vector problem in lattices. In: FOCS, pp. 126–135 (2004)
22. Klein, P.: Finding the closest lattice vector when it's unusually close. In: SODA, pp. 937–941 (2000)
23. Kleinjung, T.: Private communication (2014)
24. Laarhoven, T.: Sieving for shortest vectors in lattices using angular locality-sensitive hashing. Full version at `http://eprint.iacr.org/2014/744` (2015)
25. Laarhoven, T., de Weger, B.: Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In: LATINCRYPT (2015)
26. Lenstra, A. K., Lenstra, H. W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), pp. 515–534 (1982)
27. Li, P., Hastie, T. J., Church, K. W.: Very sparse random projections. In: KDD, pp. 287–296 (2006)
28. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: CT-RSA, pp. 319–339 (2011)
29. Mariano, A., Timnat, S., Bischof, C.: Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In: SBAC-PAD (2014)
30. Mariano, A., Dagdelen, Ö., Bischof, C.: A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In: APCI&E (2014)
31. Mariano, A., Laarhoven, T., Bischof, C.: Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In: ICPP (2015)
32. Micciancio, D., Voulgaris, P.: A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In: STOC (2010)
33. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA, pp. 1468–1480 (2010)
34. Micciancio, D., Walter, M.: Fast lattice point enumeration with minimal overhead. In: SODA, pp. 276–294 (2015)
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. Panigraphy, R.: Entropy based nearest neighbor search in high dimensions. In: SODA, pp. 1186–1195 (2006)
37. Plantard, T., Schneider, M.: Ideal lattice challenge. Online at `http://latticechallenge.org/ideallattice-challenge/` (2014)
38. 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)
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, Report 2009/605 (2009)
41. Schneider, M.: Sieving for short vectors in ideal lattices. In: AFRICACRYPT, pp. 375–391 (2013)
42. Schneider, M., Gama, N., Baumann, P., Nobach, L.: SVP challenge. Online at `http://latticechallenge.org/svp-challenge` (2014)
43. Schnorr, C.-P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science 53(2), pp. 201–224 (1987)
44. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Math. Programming 66(2), pp. 181–199 (1994)
45. Wang, X., Liu, M., Tian, C., Bi, J.: Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In: ASIACCS, pp. 1–9 (2011)
46. Zhang, F., Pan, Y., Hu, G.: A three-level sieve algorithm for the shortest vector problem. In: SAC, pp. 29–47 (2013)