Speed-ups and time–memory trade-offs for tuple lattice sieving

Gottfried Herold[1], Elena Kirshanova[1], and Thijs Laarhoven[2]

[1] Laboratoire LIP, ENS de Lyon, France
{gottfried.herold, elena.kirshanova}@ens-lyon.fr
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
mail@thijs.com

Abstract. In this work we study speed-ups and time–space trade-offs for solving the shortest vector problem (SVP) on Euclidean lattices based on tuple lattice sieving.

Our results extend and improve upon previous work of Bai–Laarhoven–Stehlé [ANTS'16] and Herold–Kirshanova [PKC'17], with better complexities for arbitrary tuple sizes and offering tunable time–memory trade-offs. The trade-offs we obtain stem from the generalization and combination of two algorithmic techniques: the configuration framework introduced by Herold–Kirshanova, and the spherical locality-sensitive filters of Becker–Ducas–Gama–Laarhoven [SODA'16].

When the available memory scales quasi-linearly with the list size, we show that with triple sieving we can solve SVP in dimension n in time $2^{0.3588n+o(n)}$ and space $2^{0.1887n+o(n)}$, improving upon the previous best triple sieve time complexity of $2^{0.3717n+o(n)}$ of Herold–Kirshanova. Using more memory we obtain better asymptotic time complexities. For instance, we obtain a triple sieve requiring only $2^{0.3300n+o(n)}$ time and $2^{0.2075n+o(n)}$ memory to solve SVP in dimension n. This improves upon the best double Gauss sieve of Becker–Ducas–Gama–Laarhoven, which runs in $2^{0.3685n+o(n)}$ time when using the same amount of space.

Keywords: lattice-based cryptography, shortest vector problem (SVP), nearest neighbor algorithms, lattice sieving

1 Introduction

Lattice-based cryptography. Over the past few decades, lattice-based cryptography has emerged as a prime candidate for developing efficient, versatile, and (potentially) quantum-resistant cryptographic primitives; e.g. [Reg05,ADPS16]. The security of these primitives relies on the hardness of certain lattice problems, such as finding short lattice vectors. The fastest known method for solving many hard lattice problems is to use (a variant of) the BKZ lattice basis reduction algorithm [Sch87,SE94], which internally uses an algorithm for solving the so-called Shortest Vector Problem (SVP) in lower-dimensional lattices: given a description of a lattice, the Shortest Vector Problem asks to find a shortest non-zero vector in this lattice. These SVP calls determine the complexity of BKZ, and hence an accurate assessment of the SVP hardness directly leads to sharper security estimates and tighter parameter choices for lattice-based primitives.

Algorithms for solving SVP. Currently, state-of-the-art algorithms for solving exact SVP can be classified into two groups, based on their asymptotic time and memory complexities in terms of the lattice dimension n: (1) algorithms requiring super-exponential time ($2^{\omega(n)}$) and poly(n) space; and (2) algorithms requiring both exponential time and space ($2^{\Theta(n)}$). The former includes a family of so-called lattice enumeration algorithms [Kan83,FP85,GNR10], which currently perform best in practice and are used inside BKZ [Sch87,CN11]. The latter class of algorithms includes lattice sieving [AKS01,NV08,MV10], Voronoi-based approaches [AEVZ02,MV10,Laa16] and other techniques [BGJ14,ADRS15]. Due to the superior asymptotic scaling, these latter techniques will inevitably outperform enumeration in sufficiently high dimensions, but the large memory requirement remains a major obstacle in making these algorithms practical.

Heuristic SVP algorithms. In practice, only enumeration and sieving are currently competitive for solving SVP in high dimensions, and the fastest variants of both algorithms are based on *heuristic analyses*: by making certain natural (but unproven) assumptions about average-case behavior of these algorithms, one can (1) improve considerably upon worst-case complexity bounds, thus narrowing the gap between experimental and theoretical results; and (2) apply new techniques, supported by heuristics, to make these algorithms even more viable in practice. For enumeration, heuristic analyses of *pruning* [GNR10,AN17] have contributed immensely to finding the best pruning techniques, and making these algorithms as practical as they are today [LRBN]. Similarly, heuristic assumptions for sieving [NV08,MV10,Laa15a,BDGL16] have made these algorithms much more practical than their best provable counterparts [PS09,ADRS15].

Heuristic sieving methods. In 2008, Nguyen–Vidick [NV08] were the first to show that lattice sieving may be practical, proving that under certain heuristic assumptions, SVP can be solved in time $2^{0.415n+o(n)}$ and space $2^{0.208n+o(n)}$. Micciancio–Voulgaris [MV10] later described the so-called GaussSieve, which is expected to have similar asymptotic complexities as the Nguyen–Vidick sieve, but is several orders of magnitude faster in practice. Afterwards, a long line of work focused on locality-sensitive techniques succeeded in further decreasing the runtime exponent [WLTB11,ZPH13,BGJ14,Laa15a,LdW15,BL16].

Asymptotically the fastest known method for solving SVP is due to Becker–Ducas–Gama–Laarhoven [BDGL16]. Using locality sensitive filters, they give a time–memory trade-off for SVP in time and space $2^{0.292n+o(n)}$, or in time $2^{0.368n+o(n)}$ when using only $2^{0.208n+o(n)}$ memory. A variant can even solve SVP in time $2^{0.292n+o(n)}$ retaining a memory complexity of $2^{0.208n+o(n)}$, but that variant is not compatible with the Gauss Sieve (as opposed to the Nguyen–Vidick sieve) and behaves worse in practice.

Tuple lattice sieving. In 2016, Bai–Laarhoven–Stehlé [BLS16] showed that one can also obtain trade-offs for heuristic sieving methods in the other direction: reducing the memory requirement at the cost of more time. For instance, with a *triple sieve* they showed that one can solve SVP in time $2^{0.481n+o(n)}$, using

$2^{0.189n+o(n)}$ space. Various open questions from [BLS16] were later answered by Herold–Kirshanova [HK17], who proved some of the conjectures from [BLS16], and greatly reduced the time complexity of the triple sieve to $2^{0.372n+o(n)}$. An open question remained whether these complexities were optimal, and whether it would be possible to obtain efficient time–memory trade-offs to interpolate between classical 'double' sieving methods and tuple lattice sieving.

1.1 Contributions

Results. In this work, we study both how to further speed up tuple lattice sieving, and how to obtain the best time–memory trade-offs for solving SVP with sieving[3]. Our contributions include the following main results:

1. For triple sieving, we obtain a time complexity of $2^{0.3588n+o(n)}$ with a memory complexity of $2^{0.1887n+o(n)}$. This improves upon the previous best asymptotic time complexity of $2^{0.3717n+o(n)}$ of Herold–Kirshanova [HK17], and both the time *and* memory are better than the Gauss sieve algorithm of Becker–Ducas–Gama–Laarhoven [BDGL16], which runs in time $2^{0.3685n+o(n)}$ and memory $2^{0.2075n+o(n)}$.
2. For triple sieving with arbitrary time–memory trade-offs, we obtain the trade-off curve depicted in Fig. 1, showing that our triple sieve theoretically outperforms the best double Gauss sieve up to a memory complexity of $2^{0.2437n+o(n)}$ (the intersection point of yellow and blue curves). For instance, with equal memory $2^{0.2075n+o(n)}$ as a double sieve, we can solve SVP in time $2^{0.3300n+o(n)}$, compared to the previous best $2^{0.3685n+o(n)}$ [BDGL16].
3. For larger tuple sizes (i.e., $k \geq 3$), in the regime when the space complexity is restricted to the input-sizes as considered by Bai–Laarhoven–Stehlé [BLS16] and Herold–Kirshanova [HK17], we improve upon all previous results. These new asymptotics are given in Table 3 on page 25.
4. Our experiments on lattices of dimensions 60 to 80 demonstrate the practicability of these algorithms, and highlight possible future directions for further optimizations of tuple lattice sieving.

Techniques. To obtain these improved time–memory trade-offs for tuple lattice sieving, this paper presents the following technical contributions:

1. We generalize the configuration search approach, first initiated by Herold–Kirshanova [HK17], to obtain optimized time–space trade-offs for tuple lattice sieving (Sect. 3).
2. We generalize the Locality-Sensitive Filters (LSF) framework of Becker–Ducas–Gama–Laarhoven [BDGL16], and apply the results to tuple lattice

[3] All our results are also applicable when we solve the closest vector problem (CVP) via sieving as was done in [Laa16]. Asymptotic complexities for CVP are the same as for SVP.

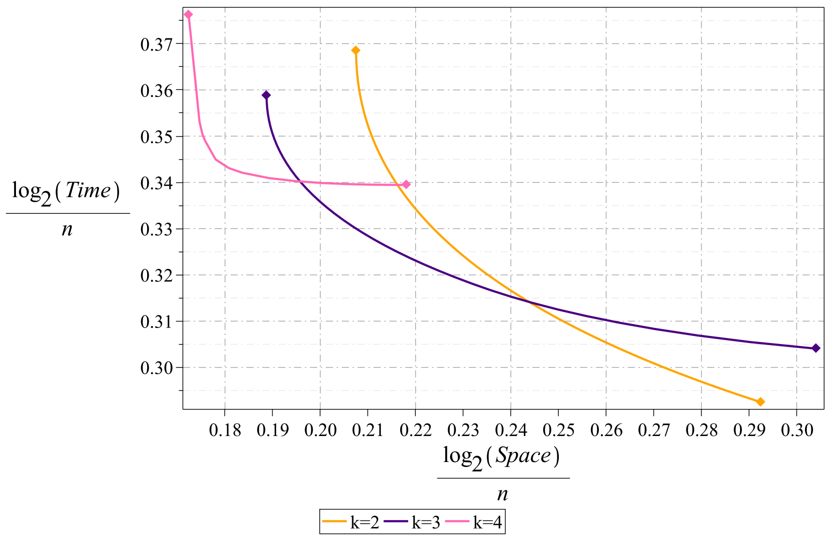


Fig. 1: Trade-offs for $k = 2, 3, 4$. Memory-exponents are on the X-axis, time-exponents are on the Y-axis. That means that for $x = m, y = t$, an algorithm will be of time-complexity $2^{t \cdot n + o(n)}$ and of memory-complexity $2^{m \cdot n + o(n)}$. Left-most points represent time and memory complexities for k-tuple sieving optimized for memory, right-most points represent complexities optimized for time.

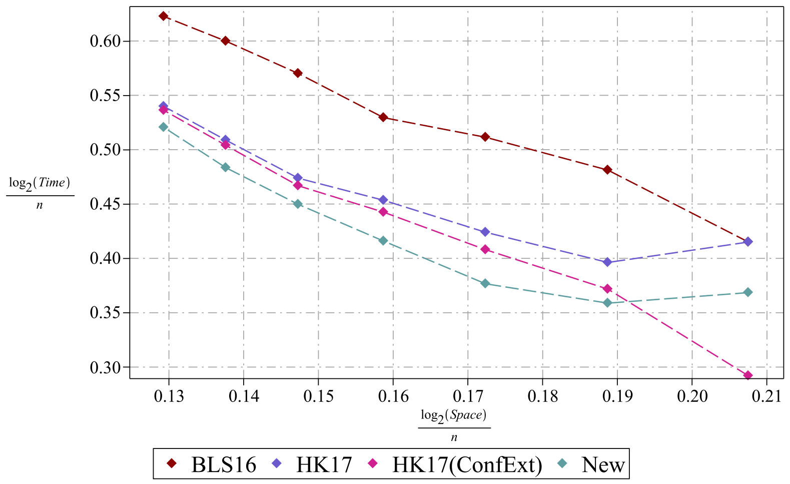


Fig. 2: Our runtime improvements for k-tuple lattice sieving over previous works for $3 \leq k \leq 8$ when we optimize for *memory*. For the $k = 2$ case, note that the results from [HK17] that use ConfExt with $T = 2^{0.292n+o(n)}$ cannot be applied to the Gauss-Sieve, but only to the NV-Sieve, which performs much worse in practice. Our results work for the Gauss-Sieve. See Sect. 4.2 for details.

sieving (Sect. 4). As an independent side result, we obtain explicit asymptotics for LSF for the approximate near neighbor problem on the sphere.[4]

3. We combine both techniques to obtain further improved asymptotic results compared to only using either technique (Sect. 5).

The remainder of the introduction is devoted to a high-level explanation of these techniques, and how they relate to previous work. We first introduce the approximate k-list problem. It serves as a useful abstraction of the tuple lattice sieving problem for which our results are *provable* – the results only become heuristic when applying them to tuple lattice sieving.

1.2 Approximate k-list problem

The approximate k-list problem is the central computational problem studied in this paper. We denote by $\mathsf{S}^{n-1} \subset \mathbb{R}^n$ the $(n-1)$-dimensional unit sphere. We use soft-$\widetilde{\mathcal{O}}$ notation, e.g. $\widetilde{\mathcal{O}}(2^n)$ means that we suppress sub-exponential factors.

Definition 1 (Approximate k-list problem). *Given k lists of i.i.d. uniformly random vectors $L_1, \ldots, L_k \subset \mathsf{S}^{n-1}$ and a target norm $t \in \mathbb{R}$, we are asked to find k-tuples $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \in L_1 \times \cdots \times L_k$ such that $\|\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_n\| \leq t$.*

We do not necessarily require to output all the solutions.

This problem captures the main subroutine of lattice sieving algorithms and allows us to describe precise, *provable* statements without any heuristic assumptions: all our results for the approximate k-list problem in the remainder of this paper are *unconditional.* In these application to lattice sieving, the lists will be identical (i.e., $|L_1| = \ldots = |L_k|$) and the number of such k-tuples required to be output will be $\widetilde{\mathcal{O}}(|L_1|)$.

To translate our results about the approximate k-list problem to lattice sieving, one needs to make additional heuristic assumptions, such as that the lists of lattice points appearing in sieving algorithms can be thought of as i.i.d. uniform vectors on a sphere (or a thin spherical shell). This essentially means that we do not 'see' the discrete structure of the lattice when we zoom out far enough, i.e. when the list contains very long lattice vectors. When the vectors in the list become short, we inevitably start noticing this discrete structure, and this heuristic assumption becomes invalid. Although experimental evidence suggests that these heuristic assumptions quite accurately capture the behavior of lattice sieving algorithms on random lattices, the results in the context of lattice sieving can only be proven under these additional (unproven) assumptions.

Under these heuristic assumptions, any algorithm for the approximate k-list problem with $t < 1$ and $|L_1| = \ldots = |L_k| = |L_{\text{out}}|$ will give an algorithm for SVP with the same complexity (up to polynomial factors). We dedicate Sect. 6 to such a SVP algorithm.

[4] The main difference with the works [ALRW17,Chr17], published after a preliminary version of some of these results [Laa15b], is that those papers focused on the case of list sizes scaling subexponentially in the dimension. Due to the application to lattice sieving, here we exclusively focus on exponential list sizes.

1.3 Generalized configuration search

By a concentration result on the distribution of scalar products of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathsf{S}^{n-1}$ previously shown in [HK17], the approximate k-list problem (Def. 1) can be reduced to the following configuration problem:

Definition 2 (Configuration problem). *Given k lists of i.i.d. uniform vectors $L_1, \ldots, L_k \subset \mathsf{S}^{n-1}$ and a target configuration given by $C_{i,j}$'s, find a $1-o(1)$-fraction of k-tuples $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \in L_1 \times \cdots \times L_k$ s.t. all pairs $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ in a tuple satisfy given inner-product constraints: $\langle \boldsymbol{x}_i \,, \boldsymbol{x}_j \rangle \approx C_{i,j}$.*

We consider this problem only for k and $C_{i,j}$'s fixed. The approximation sign $\approx$ in Def. 2 above is shorthand for $|\langle \boldsymbol{x}_i \,, \boldsymbol{x}_j \rangle - C_{i,j}| \leq \varepsilon$ for some small $\varepsilon > 0$, as we deal with real values. This approximation will affect our asymptotical results as $\widetilde{\mathcal{O}}(2^{cn+\nu(\varepsilon)n})$ for some $\nu(\varepsilon)$ that tends to 0 as we let $\varepsilon \to 0$. Eventually, $\nu(\varepsilon)$ will be hidden in the $\widetilde{\mathcal{O}}$-notation and we usually omit it.

We arrange these constraints into a $k \times k$ real matrix C – the Gram matrix of the $\boldsymbol{x}_i$'s – which we call a *configuration*. The connection to the k-list problem becomes immediate once we notice that a k-tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ with $\langle \boldsymbol{x}_i \,, \boldsymbol{x}_j \rangle \approx C_{i,j}, \forall i, j$, produces a sum vector whose norm satisfies $\|\sum_i \boldsymbol{x}_i\|^2 = \sum_{i,j} C_{i,j}$.

Consequently, solving the configuration problem for an appropriate configuration C with $\sum_{i,j} C_{i,j} \leq t^2$ will output a list of solutions to the Approximate k-list problem. The result [HK17, Theorem 1] shows that this will in fact return almost all solutions for a certain choice of $C_{i,j}$, i.e., k-tuples that form short sums are concentrated around one specific set of inner product constraints $C_{i,j}$.

The main advantage of the configuration problem is that it puts *pair-wise* constraints on solutions, which significantly speeds up the search. Moreover, C determines the expected number of solutions via a simple but very useful fact: the probability that a uniform i.i.d. tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ satisfies C is for fixed C, k given by (up to poly(n) factors) $\det(C)^{n/2}$ [HK17, Theorem 2]. Hence, if our goal is to output $|L|$ tuples, given $|L|^k$ possible k-tuples from the input, we must have $|L|^k \cdot \det(C)^{n/2} = |L|$. Now there are two ways to manipulate this equation. We can either fix the configuration C and obtain a bound on $|L|$ (this is precisely what [HK17] does), or vary C and deduce the required list-sizes for a given C. The latter option has the advantage that certain choices of C may result in a faster search for tuples. In this work, we investigate the second approach and present the trade-offs obtained by varying C.

Let us first detail on trade-offs obtained by varying the target configuration C. In [HK17], lattice sieving was optimized for memory with the optimum attained at to the so-called *balanced* configuration C (a configuration is called balanced if $C_{i,j} = -1/k, \forall i \neq j$). Such a configuration maximizes $\det(C)$, which in turn maximizes the number of k tuples that satisfy C. Hence, the balanced configuration minimizes the size of input lists (remember, in lattice-sieving we require the number of returned tuples be asymptotically equal to the input list size) and, hence, gives one of two extreme points of the trade-off.

Now assume we change the target configuration C. As the result, the number of returned tuples will be exponentially smaller than in the balanced case (as the value for $\det(C)$ decreases, the probability that a random k-tuple satisfies C decays by a factor exponential in n). To maintain the requirement on the size of the output, we need to increase the input lists. However, the *search* for tuples that satisfy C becomes faster for some choices of C. A choice for C with the fastest search gives another extreme point of the trade-off. In Sect. 3, we analyze the algorithm for the configuration problem and explain why certain C's result in faster algorithms. For small k, we give explicit time–memory trade-off curves.

1.4 Generalized locality-sensitive filters

Our second contribution improves the running time for the configuration search using a near neighbor method called spherical locality-sensitive filtering (LSF), first introduced in the context of lattice sieving in [BDGL16]. This method was later shown to be optimal for other applications in [ALRW17,Chr17].

LSF is an algorithm which receives on input a (typically large) set of points, a so-called query point usually not from the set, and a target distance d. It returns all points from the set that are within target distance d from the query point (the metric can be any, in our case, it will be angular). The aim of LSF is to answer many such queries fast by cleverly preprocessing this set so that the time of preprocessing is amortized among many query points. Depending on the choice of preprocessing, LSF may actually require more memory than the size of the input set. This expensive preprocessing results in faster query complexity, and the whole algorithm can be optimized (either for time or for memory) when we know how many query points we have.

In the application to tuple lattice sieving, we make use of the locality-sensitive filtering technique of [BDGL16] to speed up the configuration search routine. Time–memory trade-offs offered by LSF naturally translate to time–memory trade-offs for configuration search and, hence, for sieving.

There are several ways we can make use of LSF. First, we can apply LSF to the balanced configuration search and remain in the minimal memory regime (i.e., the memory bound for the LSF data structure is upper-bounded by the input list sizes). Interestingly, even in such a restricted regime we can speed up the configuration search and, in turn, asymptotically improve lattice sieving. Secondly, we allow LSF to use more memory while keeping the target configuration balanced. This has the potential to speed up the query cost leading to a faster configuration search. We can indeed improve k-tuple sieving in this regime for $k = 2, 3, 4$. In Sect. 4 we give exact figures in both aforementioned LSF scenarios.

1.5 Combining both techniques

Finally, we can combine LSF with changing the target configuration C. We search for an optimal C taking into account that we exploit LSF as a subroutine in the search for tuples satisfying C. Essentially, if we write out all the requirements on input/output list sizes and running time formulas, the search for such optimal C becomes a high-dimensional optimization problem (the number of variables to optimize for is of order k^2). Here, 'optimal' C may either mean the configuration that minimizes time, or memory, or time given a bound on memory. When we optimize C for time, we obtain exponents given in Table 1. Interestingly, for $k = 2, 3$, k-tuple sieve achieves the 'time=memory' regime.

Tuple size (k)	2	3	4	5	6
Time	0.2925	0.3041	0.3395	0.3459	0.4064
Space	0.2925	0.3041	0.2181	0.2553	0.2435

Table 1: Asymptotic complexities when using **LSF** and **arbitrary configurations**, when **optimizing for time**. I.e., we put no restriction on memory, but using more memory than stated in the table does not lead to a faster algorithm.

To obtain trade-off curves for $k = 2, 3, 4$, we set-up an optimization problem for several memory bounds and find the corresponding solutions for time. The curves presented in Fig. 1 are the best curve-fit for the obtained (memory, time) points. The right-most diamond-shaped points on the curves represent k-tuple algorithms when optimized for time (Table 1). The left-most are the points when we use the smallest possible amount of memory for this tuple size (see Table 3 in Sect. 4 for exact numbers). The curves do not extend further to the left since the k-tuple search will not succeed in finding tuples if the starting lists will be below a certain bound.

Already for $k = 4$, the optimization problem contains 27 variables and non-linear inequalities, so we did not attempt to give full trade-off curves for higher k's. We explain the constraints for such an optimization problem later in Sect. 5 and provide access to the Maple program we used. Extreme points for larger k's are given in Table 1 and Table 3 in Sect. 4.

1.6 Open problems

Although this work combines and optimizes two of the most prominent techniques for tuple lattice sieving, some open questions for future work remain, which we state below:

- As explained in [Laa16], sieving (with LSF) can also be used to solve CVPP (the closest vector problem *with preprocessing*) with time–memory trade-offs depending on the size of the preprocessed list and the LSF parameters. Tuple sieving would provide additional trade-off parameters in terms of the tuple size and the configuration to search for, potentially leading to better asymptotic time–memory trade-offs for CVPP.
- Similar to [LMvdP15], it should be possible to obtain asymptotic quantum speed-ups for sieving using Grover's algorithm. Working out these quantum trade-offs (and potentially finding other non-trivial applications of quantum algorithms to tuple sieving) is left for future work.

– An important open problem remains to study what happens when k is superconstant in n. Unfortunately, our analysis cruicially relies on k being fixed, as various subexponential terms depend on k.

2 Preliminaries

We denote vectors by bold letters, e.g., $\boldsymbol{x}$. In this paper we consider the ℓ_2-norm and denote the length of a vector $\boldsymbol{x}$ by $\|\boldsymbol{x}\|$. We denote by $\mathsf{S}^{n-1} \subset \mathbb{R}^n$ the $(n-1)$-dimensional unit sphere. For any square matrix $C \in \mathbb{R}^{k\times k}$ and $I \subset \{1, \ldots, k\}$, we denote by $C[I]$ the $|I| \times |I|$ submatrix of C obtained by restricting to the rows and columns indexed by I.

Lattices. Given a basis $B = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d\} \subset \mathbb{R}^n$ of linearly independent vectors, the lattice generated by B, denoted $\mathcal{L}(B)$, is given by $\mathcal{L}(B) \coloneqq \{\sum_{i=1}^{d} \lambda_i \boldsymbol{b}_i : \lambda_i \in \mathbb{Z}\}$. For simplicity of exposition, we assume $d = n$, i.e. the lattices considered are full rank. One of the central computational problems in the theory of lattices is the shortest vector problem (SVP): given a basis B, find a shortest non-zero vector in $\mathcal{L}(B)$. The length of this vector, known as the first successive minimum, is denoted $\lambda_1(\mathcal{L}(B))$.

The fastest (heuristic) algorithms for solving SVP in high dimensions are based on lattice sieving, originally described in [AKS01] and later improved in e.g. [NV08,PS09,MV10,Laa15a,LdW15,BL16,BDGL16]. These algorithms start by sampling an exponentially large list L of $2^{\ell n}$ (long) lattice vectors. The points from L are then iteratively combined to form shorter and shorter lattice points as $\boldsymbol{x}_{\text{new}} = \boldsymbol{x}_1 \pm \boldsymbol{x}_2 \pm \cdots \pm \boldsymbol{x}_k$ for some k.[5] The complexity is determined by the cost to find k-tuples whose combination produces shorter vectors.

In order to improve and analyze the cost of sieving algorithms, we consider the following problem, adapted from [HK17], generalizing Def. 1.

Definition 3 (Approximate k-list problem[HK17]). *Given k lists $L_1, \ldots, L_k$ of respective exponential sizes $2^{\ell_1 n}, \ldots, 2^{\ell_k n}$ whose elements are i.i.d. uniformly random vectors from S^{n-1}, the approximate k-list problem consists of finding $2^{\ell_{\text{out}} n}$ k-tuples $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \in L_1 \times \cdots \times L_k$ that satisfy $\|\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_k\| \le t$, using at most $M = \widetilde{\mathcal{O}}(2^{mn})$ memory.*

We are interested in the asymptotic complexity for $n \to \infty$ with all other parameters fixed. Note that the number of solutions to the problem is concentrated around its expected number and since we only want to solve the problem with high probability, we will work with expected list sizes throughout. See Appendix A in the full version [HKL] for a justification. We only consider cases where $m \ge \ell_{\text{out}}$ and where the expected number of solutions is at least $\widetilde{\Omega}(2^{\ell_{\text{out}} n})$. Part of our improvements over [HK17] comes from the fact that we consider the case where the total number of solutions to the approximate k-list problem is

[5] For the approximate k-list problem, we stick to all $+$ signs. This limitation is for analysis only, does not affect asymptotics and is easy to solve in practice.

exponentially larger than $2^{\ell_{\text{out}}n}$. By only requiring to find an exponentially small fraction of solutions, we can focus on solutions that are easier to find.

In the applications to sieving, we care about the case where $\ell_1 = \ldots = \ell_k = \ell_{\text{out}}$ and $t = 1$. Allowing different ℓ_i is mostly done for notational consistency with Def. 5 below, where different ℓ_i's naturally appear as subproblem in a recursive analysis of our algorithm.

2.1 Configurations and concentration results

One of the main technical tools introduced by [HK17] is so-called configurations. We repeat their definitions and results here, adapted to our case.

Definition 4 (Configuration [HK17, Definition 2]). *The configuration $C = \operatorname{Conf}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ of k points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ from the n-sphere is defined to be the Gram matrix of the $\boldsymbol{x}_i$, i.e. $C_{i,j} \coloneqq \langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle$.*

The connection between the approximate k-list problem and configurations is as follows: the configuration of a k-tuple $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ determines the length $\|\sum_i \boldsymbol{x}_i\|$:

$$\Big\|\sum_i \boldsymbol{x}_i\Big\|^2 = \sum_{i,j} C_{i,j} = \mathbf{1}^{\mathrm{t}} C \mathbf{1} \ , \tag{1}$$

where $\mathbf{1}$ is a column vector of 1's. So a k-tuple is a solution to the approximate k-list problem if and only if their configuration satisfies $\mathbf{1}^{\mathrm{t}} C \mathbf{1} \leq t^2$. Let

$$\begin{aligned} \mathscr{C} &= \{C \in \mathbb{R}^{k\times k} \mid C \text{ symmetric positive semi-definite, } C_{i,i} = 1\} \\ \mathscr{C}_{\text{good}} &= \{C \in \mathscr{C} \mid \mathbf{1}^{\mathrm{t}} C \mathbf{1} \leq t^2\} \end{aligned}$$

be the set of all possible configuration resp. of the configurations of solutions. We call the latter *good* configurations.

Following [HK17], rather than only looking for k-tuples that satisfy $\|\sum_i \boldsymbol{x}_i\| \leq t$, we look for k-tuples that additionally satisfy a constraint on their configuration as in the following problem (generalizing Def. 2)

Definition 5 (Configuration problem[HK17]). *Let $k \in \mathbb{N}$, let $m > 0$, let $\varepsilon > 0$, and suppose we are given a target configuration $C \in \mathscr{C}$. Given k lists $L_1, \ldots, L_k$ of respective exponential sizes $2^{\ell_1 n}, \ldots, 2^{\ell_k n}$ whose elements are i.i.d. uniform from S^{n-1}, the k-list configuration problem asks to find a $1-o(1)$ fraction of all solutions using at most $M = \widetilde{\mathcal{O}}(2^{mn})$ memory, where a solution is a k-tuple $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ with $\boldsymbol{x}_i \in L_i$ such that $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle - C_{i,j}| \leq \varepsilon$ for all i, j.*

Clearly, solving the configuration problem for any good configuration C yields solutions to the approximate k-list problem. If the number of solutions to the configuration problem is large enough, this is then sufficient to solve the approximate k-list problem. The number of expected solutions for a given configuration C can be easily determined from the distribution of $\operatorname{Conf}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ for i.i.d. uniform $\boldsymbol{x}_i \in \mathsf{S}^{n-1}$. For this we have

Theorem 1 (Distribution of configurations[HK17]). *Let $\boldsymbol{x}_1, \dots, \boldsymbol{x}_k$ be independent uniformly distributed from S^{n-1} with $n > k$. Then their configuration $C = \mathrm{Conf}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k)$ follows a distribution with density function*

$$\mu = W_{n,k} \cdot \det(C)^{\frac{1}{2}(n-k)} \mathrm{d}C_{1,2} \dots \mathrm{d}C_{n-1,n} \ , \tag{2}$$

where $W_{n,k} = \mathcal{O}_k(n^{\frac{1}{4}(k^2-k)})$ is an (explicitly known) normalization constant that only depends on n and k.

This implies that we need to solve the configuration problem for any good configuration such that $\prod_i |L_i| \cdot (\det C)^{n/2} = \widetilde{\Omega}(2^{\ell_{\mathrm{out}} n})$.

In [HK17], the authors ask to find *essentially all* solutions to the approximate k-list problem. For this, they solve the configuration problem for a particular target configuration $\overline{C}$, such that the solutions to the configuration problem for $\overline{C}$ comprise a $1-o(1)$-fraction of the solutions to the approximate k-list problem. $\overline{C}$ has the property that all non-diagonal entries are equal; such configurations are called *balanced*. In the case $t = 1$, we have $\overline{C}_{i,j} = -\frac{1}{k}$ for $i \neq j$.

By contrast, we are fine with only finding *enough* solutions to the approximate k-list problem. This relaxation allows us to consider other, non-balanced, configurations.

2.2 Transformation

In our application, we will have to deal with lists L whose elements are not uniform from S^{n-1}. Instead, the elements $\boldsymbol{x} \in L$ have prescribed scalar products $\langle \boldsymbol{v}_i \, , \boldsymbol{x} \rangle$ with some points $\boldsymbol{v}_1, \dots, \boldsymbol{v}_r$.

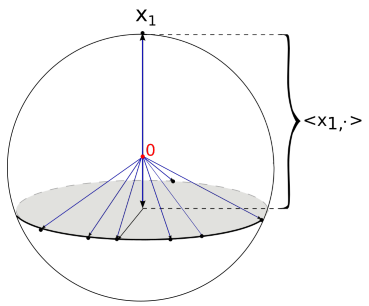


Fig. 3: Taking a vector $\boldsymbol{x}_1 \in L_1$ and applying filtering to $L_2, \dots, L_k \subset \mathsf{S}^{n-1}$ w.r.t. $\boldsymbol{x}_1$ fixes the distances between $\boldsymbol{x}_1$ and all the vectors from $L_2, \dots, L_k$ that 'survive' the filtering. All filtered vectors (i.e., vectors from $L_2^{(1)}, \dots, L_k^{(1)}$) can be considered as vectors from S^{n-2} – a scaled sphere of one dimension less.

The effect of these restriction is that the elements $\boldsymbol{x} \in L$ are from some (shifted, scaled) sphere of reduced dimension S^{n-1-r}, cf. Fig. 3. We can apply a transformation (Transform in Alg. 1 below), eliminating the shift and rescaling. This reduces the situation back to the uniform case, allowing us to use recursion. Note that we have to adjust the target scalar products to account for the shift and scalings. A formal statement with formulas how to adjust the scalar products is given by Lemma 3 in Appendix B in the full version [HKL].

3 Generalized configuration search

Now we present our algorithm for the Configuration problem (Def. 5). We depict the algorithm in Fig. 4. Its recursive version is given in Alg. 1.

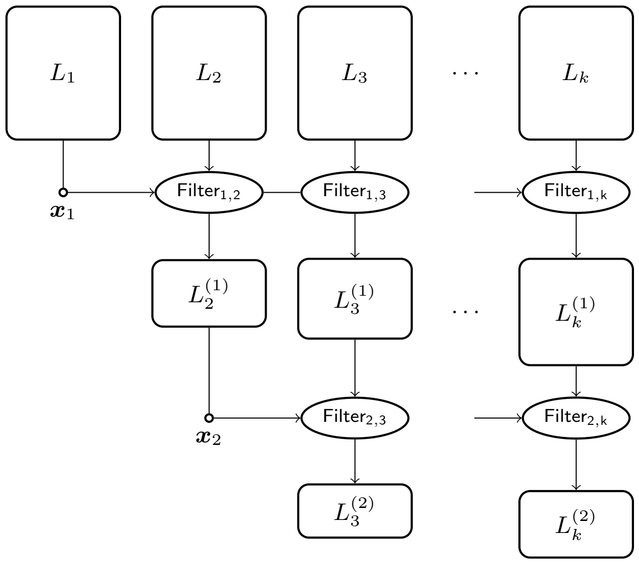


Fig. 4: Our algorithm for the Configuration problem. Procedures $\mathsf{Filter}_{i,j}$ receive on input a vector (e.g. $\boldsymbol{x}_1$), a list of vectors (e.g. L_2), and a real number $C_{i,j}$ - the target inner product. It creates another shorter list (e.g. $L_2^{(1)}$) that contains all vectors from the input list whose inner product with the input vector is within some small ε from the target inner product. Time–memory trade-offs are achieved by taking different values $C_{i,j}$'s for different i, j. In particular, an asymptotically faster k-list algorithm can be obtained by more 'aggressive' filtering to the left-most lists on each level. In other words, for fixed i, the value $C_{i,j}$ is the largest (in absolute value) for $j = i + 1$.

The algorithm receives on input k lists $L_1, \dots, L_k$ of exponential (and potentially different) sizes and the target configuration $C \in \mathbb{R}^{k \times k}$. It outputs a list L_{out} of tuples $\boldsymbol{x}_1, \dots, \boldsymbol{x}_k \in L_1 \times \cdots \times L_k$ s.t. $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle - C_{i,j}| \leq \varepsilon$ for all i, j.

The algorithm processes the lists from left to right (see Fig. 4). Namely, for each $\boldsymbol{x}_1 \in L_1$, it creates lists $L_2^{(1)}, \dots, L_k^{(1)}$ by applying a filtering procedure

Algorithm 1 Recursive algorithm for the configuration problem

Input: $L_1, \dots, L_k$− lists of vectors from S^{n-1}, $C_{i,j} = \langle \boldsymbol{x}_i , \boldsymbol{x}_j \rangle \in \mathbb{R}^{k\times k}$− Gram matrix, $\varepsilon > 0$ - fudge-factor.

Output: L_{out}− list of k-tuples $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k) \in L_1 \times \dots \times L_k$, s.t. $|\langle \boldsymbol{x}_i , \boldsymbol{x}_j \rangle - C_{ij}| \le \varepsilon$ for all i, j.

```
 1: function SolveConfigurationProblem($L_1, \dots, L_k, \varepsilon, C_{i,j} \in \mathbb{R}^{k\times k}$)
 2:     $L_{\text{out}} \leftarrow \emptyset$
 3:     if $k = 1$ then
 4:         $L_{\text{out}} \leftarrow L_1$
 5:     else
 6:         for all $\boldsymbol{x}_1 \in L_1$ do
 7:             for all $j = 2 \dots k$ do
 8:                 $L_j^{\mathsf{f}} \leftarrow$ Filter($\boldsymbol{x}_1, L_j, C_{1,j}, \varepsilon$)        ▷ Create all filtered lists
 9:                 $L'_j, C' \leftarrow$ Transform($\boldsymbol{x}_1, L_j^{\mathsf{f}}, C, i$)        ▷ Remove the dependency on $\boldsymbol{x}_1$
10:             $\hat{L} \leftarrow$ SolveConfigurationProblem($L'_2, \dots L'_k, \varepsilon, C_{[2..k],[2..k]}$)
11:             for all $\boldsymbol{x} \in \hat{L}$ do
12:                 $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\boldsymbol{x}_1, \boldsymbol{x})\}$        ▷ Append $\boldsymbol{x}$ to all tuples that contain $\boldsymbol{x}_1$
13:     return $L_{\text{out}}$
```

```
 1: function Filter($\boldsymbol{v}, L, c, \varepsilon$)
 2:     $L' \leftarrow \emptyset$
 3:     for all $\boldsymbol{x} \in L$ do
 4:         if $|\langle \boldsymbol{x} , \boldsymbol{v} \rangle - c| \le \varepsilon$ then
 5:             $L' \leftarrow L' \cup \{\boldsymbol{x}\}$
 6:     return $L'$
```

```
 1: function Transform($\boldsymbol{v}, L, C, i$)        ▷ Changes the $i^{\text{th}}$ row of $C$
 2:     $L' \leftarrow \emptyset$
 3:     Transform all $C_{i,j}$ to $C'_{j,k}$ for $j, k > i$ as follows:
```

$$C'_{j,k} = \frac{1}{\sqrt{(1 - C^2_{i-1,j})(1 - C^2_{i-1,k})}} (C_{j,k} - C_{i-1,j} \cdot C_{i-1,k})$$

```
    (see Lemma 3 in Appendix B of the full version for justification).
 4:     for all $\boldsymbol{x} \in L$ do
 5:         $\boldsymbol{x}^{\perp} \leftarrow \boldsymbol{x} - \langle \boldsymbol{x} , \boldsymbol{v} \rangle \boldsymbol{v}$
 6:         $L' \leftarrow L' \cup \left\{ \frac{\boldsymbol{x}^{\perp}}{\|\boldsymbol{x}^{\perp}\|} \right\}$
 7:     return $L', C'$
```

w.r.t. $\boldsymbol{x}_1$ to these $k-1$ lists $L_2, \dots, L_k$. This filtering takes as input a vector, a list L_j, and target inner-product $C_{1,j}$ for all $j \geq 2$. Its output $L_j^{(1)}$ contains all the vectors $\boldsymbol{x}_j$ from L_j that satisfy $|\langle \boldsymbol{x}_1 , \boldsymbol{x}_j \rangle - C_{1,j}| \leq \varepsilon$ for some small fixed $\varepsilon > 0$. Note the upper index of $L_j^{(1)}$: it denotes the number of filterings applied to the original list L_j.

Applying filtering to all $k-1$ lists, we obtain $k-1$ new lists that contain vectors with a fixed angular distance to $\boldsymbol{x}_1$ (see Fig. 3). Now our task is to find all the $k-1$ tuples $(\boldsymbol{x}_2, \dots, \boldsymbol{x}_k) \in L_2^{(1)}, \dots, L_k^{(1)}$ that satisfy $C[2 \dots k]$. Note that this is almost an instance of the $(k-1)$ configuration problem, except for the distribution of the filtered elements. To circumvent this issue we apply to all elements of filtered lists the transformation described in Lemma 2 (Appendix B, full version [HKL]), where our fixed $\boldsymbol{x}_1$ plays the role of $\boldsymbol{v}$.

It is easy to see that the time needed for the transformation is equal to the size of filtered lists and thus, asymptotically irrelevant. Finally, we can apply the algorithm recursively to the $k-1$ transformed lists.

Note that the transformation is merely a tool for analysis and the algorithm can easily be implemented without it. We only need it for the analysis of the LSF variant in Sect. 4.

3.1 Analysis

In this section we analyse the complexity of Alg. 1. Speed-ups obtained with Nearest Neighbor techniques are discussed in the next section. In this 'plain' regime, the memory complexity is completely determined by the input list sizes. Applying filtering can only decrease the list sizes. Recall that k is assumed to be a fixed constant. Further, as our algorithm is exponential in n and we are only interested in asymptotics, in our analysis we ignore polynomial factors, i.e. computations of inner-products, summations, etc.

Our main objective is to compute the (expected) size of lists on each level (by the term 'level' we mean the number of filterings applied to a certain list or, equivalently, the depth of the recursion). We are interested in $|L_i^{(j)}|$ - the size of list L_i after application of j filterings. For the input lists, we set $L_i^{(0)} := L_i$.

Once we know how the list-sizes depend on the chosen configuration C, it is almost straightforward to conclude on the running time. Consider the j^{th} recursion-level. On this level, we have j points $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_j)$ fixed during the previous recursive calls and the lists $L_{j+1}^{(j-1)}, \dots, L_k^{(j-1)}$ of (possibly) different sizes which we want to filter wrt. $\boldsymbol{x}_j$. Asymptotically, all filterings are done in time $\max_{j+1 \leq i \leq k} \big| L_i^{(j-1)} \big|$. This process will be recursively called as many times as many fixed tuples $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_j)$ we have, namely, $\prod_{r=1}^{j} \big| L_r^{(r-1)} \big|$ times (i.e., the product of all left-most list-sizes). Hence, the cost to create *all* lists on level j can be expressed as

$$T_j = \prod_{r=1}^{j} \big| L_r^{(r-1)} \big| \cdot \max_{j+1 \leq i \leq k} \Big\{ \big| L_i^{(j-1)} \big|, 1 \Big\}. \tag{3}$$

In the above formula, considering the maximum between a filtered list and 1 takes care about the lists of size 0 or 1 (i.e., the exponent for the list-sizes might be negative). Assume on a certain level all filtered lists contain only one or no elements in expectation. Technically, to create the next level, we still need to check if theses lists are indeed empty or not. This also means that the level above – the one that created these extremely short lists – takes more time than the subsequent level.

Finally, the total running time of the algorithm is determined by the level j for which T_j is maximal (we refer to such a level as dominant):

$$T = \max_{1 \le j \le k} \left[\prod_{r=1}^{j} \left| L_r^{(r-1)} \right| \cdot \max_{j+1 \le i \le k} \left| L_i^{(j-1)} \right| \right]. \tag{4}$$

Note that we do not need to take into account using lists of expected sizes ≤ 1. As mentioned above, creating these lists takes more time than using them.

In the following lemma, we use the results of Thm. 1 to determine $|L_i^{(j)}|$ for $0 \le j \le i-1, 1 \le i \le k$. We denote by $C[1 \dots j, i]$ the $(j+1) \times (j+1)$ the submatrix of C formed by restricting its columns and rows to the set of indices $\{1, \dots, j, i\}$.

Lemma 1. *During a run of Alg. 1 that receives on input a configuration $C \in \mathbb{R}^{k \times k}$ and lists $L_1, \dots, L_k$, the intermediate lists $L_i^{(j)}$ for $1 \le i \le k$, $i-1 \le j \le k$ are of expected sizes*

$$\mathbb{E}\left[\left| L_i^{(j)} \right| \right] = |L_i| \cdot \left(\frac{\det(C[1 \dots j, i])}{\det(C[1 \dots j])} \right)^{n/2}. \tag{5}$$

Proof. The list $L_i^{(j)}$ is created when we have the tuple $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_j)$ fixed. The probability for such a tuple to appear is $\det(C[1 \dots j])^{n/2}$. Moreover, the probability that we ever consider a tuple $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_j, \boldsymbol{x}_i)$ for any $\boldsymbol{x}_i \in L_i$ is given by $(\det(C[1 \dots j, i]))^{n/2}$. The result follows when we take the latter probability conditioned on the event that $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_j)$ is fixed.

Using the result of Thm. 1 once again, we obtain the expected number of the returned tuples, i.e. the size of L_{out}.

Corollary 1. *Alg. 1 receiving on input a configuration $C \in \mathbb{R}^{k \times k}$ and the lists $L_1, \dots, L_k$ of respective sizes $2^{\ell_1 n}, \dots, 2^{\ell_k n}$, outputs $|L_{\text{out}}|$ solutions to the configuration problem, where*

$$\mathbb{E}\left[|L_{\text{out}}| \right] = 2^{(\sum_i^k \ell_i) \cdot n} \cdot (\det C)^{n/2}. \tag{6}$$

In particular, if all k input lists are of the same size $|L|$ and the output list size is required to be of size $|L|$ as well (the case of sieving), we must have

$$|L| = (\det C)^{-\frac{n}{2(k-1)}}. \tag{7}$$

Proof. The statement follows immediately from the total number of k-tuples and the fact that the probability that a random k-tuple has the desired configuration is (up to polynomial factors) $(\det C)^{n/2}$ (see Thm. 1).

We remark that our Alg. 1 outputs all k-tuples that satisfy a given configuration C. This is helpful if one wants to solve the k-list problem using the output of Alg. 1. All one needs to do is to provide the algorithm with the lists $L_1, \ldots, L_k$ and a configuration C, such that according to Eq. (6), we expect to obtain $2^{\ell_{\mathrm{out}}}$ k-tuples. Furthermore, we require $\mathbf{1}^{\mathrm{t}} C \mathbf{1} \leq t^2$, so $C \in \mathscr{C}_{\mathrm{good}}$ and every solution to the configuration problem is a solution to the k-list problem.

Note also that the k-list problem puts a bound M on the memory used by the algorithm. This directly translates to the bound on the input lists for the configuration problem (recall that filtering only shrinks the lists).

The above discussion combined with Lemma 1 yields the next theorem, which states the complexity of our algorithm for the k-list problem.

Theorem 2. *Alg. 1 is expected to output $|L_{\mathrm{out}}| = 2^{\ell_{\mathrm{out}}}$ solutions to the k-list problem in time T provided that the input $L_1, \ldots, L_k$ and $C \in \mathscr{C}_{\mathrm{good}}$ satisfy Eq. (6) and $\max_i |L_i| \leq M$, where T is (up to polynomial factors) equal to*

$$\max_{1 \leq j \leq k} \left[\prod_{r=1}^{j} |L_r| \left(\frac{\det C[1 \ldots r]}{\det C[1 \ldots r-1]} \right)^{\frac{n}{2}} \cdot \max_{j+1 \leq i \leq k} |L_i| \left(\frac{\det(C[1 \ldots j-1, i])}{\det(C[1 \ldots j-1])} \right)^{\frac{n}{2}} \right] \quad (8)$$

Note that we miss exponentially many solutions to the *k-list problem*, yet for the *configuration* search, we expect to obtain all the tuples that satisfy the given target configuration C. This loss can be compensated by increased sizes of input lists. Since the running time T (see Eq. (8)) depends on C and on the input list-sizes in a rather intricate manner, we do no simplify the formula for T, but rather discuss an interesting choice for input parameters in case $k = 3$.

3.2 Case of interest: $k = 3$

The case $k = 3$ is the most relevant one if we want to apply our 3-list algorithm to SVP sieving algorithms (see Sect. 6 for a discussion on the k-Gauss sieve algorithm for SVP). In [HK17], the k-sieve algorithm was proved to be the most time-efficient when $k = 3$ among all the non-LSF based algorithms. In particular, a 3-sieve was shown to be faster than non-LSF 2-sieve.

To be more concrete, the 3-list problem (the main subroutine of the 3-Gauss sieve) can be solved using the *balanced* configuration search (i.e., $C_{i,j} = -1/3$ for $i \neq j$) in time $T_{\mathrm{bal}} = 2^{0.396n}$ requiring memory $M = 2^{0.1887n}$. Up to poly(n) factors these numbers are also the complexities of the 3-Gauss sieve algorithm. The main question we ask is whether we can reduce the time T at the expense of the memory M?

If we take a closer look at the time complexity of the 3-Configuration search for the balanced C, we notice that among the two levels – on the first level we filter wrt. $\boldsymbol{x}_1 \in L_1$, on the second wrt. $\boldsymbol{x}_2 \in L_2^{(1)}$ – the second one dominates.

In other words, if we expand Eq. (8), we obtain $T = \max\{T_1 = 2^{0.377n}, T_2 = 2^{0.396n}\} = T_2$. We denote by T_i the time needed to create all lists on the i^{th} level (see Eq. (3)). Hence, there is potential to improve T by putting more work on the first level hoping to reduce the second one.

This is precisely what our optimization for T, M, and configuration C suggests: (1) increase the input list sizes from $2^{0.1887n}$ to $2^{0.1895n}$, and (2) use more 'aggressive' filtering on the first level to balance out the two levels. We spend more time creating $L_2^{(1)}$ and $L_3^{(1)}$ as the input lists became larger, but the filtered $L_2^{(1)}, L_3^{(1)}$ are shorter and contain more 'promising' pairs. With the configuration C given below, the time complexity of the 3-Configuration search becomes $T = \max\{T_1 = T_2\} = 2^{0.3789n}$ with input and output list sizes equal to $2^{0.1895n}$. We remark that in our optimization we put the constraint that $|L_{\text{out}}| = |L_1|$.

$$C = \begin{pmatrix} 1 & -0.3508 & -0.3508 \\ -0.3508 & 1 & -0.2984 \\ -0.3508 & -0.2984 & 1 \end{pmatrix}.$$

In fact, for $k = 3, \ldots, 8$, when we optimize for time (cf. Table 2), the same phenomenon occurs: the time spent on each level is the same. We conjecture that such a feature of the optimum continues for larger values of k, but we did not attempt to prove it.

3.3 Trade–off curves

Now we demonstrate time-memory trade-offs for larger values of k. In Fig. 5 we plot the trade-off curves obtained by changing of the target configuration C. For each $3 \leq k \leq 7$, there are two extreme cases on each curve: the left-most points describe the algorithm that uses balanced configuration. Here, the memory is as low as possible. The other endpoint gives the best possible time complexity achievable by a change of the target configuration (the right-most points on the plot). Adding more memory will not improve the running time. Table 2 gives explicit values for the time-optimized points.

Tuple size (k)	2	3	4	5	6	7	8
Time	0.4150	0.3789	0.3702	0.3707	0.3716	0.3722	0.3725
Space	0.2075	0.1895	0.1851	0.1853	0.1858	0.1861	0.1862

Table 2: Asymptotic complexities for **arbitrary configurations** when **optimizing for time**. These are the right-most points for each k on the time–memory trade-off curve from Fig. 5.

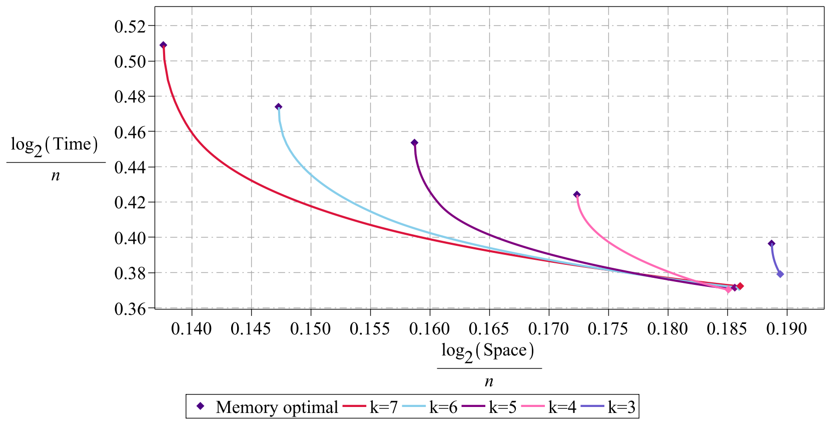


Fig. 5: Time–memory trade-offs for $k = 3, \dots, 7$ and for **arbitrary configurations**. Similar to Fig. 1, the complexities are on a logarithmic scale. The left-most points give complexities for the configuration search in case of balanced configurations. The right-most point indicates the lowest time. Increasing memory even further will not result in improved running times.

4 Generalized locality-sensitive filters

In our Alg. 1, a central sub-problem that arises is that of efficient filtering: given c, $\boldsymbol{x}_1$ and a list L_j, find all $\boldsymbol{x}_j \in L_j$ such that $\langle \boldsymbol{x}_j , \boldsymbol{x}_1 \rangle \approx c$, where c is determined by the target configuration. Note that, by using Lemma 3 (Appendix B, full version), we may assume that the points in L_j are uniform from some sphere of dimension $n - o(n)$. To simplify notation, we ignore the $o(n)$-term, since it will not affect the asymptotics. For notational reasons[6], we shall assume $c \geq 0$; the case $c \leq 0$ is equivalent by changing $\boldsymbol{x}_1$ to $-\boldsymbol{x}_1$.

So far, we solved the problem of finding all such $\boldsymbol{x}_j$ in time $|L_j|$. However, better algorithms exist if we preprocess L_j. This leads to the following problem:

Definition 6 (Near neighbor on the sphere). *Let L consist of N points drawn uniformly at random from S^{n-1}, and let $c \geq 0$. The c-near neighbor problem is to preprocess L such that, given a query vector $\boldsymbol{q} \in \mathsf{S}^{n-1}$, one can quickly find points $\boldsymbol{p} \in L$ with $\langle \boldsymbol{p} , \boldsymbol{q} \rangle \geq c$.*

Depending on the magnitude of N there is a distinction between the near neighbor problem for *sparse* data sets ($N = 2^{o(n)}$) and for *dense* data sets ($N = 2^{\Theta(n)}$). In many applications of near neighbor searching one is interested in sparse data sets, and various lower bounds matching upper bounds have been derived for this regime [OWZ14,AIL+15,ALRW17,Chr17]. For our purposes, we are only interested in *dense* data sets.

[6] The literature is concerned with finding points that are close to each other (corresponding to $c > 0$) rather than points that are far apart (corresponding to $c < 0$). Note that in our applications, we typically would obtain $c < 0$.

The target scalar product c corresponds to a target angular distance $\phi = \arccos c$ and we want to find points in a spherical cap centered around $\boldsymbol{q}$ with angular distance ϕ. With $c \geq 0$, we have $\phi < \frac{1}{2}\pi$, so we are really concerned with near neighbors. In order to simplify incorporating our results for LSF into the configurations framework, we analyze and express everything in terms of scalar products rather than angular distances. This allows us to use Eq. (2) to express the involved complexities.

Let us denote the data structure resulting from the preprocessing phase by $\mathcal{D}$. In algorithms for the near neighbor problem, we are interested in time and memory complexities of both preprocessing and of querying and also in the size of $\mathcal{D}$. Spending more resources on preprocessing and using more space will generally reduce the time complexity of querying.

In our applications, we may want to modify the list L by adding/removing vectors in between queries, without having to completely rebuild all of $\mathcal{D}$. So we assume that $\mathcal{D}$ is built up by processing the $\boldsymbol{x} \in L$ one element at a time and updating $\mathcal{D}$. We therefore analyze the preprocessing cost in terms of the cost to update $\mathcal{D}$ when L changes by one element.

Spherical locality-sensitive filters. To solve the near neighbor problem on the sphere, [BDGL16] introduced spherical locality-sensitive filters, inspired by e.g. the spherical cap LSH of [AINR14]. The idea is to create a data structure $\mathcal{D}$ of many *filter buckets*, where each bucket $B_{\boldsymbol{u}}$ contains all vectors from L which are α-close to a filter vector $\boldsymbol{u}$, where $\boldsymbol{u}$ is typically not from L, but drawn uniformly at random from S^{n-1}. Here, two vectors $\boldsymbol{u}, \boldsymbol{p}$ are considered α-close iff $\langle \boldsymbol{u}, \boldsymbol{p} \rangle \geq \alpha$. Let F be the set of all such chosen $\boldsymbol{u}$'s. Ideally, one generates $|F| \gg 1$ of these buckets, each with $\boldsymbol{u} \in F$ chosen independently and uniformly at random from S^{n-1}. We build up the data structure by processing the elements $\boldsymbol{x} \in L$ one by one and updating $\mathcal{D}$ each time, as mentioned above. This means that for each update, we need to find all $\boldsymbol{u} \in F$ that are α-close to a given $\boldsymbol{x}$.

If we then want to find points $\boldsymbol{x} \in L$ that are c-close to a given query point $\boldsymbol{q}$, we first find all $\boldsymbol{u} \in F$ that are β-close to $\boldsymbol{x}$ for some β. Then we search among those buckets $B_{\boldsymbol{u}}$ for points $\boldsymbol{x} \in B_{\boldsymbol{u}}$ that are c-close to $\boldsymbol{x}$. The idea here is that points in $B_{\boldsymbol{u}}$ for $\boldsymbol{u}$ close to $\boldsymbol{q}$ have a higher chance to be close to $\boldsymbol{q}$ than a random point from L. The algorithm is detailed in Alg. 2.

Structured filters. In the above idea, one has to find all $\boldsymbol{u} \in F$ that are close to a given point $\boldsymbol{x}$. A naive implementation of this would incur a cost of $|F|$, which would lead to an impractically large overhead. To surmount this problem, a small amount of *structure* is added to the filter vectors $\boldsymbol{u}$, making them dependent: small enough so that their joint distribution is sufficiently close to $|F|$ independent random vectors, but large enough to ensure that finding the filter vectors that are close to $\boldsymbol{x}$ can be done in time (up to lower-order terms) proportional to the number of such close filters vectors. This is the best one can hope for. This technique was later called "tensoring" in [Chr17], and replaced with a tree-based data structure in [ALRW17]. For further details regarding this technique we refer the reader to [BDGL16]; below, we will simply assume that

Algorithm 2 Algorithm for spherical locality-sensitive filtering

```
Parameters: $\alpha, \beta, c, F$
$\mathcal{D}$ and $L$ are kept as global state, modified by functions below. $\mathcal{D} = \{B_{\boldsymbol{u}} \mid \boldsymbol{u} \in F\}$.
INSERT and REMOVE modify $L$ and $\mathcal{D}$. We assume that $\mathcal{D}$ and $L$ are constructed
by calling INSERT repeatedly. Initially, all $B_{\boldsymbol{u}}$ and $L$ are empty.
Query($\boldsymbol{q}$) outputs $\boldsymbol{x} \in L$ that are $c$-close to $\boldsymbol{q}$.

1: function INSERT($\boldsymbol{x}$)                          ▷ Add $\boldsymbol{x}$ to $L$ and update $\mathcal{D}$
2:     $L \leftarrow L \cup \{\boldsymbol{x}\}$
3:     for all $\boldsymbol{u} \in F$ s.t. $\langle \boldsymbol{u}, \boldsymbol{x} \rangle \geq \alpha$ do
4:         $B_{\boldsymbol{u}} \leftarrow B_{\boldsymbol{u}} \cup \{\boldsymbol{x}\}$

1: function REMOVE($\boldsymbol{x}$)                          ▷ Remove $\boldsymbol{x}$ from $L$ and update $\mathcal{D}$
2:     $L \leftarrow L \setminus \{\boldsymbol{x}\}$
3:     for all $\boldsymbol{u} \in F$ s.t. $\langle \boldsymbol{u}, \boldsymbol{x} \rangle \geq \alpha$ do
4:         $B_{\boldsymbol{u}} \leftarrow B_{\boldsymbol{u}} \setminus \{\boldsymbol{x}\}$

1: function QUERY($\boldsymbol{q}$)                           ▷ Find $\boldsymbol{x} \in L$ with $\langle \boldsymbol{x}, \boldsymbol{q} \rangle \geq c$
2:     PointsFound $\leftarrow \emptyset$
3:     for all $\boldsymbol{u} \in F$ s.t. $\langle \boldsymbol{u}, \boldsymbol{q} \rangle \geq \beta$ do
4:         for all $\boldsymbol{x} \in B_{\boldsymbol{u}}$ do
5:             if $\langle \boldsymbol{x}, \boldsymbol{q} \rangle \geq c$ then
6:                 PointsFound $\leftarrow$ PointsFound $\cup \{\boldsymbol{x}\}$
7:     return PointsFound
```

filter vectors are essentially independent, and that we can solve the problem of finding all $\boldsymbol{u} \in F$ close to given point with a time complexity given by the number of solutions.

Complexity. Let us now analyze the complexity of our spherical locality-sensitive filters and set the parameters α and β. For this we have the following theorem:

Theorem 3 (Near neighbor trade-offs). *Consider Alg. 2 for fixed target scalar product $0 \leq c \leq 1$, fixed $0 \leq \alpha, \beta \leq 1$ and let L be a list of i.i.d. uniform points from S^{n-1}, with $|L|$ exponential in n. Then any pareto-optimal*[7] *parameters satisfy the following restrictions:*

$$\begin{aligned} |L|\,(1-\alpha^2)^{n/2} &= 1 \\ \alpha c \leq \beta &\leq \min\left\{\frac{\alpha}{c}, \frac{c}{\alpha}\right\} \end{aligned} \tag{9}$$

Assume these restrictions hold and that $|F|$ is chosen as small as possible while still being large enough to guarantee that on a given query, we find all c-close points except with superexponentially small error probability. Then the complexity of spherical locality-sensitive filtering is, up to subexponential factors, given by:

[7] This means that we cannot modify the parameters α, β in a way that would reduce either the preprocessing or the query cost without making the other cost larger.

- ***Bucket size**: The expected size of each bucket is* 1.
- ***Number of buckets**: The number of filter buckets is*

$$|F| = \frac{(1-c^2)^{n/2}}{(1+2c\alpha\beta - c^2 - \alpha^2 - \beta^2)^{n/2}}.$$

- ***Update time**: For each $\boldsymbol{x} \in L$, updating the data structure $\mathcal{D}$ costs time* $T_{\text{Update}} = |F| \cdot (1-\alpha^2)^{n/2}$
- ***Preprocessing time**: The total time to build $\mathcal{D}$ is $T_{\text{Preprocess}} = |F|$.*
- ***Memory used**: The total memory required is $|F|$, used to store $\mathcal{D}$.*
- ***Query time**: After $\mathcal{D}$ has been constructed, each query takes time* $T_{\text{Query}} = |F| \cdot (1-\beta^2)^{n/2}$.

Note that the formulas for the complexity in the theorem rely on Eqns. (9) to hold. As the proof of this theorem (mainly the conditions for pareto-optimality) is very technical, we defer it to Appendix D (see full version [HKL]). In the following Rmk. 1, we only discuss the meaning of the restrictions that are satisfied in the pareto-optimal case and sketch how to derive the formulas. After that, we discuss what the extreme cases for β mean in terms of time/memory trade-offs.

Remark 1. Using Thm. 1, for a given point $\boldsymbol{u}$ and uniformly random $\boldsymbol{x} \in \mathsf{S}^{n-1}$, the probability that $\langle \boldsymbol{u}, \boldsymbol{x} \rangle \approx \alpha$ is given by $(1-\alpha^2)^{n/2}$, ignoring subexponential factors throughout the discussion. So the expected size of the filter buckets is $|L|\,(1-\alpha^2)^{n/2}$ and the first condition ensures that this size is actually 1.

If the expected bucket size was exponentially small, we would get a lot of empty buckets. Since F needs to have a structure that allows to find all $\boldsymbol{u} \in F$ close to a given point quickly, we cannot easily remove those empty buckets. Consequently, the per-query cost would be dominated by looking at (useless) empty buckets. It is strictly better (in both time and memory) to use fewer, but more full buckets in that case. Conversely, if the buckets are exponentially large, we should rather use more, but smaller buckets, making $\mathcal{D}$ more fine-grained.

Now consider a "solution triple" $(\boldsymbol{x}, \boldsymbol{q}, \boldsymbol{u})$, where $\boldsymbol{q}$ is a query point, $\boldsymbol{x} \in L$ is a solution for this query and $\boldsymbol{u} \in F$ is the center of the filter bucket used to find the solution. By definition, this means $\langle \boldsymbol{x}, \boldsymbol{q} \rangle \geq c$, $\langle \boldsymbol{x}, \boldsymbol{u} \rangle \geq \alpha$ and $\langle \boldsymbol{q}, \boldsymbol{u} \rangle \geq \beta$. The second set of conditions from Eq. (9) imply that these 3 inequalities are actually satisified with (near-)equality whp. Geometrically, it means that the angular triangle formed by a triple $\boldsymbol{q}, \boldsymbol{x}, \boldsymbol{u}$ in a 2-dimensional S^2 with these pairwise inner products has no obtuse angles.

The required size of $|F|$ is determined by the conditional probability $P = \frac{1}{|F|}$ that a triple $(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{q})$ has these pairwise scalar products, conditioned on $\langle \boldsymbol{x}, \boldsymbol{q} \rangle \approx c$. Using Thm. 1, this evaluates to

$$P = \Pr_{\boldsymbol{u} \in \mathsf{S}^{n-1}}[\langle \boldsymbol{x}, \boldsymbol{u} \rangle \approx \alpha, \langle \boldsymbol{q}, \boldsymbol{u} \rangle \approx \beta \mid \langle \boldsymbol{x}, \boldsymbol{q} \rangle \approx c] = \frac{\left(\det\left(\begin{smallmatrix} 1 & \alpha & \beta \\ \alpha & 1 & c \\ \beta & c & 1 \end{smallmatrix}\right)\right)^{n/2}}{\left(\det\left(\begin{smallmatrix} 1 & c \\ c & 1 \end{smallmatrix}\right)\right)^{n/2}}.$$

Taking the inverse gives $|F|$. The other formulas are obtained even more easily by applying Thm. 1: the probability that a point $x \in L$ is to be included in a bucket

$\boldsymbol{B_u}$ is $(1-\alpha^2)^{n/2}$, giving the update cost (using the structure of F). Similarly for the per-query cost, we look at $|F|\,(1-\beta^2)^{n/2}$ buckets with $|L|\,(1-\alpha^2)^{n/2}$ elements each. Using $|L|\,(1-\alpha^2)=1$ then gives all the formulas.

Note that Theorem 3 offers some trade-off. With $|L|$ and c given, the parameter α is determined by the condition $(1-\alpha^2)^{n/2}\,|L|=1$. The complexities can then be expressed via β, which we can chose between $c\cdot\alpha$ and either α/c or c/α.

For $\beta=c\cdot\alpha$, we have (up to subexponential terms) $|F|=\frac{1}{(1-\alpha^2)^{n/2}}=|L|$, so the update cost is subexponential. The memory complexity is only $\widetilde{\mathcal{O}}(|L|)$, which is clearly minimal. The query complexity is at $\widetilde{\mathcal{O}}(|L|\,(1-c^2\alpha^2)^{n/2})$. Increasing β increases both the time complexity for updates and the memory complexity for $\mathcal{D}$, whereas the query complexity decreases. If $\alpha\le c$, we can increase β up to $\beta=\frac{\alpha}{c}$. At that point, we have subexponential expected query complexity. The total complexity of preprocessing is given by $\frac{1}{(1-\alpha^2/c^2)^{n/2}}$.

If $\alpha\ge c$, we may increase β up to $\frac{c}{\alpha}$. At that point, we obtain a query complexity of $|L|\,(1-c^2)^{n/2}$. This is equal to the number of expected solutions to a query, hence the query complexity is optimal. The preprocessing cost is equal to $|L|\,(1-c^2)^{n/2}(1-\alpha^2/c^2)^{-n/2}$. Increasing β further will only make both the query and preprocessing costs worse. Note that, if the total number of queries is less than $\frac{1}{(1-\beta_{\max}^2)^{n/2}}$ with $\beta_{\max}=\min\{\frac{\alpha}{c},\frac{c}{\alpha}\}$, it does not make sense to increase β up to $\beta_{\max}$ even if we have arbitrary memory, as the total time of preprocessing will exceed the total time of all queries. In the special case where the number of queries $|Q|$ is equal to the list size $|L|$, preprocessing and total query costs are equal for $\alpha=\beta$. This latter choice corresponds to the case used in [BDGL16].

In the LSH literature, the quality of a locality sensitive hash function is usually measured in terms of update and query exponents ρ_u resp. ρ_q. Re-phrasing our results in these quantities, we obtain the following corollary:

Corollary 2. *Let $c>0$, corresponding to an angular distance $\phi=\arccos c$, $0<\phi<\frac{1}{2}\pi$ and consider the c-near neighbor problem on the n-dimensional unit sphere for dense data sets of size $N=2^{\Theta(n)}$. Then, for any value of γ with $c\le\gamma\le\min\{\frac{1}{c},\frac{c}{1-N^{-2/n}}\}$, spherical locality sensitive filtering solves this problem with update and query exponents given by:*

$$\rho_u=\log\Big(\frac{\sin^2\phi}{\sin^2\phi-(1-N^{-2/n})(1-2\gamma\cos\phi+\gamma^2)}\Big)/\log(N^{2/n})-1$$

$$\rho_q=\log\Big(\frac{(1-\gamma^2+\gamma^2N^{-2/n})\sin^2\phi}{\sin^2\phi-(1-N^{-2/n})(1-2\gamma\cos\phi+\gamma^2)}\Big)/\log(N^{2/n})$$

The data structure requires $N^{1+\rho_u+o(1)}$ memory, can be initialized in time $N^{1+\rho_u+o(1)}$ and allows for updates in time $N^{\rho_u+o(1)}$. Queries can be answered in time $N^{\rho_q+o(1)}$.

Proof. This is just a restatement of Thm. 3 with $\gamma:=\beta/\alpha$, plugging in $\alpha^2=1-N^{-2/n}$ and $\cos\phi=c$.

4.1 Application to the configuration problem

We now use spherical locality-sensitive filtering as a subroutine of Alg. 1, separately replacing each naive $\textsc{Filter}_{i,j}$ subroutine of Alg. 1 resp. Fig. 4 by spherical locality-sensitive filtering. In this application, in each near neighbor problem the number of queries is equal to the list size. We caution that in Alg 1, the lists for which we have to solve a c-near neighbor problem were obtained from the initial lists by restricting scalar products with known points. This changes the distribution, which renders our results from Thm. 3 not directly applicable. In order to obtain a uniform distribution on a sphere, we use TRANSFORM in Alg. 1, justified by Lemma 3 (Appendix B in the full version [HKL]), to perform an affine transformation on all our point. This transformation affects scalar products and, as a consequence, the parameter c in each near neighbor problem is not directly given by $\pm C_{i,j}$.

For the case of the k-sieve with the balanced configuration $C_{i,j} = -\frac{1}{k}$ for $i \neq j$, the parameter c used in the replacement of $\textsc{Filter}_{i,j}$ is given by $\frac{1}{k+1-i}$, as a simple induction using Lemma 3 (full version) shows.

Using LSF changes the time complexities as follows: recall the time complexity is determined by the cost to create various sublists $L_j^{(i)}$ as in Sect. 3.1, where $L_j^{(i)}$ is obtained from the input list L_j by applying i filterings. These filterings and $L_j^{(i)}$ depend on the partial solution $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$. The cost $T_{i,j}$ to create all instances (over all choices of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$) of $L_j^{(i)}$ is then given by (cf. Eq. (3))

$$T_{i,j} = \prod_{r=1}^{i-1} \left|L_r^{(r-1)}\right| \cdot \left(T_{\text{Preprocess},i,j} + \left|L_i^{(i-1)}\right| \cdot T_{\text{Query},i,j}\right),$$

where $T_{\text{Preprocess},i,j}$ and $T_{\text{Query},i,j}$ denote the time complexities for LSF when replacing $\textsc{Filter}_{i,j}$. Here, $\boldsymbol{x}_i \in L_i^{(i-1)}$ takes the role of the query point.

Applying LSF to this configuration gives time/memory trade-offs depicted in Fig. 6. Optimizing for time yields Table 4 and for memory yields Table 3. Note that for $k \geq 5$, there is no real trade-off. The reason for this is as follows: for large k and balanced configuration, the running time is dominated by the creation of sublists $L_j^{(i)}$ with large i. At these levels, the list sizes $L_j^{(i)}$ have already shrunk considerably compared to the input list sizes. The memory required even for time-optimal LSF at these levels will be below the size of the input lists. Consequently, increasing the memory complexity will not help. We emphasize that for any $k > 2$, the memory-optimized algorithm has the same memory complexity as [HK17], but strictly better time complexity.

4.2 Comparison with Configuration Extension

There exists a more memory-efficient variant [BGJ14] of the LSF techniques than described above, which can reach optimal time without increasing the memory complexity. Roughly speaking, the idea is to immediately process the filter buckets after creating them and never storing them all. However, even ignoring

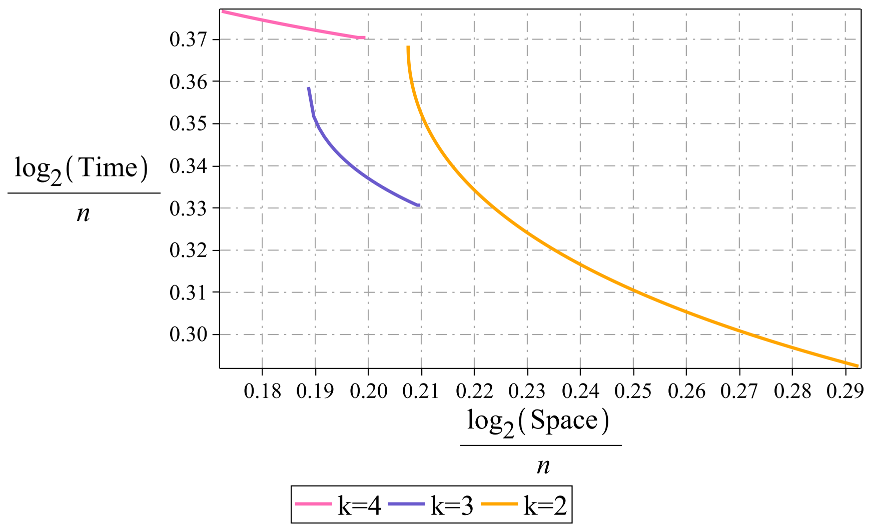


Fig. 6: Trade-offs for $k = 2, 3, 4$. Analogous to Figure 1, the axes are on a logarithmic scale. The exact values for the left-most points for each curve (optimized for memory) are given in Table 3. The right-most points (optimized for time) are given in Table 4.

subexponential overhead, this variant has two limitations: Firstly, we need to know all query points $\boldsymbol{q} \in Q$ in advance. This makes it unsuitable for the top level (which is the only level for $k = 2$) in the Gauss Sieve, because we build up the list L vector by vector. Secondly, in this variant we obtain the solutions $(\boldsymbol{q}, \boldsymbol{x})$ with $\boldsymbol{q} \in Q, \langle \boldsymbol{q}, \boldsymbol{x} \rangle \approx c$ in essentially random order. In particular, to obtain all solutions $\boldsymbol{x}$ for *one* given query point $\boldsymbol{q}$, there is some overhead[8]. Note that, due to the structure of Alg. 1, we really need the solutions for one query point after another (except possibly at the last level). So the memory-less variant does not seem well-suited for our algorithm for $k > 2$.

A generalization of memory-efficient LSF to $k > 2$ was described in [HK17] under the name configuration extension and applied to sieving. However, due to the second limitation described above, they achieve smaller speed-ups than we do for $k > 2$.

5 Combining both techniques

In the previous sections we showed how to obtain time–memory trade-offs by either (1) changing the target configuration C (Sect. 3), or (2) using locality-sensitive filters for the balanced configuration (Sect. 4). An obvious next step

[8] E.g. we could first output the solutions for *all* query points and sort these solutions wrt. the query point, but that requires storing the set of all solutions for all queries and increases the memory complexity.

Tuple size (k)	2	3	4	5	6	7	8
Time ([HK17])	0.415	0.3962	0.4240	0.4534	0.4738	0.5088	0.5398
Time (with LSF)	0.3685	0.3588	0.3766	0.4159	0.4497	0.4834	0.5205
Space	0.2075	0.1887	0.1723	0.1587	0.1473	0.1376	0.1293

Table 3: Asymptotic complexities when using **LSF** with the **balanced/any configuration**, when **optimizing for memory**: we keep the memory equal to the input list sizes. The running time for $k = 2$ with LSF is equal to the one obtained in [BDGL16]. Without LSF, the runtime exponent 0.4150 for $k = 2$ was first shown in [NV08]. Note that, when optimizing for memory, we implicitly restrict to the balanced configuration, because this minimizes the input list size.

Tuple size (k)	2	3	4	5	6	7	8
Time (with LSF)	0.3685	0.3307	0.3707	0.4159	0.4497	0.4834	0.5205
Space	0.2075	0.2092	0.1980	0.1587	0.1473	0.1376	0.1293

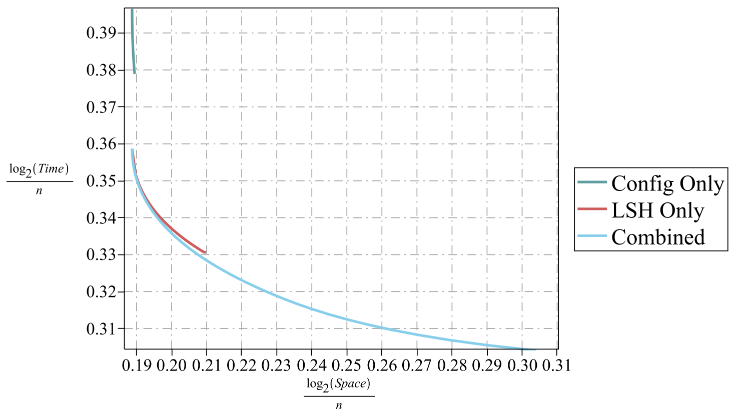
Table 4: Asymptotic complexities when using **LSF** with the **balanced configuration**, when **optimizing for time** (i.e. using the largest amount of memory that still leads to better time complexities). With this we can obtain improved time complexities for our k-list algorithm for $k = 2, 3, 4$. Starting from $k = 5$, giving more memory to LSF without changing the target configuration does not result in an asymptotically faster algorithm.

would be to try to combine both techniques to obtain even better results, i.e. by solving the configuration problem for an *arbitrary* target configuration C using LSF as a subroutine. The memory complexity will be either dominated by the input list sizes (which are determined by C from the condition that the output list size equals the input list sizes) or by the filter buckets used for LSF.

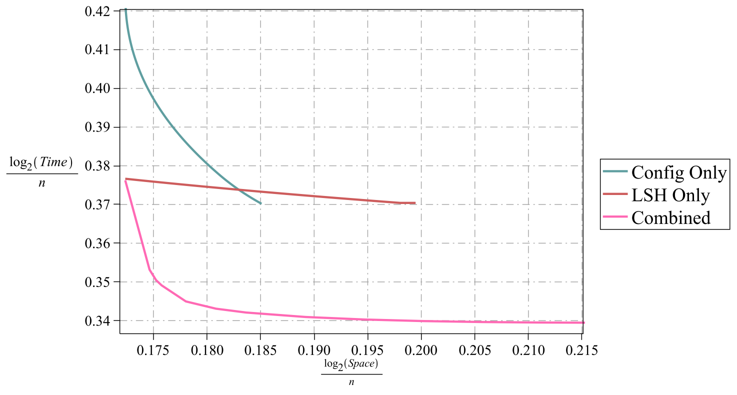
To obtain time–space trade-offs for sieving, we optimize over all possible configurations C with $\sum_{i,j} C_{i,j} \leq 1$ and parameters $\beta_{i,j}$ used in each application of LSF. We used MAPLE [M] for the optimization. To obtain the complete trade-off curve, we optimized for time while prescribing a bound on memory, and varying this memory bound to obtain points on the curve. Note that the memory-optimized points are the same as in Sect. 4.1, since we have to keep C balanced – in a memory-restricted regime, LSF only gives the same improvement as combining LSF with arbitrary target configurations. However, as soon as the memory bound is slightly larger, we already observe that this combination of both techniques leads to results strictly better than ones produced by using neither or only one of the two techniques.

The resulting trade-off curves are depicted in Fig. 1 for $k = 2, 3, 4$. The same curves for $k = 3, 4$ are also depicted in Fig. 7, where the trade-offs are compared to using either a modified configuration (without LSF) or LSF (with a balanced configuration). The time-optimal points for fixed k are given in Table 1.

In general, for small k we can gain a lot by using LSF, whereas for large k, most of the improvement comes from changing the target configuration. Note also that the trade-offs obtained by combining both techniques are strictly superior to using only either improvement.



(a) $k = 3$



(b) $k = 4$

Fig. 7: Trade-offs for $k = 3, 4$ with different improvements: changing the target configuration, LSF, or both.

6 Tuple Gauss sieve

So far we have been concerned with the algorithm for the configuration problem and how it can be used to solve the approximate k-list problem. Here we explain how our algorithm for the k-list problem can be used as a subroutine within k-tuple lattice sieving. We only give a high-level overview here. A more detailed description, including pseudo-code can be found in Appendix C in [HKL].

Lattice sieving algorithms have two flavors: the Nguyen-Vidick sieve [NV08] and the Gauss sieve [MV10]. Since in practice the latter outperforms the former, we concentrate on Gauss sieve. However, our approach can be used for both.

The algorithm receives on input a lattice represented by a basis $B \in \mathbb{R}^{n\times n}$ and outputs a vector $\boldsymbol{v} \in \mathcal{L}(B)$ s.t. $\|\boldsymbol{v}\| = \lambda_1(\mathcal{L}(B))$. During its run, the Gauss Sieve needs to efficiently sample (typically long) points $\boldsymbol{v} \in \mathcal{L}(B)$, whose direction is nearly uniformly distributed.

After preprocessing the basis by an L^3 reduction, we can use Klein's sampling procedure [Kle00], which outputs vectors of length $2^n \cdot \lambda_1(\mathcal{L}(B))$ in poly(n) time.

In the Gauss Sieve, we keep a set S of vectors and try to perform as many k-reductions on S as possible. By this, we mean that we look for k-tuples $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \in S$ with $0 < \|\boldsymbol{x}_1 \pm \cdots \pm \boldsymbol{x}_k\| < \max\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\}$ and we replace the vector of maximal length by the (shorter) sum. To find such k-tuples, we use Alg. 1 for the configuration problem for an appropriately chosen configuration C. If no k-tuples from S satisfy C, we enlarge S by sampling a new vector.

To avoid checking whether the same k-tuples satisfies C multiple times, we separate $S = L \cup Q$ into a list L and a queue Q. The list L contains the lattice vectors that we already checked: we maintain that no k-tuples from L satisfy C. The queue Q contains vectors that might still be part of a k-tuple satisfying C.

Due to our splitting of S, we may assume that one of the vectors in the k-tuples is from Q. In fact, we can just repeatedly call Alg. 1 on lists $L_1 = \{\boldsymbol{p}\}$, $L_2 = \ldots = L_k = L$ for $\boldsymbol{p} \in Q$ and perform k-reductions on the solutions.

Whenever we sample or modify a vector, we have to move it into Q; if no more reduction is possible with $L_1 = \{\boldsymbol{p}\}$, $L_2 = \ldots = L_k$, we move $\boldsymbol{p}$ from Q into L. If Q is empty, this signals that we have to sample a new vector.

Since the length of the vectors in S keeps decreasing, we hope to eventually find the shortest vector. We stop the search when the length of the shortest element of the list (i.e., the shortest lattice vector found) is equal to the first successive minimum, $\lambda_1(B)$. Since we usually do not know the value of $\lambda_1(B)$ exactly, we use some heuristics: in practice [MLB15], we stop once we found a lot of k-tuples where the k-reduction would give a zero vector.

6.1 Gauss Sieve with $k = 3$ in practice

We ran experiments with the 3-Gauss sieve algorithm with the aim to compare balanced and non-balanced configuration search. We used a Gauss-sieve implementation developed by Bai, Laarhoven, and Stehlé in [BLS16] and by Herold-Kirshanova in [HK17].

As an example, we generated 5 random (in the sense of Goldstein-Mayer [GM06]) 70-dimensional lattices and preprocessed them with β-BKZ reduction for $\beta = 10$. Our conclusions are the following:

– The unbalanced configuration C given above indeed puts more work to the (asymptotically) non-dominant first level than the balanced configuration

n	$\lvert C_{i,j}\rvert = 0.32$			$\lvert C_{i,j}\rvert = 0.333$			$\lvert C_{i,j}\rvert = 0.3508$		
	level 1	level 2	T/sec	level 1	level 2	T/sec	level 1	level 2	T/sec
60	$4.8 \cdot 10^{8}$	$1.7 \cdot 10^{8}$	$5.6 \cdot 10^{2}$	$5.8 \cdot 10^{8}$	$1.2 \cdot 10^{8}$	$6.0 \cdot 10^{2}$	$7.2 \cdot 10^{8}$	$2.1 \cdot 10^{8}$	$7.3 \cdot 10^{2}$
62	$9.4 \cdot 10^{8}$	$3.5 \cdot 10^{8}$	$1.3 \cdot 10^{3}$	$1.1 \cdot 10^{9}$	$6.7 \cdot 10^{8}$	$2.7 \cdot 10^{3}$	$1.4 \cdot 10^{9}$	$1.4 \cdot 10^{8}$	$3.4 \cdot 10^{3}$
64	$1.7 \cdot 10^{9}$	$6.8 \cdot 10^{8}$	$3.4 \cdot 10^{3}$	$2.1 \cdot 10^{9}$	$4.6 \cdot 10^{8}$	$5.0 \cdot 10^{3}$	$2.7 \cdot 10^{9}$	$2.7 \cdot 10^{8}$	$5.6 \cdot 10^{3}$
66	$3.0 \cdot 10^{9}$	$1.5 \cdot 10^{9}$	$5.2 \cdot 10^{3}$	$3.9 \cdot 10^{9}$	$8.7 \cdot 10^{8}$	$2.1 \cdot 10^{4}$	$5.1 \cdot 10^{9}$	$5.1 \cdot 10^{8}$	$1.5 \cdot 10^{4}$
68	$6.0 \cdot 10^{9}$	$2.5 \cdot 10^{9}$	$1.4 \cdot 10^{4}$	$7.3 \cdot 10^{9}$	$1.6 \cdot 10^{9}$	$1.1 \cdot 10^{4}$	$9.6 \cdot 10^{9}$	$9.5 \cdot 10^{8}$	$2.7 \cdot 10^{4}$
70	$1.1 \cdot 10^{10}$	$4.9 \cdot 10^{9}$	$3.0 \cdot 10^{4}$	$1.4 \cdot 10^{10}$	$3.2 \cdot 10^{9}$	$3.6 \cdot 10^{4}$	$1.8 \cdot 10^{10}$	$1.8 \cdot 10^{9}$	$4.9 \cdot 10^{4}$
72	$2.1 \cdot 10^{10}$	$9.4 \cdot 10^{9}$	$4.8 \cdot 10^{4}$	$2.6 \cdot 10^{10}$	$6.1 \cdot 10^{9}$	$7.2 \cdot 10^{4}$	–	–	–
74	$3.9 \cdot 10^{10}$	$1.8 \cdot 10^{10}$	$1.3 \cdot 10^{5}$	$4.7 \cdot 10^{10}$	$1.1 \cdot 10^{10}$	$1.4 \cdot 10^{5}$	$6.4 \cdot 10^{10}$	$6.2 \cdot 10^{9}$	$1.9 \cdot 10^{5}$
76	$7.0 \cdot 10^{10}$	$3.4 \cdot 10^{10}$	$2.7 \cdot 10^{5}$	$8.6 \cdot 10^{10}$	$2.1 \cdot 10^{10}$	$3.1 \cdot 10^{5}$	–	–	–

Table 5: Running times of triple lattice sieve for 3 different target configurations $C_{i,j}$. All the figures are average values over 5 different Goldstein-Mayer lattices of dimension n. Columns 'level 1' show the number of inner-products computed on the first level of the algorithm (it corresponds to the quantity $|L|^2$), columns 'level 2' count the number of inner-product computed in the second level (in corresponds to $|L| \cdot |L^{(1)}|^2$). The third columns T shows actual running times in seconds.

does. We counted the number of inner products (applications of filterings) on the first and on the seconds levels for each of the 5 lattices. As the algorithm spends most of its time computing inner-products, we think this model reasonably reflects the running time.

The average number of inner-product computations performed on the upper-level increases from $1.39 \cdot 10^{10}$ (balanced case) to $1.84 \cdot 10^{10}$ (unbalanced case), while on the lower (asymptotically dominant) level this number drops down from $3.23 \cdot 10^9$ to $1.82 \cdot 10^9$.

– As for the actual running times (when measured in seconds), 3-Gauss sieve instantiated with balanced configuration search outperforms the sieve with the unbalanced C given above on the dimensions up to 74. We explain this by the fact that the asymptotical behavior is simply not visible on such small dimensions. In fact, in our experiments the dominant level turns out to be the *upper* one (just compare the number of the inner-products computations in Table 5). So in practice one might want to reverse the balancing: put more more work on the lower level than on the upper by again, changing the target configuration. Indeed, if we relax the inner-product constraint to 0.32 (in absolute value), we gain a significant speed-up compared with balanced 1/3 and asymptotically optimal 0.3508.

Acknowledgements. We thank the anonymous reviewers of PKC 2018 for careful proof-reading and finding an error in one of our tables. Gottfried Herold and Elena Kirshanova are supported by ERC Starting Grant ERC-2013-StG-335086-LATTAC. Thijs Laarhoven is supported by ERC consolidator grant 617951.

References

[ADPS16] Erdem Alkim, Leo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *USENIX Security Symposium*, pages 327–343, 2016.

[ADRS15] Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in 2^n time via discrete Gaussian sampling. In *STOC*, pages 733–742, 2015.

[AEVZ02] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

[AIL$^+$15] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, pages 1225–1233, 2015.

[AINR14] Alexandr Andoni, Piotr Indyk, Huy Lê Nguyên, and Ilya Razenshteyn. Beyond locality-sensitive hashing. In *SODA*, pages 1018–1028, 2014.

[AKS01] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.

[ALRW17] Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *SODA*, pages 47–66, 2017.

[AN17] Yoshinori Aono and Phong Q. Nguyên. Random sampling revisited: lattice enumeration with discrete pruning. In *EUROCRYPT*, 2017.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, pages 10–24, 2016.

[BGJ14] Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. In *ANTS*, pages 49–70, 2014.

[BL16] Anja Becker and Thijs Laarhoven. Efficient (ideal) lattice sieving using cross-polytope LSH. In *AFRICACRYPT*, pages 3–23, 2016.

[BLS16] Shi Bai, Thijs Laarhoven, and Damien Stehlé. Tuple lattice sieving. In *ANTS*, pages 146–162, 2016.

[Chr17] Tobias Christiani. A framework for similarity search with space-time trade-offs using locality-sensitive filtering. In *SODA*, pages 31–46, 2017.

[CN11] Yuanmi Chen and Phong Q. Nguyên. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, pages 1–20, 2011.

[FP85] Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice. *Mathematics of Computation*, 44(170):463–471, 1985.

[GM06] Daniel Goldstein and Andrew Mayer. On the equidistribution of hecke points. *Forum Mathematicum*, 15(3):165–189, 01 2006.

[GNR10] Nicolas Gama, Phong Q. Nguyên, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

[HK17] Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in Euclidean norm. In *PKC*, 2017.

[HKL] Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. *eprint.iacr.org/2017/1228*.

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC*, pages 193–206, 1983.

[Kle00] Philip Klein. Finding the closest lattice vector when it's unusually close. In *SODA*, pages 937–941, 2000.

[Laa15a] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.

[Laa15b] Thijs Laarhoven. Tradeoffs for nearest neighbors on the sphere. *arXiv*, pages 1–16, 2015.

[Laa16] Thijs Laarhoven. Finding closest lattice vectors using approximate voronoi cells. *Cryptology ePrint Archive, Report 2016/888*, 2016.

[LdW15] Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In *LATINCRYPT*, pages 101–118, 2015.

[LMvdP15] Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2):375–400, 2015.

[LRBN] R. Lindner, M. Rückert, P. Baumann, and L. Nobach. SVP challenge generator. `http://latticechallenge.org/svp-challenge`.

[M] Standard Worksheet Interface, Maple 2016.0, February 17 2016, build id 1113130.

[MLB15] Artur Mariano, Thijs Laarhoven, and Christian Bischof. Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In *ICPP*, pages 590–599, 2015.

[MV10] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 2010, pages 1468–1480, 2010.

[NV08] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. In *Journal of Mathematical Cryptology*, volume 2, pages 181–207, 2008.

[OWZ14] Ryan O'Donnell, Yi Wu, and Yuan Zhou. Optimal lower bounds for locality-sensitive hashing (except when q is tiny). *ACM Transactions on Computation Theory*, 6(1):5:1–5:13, 2014.

[PS09] Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. *Cryptology ePrint Archive, Report 2009/605*, pages 1–7, 2009.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, pages 84–93, 2005.

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2–3):201–224, 1987.

[SE94] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(2–3):181–199, 1994.

[WLTB11] Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In *ASIACCS*, pages 1–9, 2011.

[ZPH13] Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. In *SAC*, pages 29–47, 2013.