Improved Algorithms for the Approximate k-List Problem in Euclidean Norm

Gottfried Herold, Elena Kirshanova

Faculty of Mathematics
Horst Görtz Institute for IT-Security
Ruhr University Bochum
{gottfried.herold, elena.kirshanova}@rub.de

Abstract. We present an algorithm for the approximate k-List problem for the Euclidean distance that improves upon the Bai-Laarhoven-Stehlé (BLS) algorithm from ANTS'16. The improvement stems from the observation that almost all the solutions to the approximate k-List problem form a particular configuration in n-dimensional space. Due to special properties of configurations, it is much easier to verify whether a k-tuple forms a configuration rather than checking whether it gives a solution to the k-List problem. Thus, phrasing the k-List problem as a problem of finding such configurations immediately gives a better algorithm. Furthermore, the search for configurations can be sped up using techniques from Locality-Sensitive Hashing (LSH). Stated in terms of configuration-search, our LSH-like algorithm offers a broader picture on previous LSH algorithms.
For the Shortest Vector Problem, our configuration-search algorithm results in an exponential improvement for memory-efficient sieving algorithms. For $k = 3$, it allows us to bring down the complexity of the BLS sieve algorithm on an n-dimensional lattice from $2^{0.4812n+o(n)}$ to $2^{0.3962n+o(n)}$ with the same space requirement $2^{0.1887n+o(n)}$. Note that our algorithm beats the Gauss Sieve algorithm with time resp. space of $2^{0.415n+o(n)}$ resp. $2^{0.208n+o(n)}$, while being easy to implement. Using LSH techniques, we can further reduce the time complexity down to $2^{0.3717n+o(n)}$ while retaining a memory complexity of $2^{0.1887n+o(n)}$.

1 Introduction

The k-List problem is defined as follows: given k lists $L_1, \ldots, L_k$ of elements from a set X, find k-tuples $(x_1, \ldots, x_k) \in L_1 \times \ldots \times L_k$ that satisfy some condition C. For example, Wagner [19] considers $X \subset \{0,1\}^n$, and a tuple $(x_1, \ldots, x_k)$ is a solution if $x_1 \oplus \ldots \oplus x_n = 0^n$. In this form, the problem has found numerous applications in cryptography [14] and learning theory [6].

For ℓ_2-norm conditions with $X \subset \mathbb{R}^n$ and $k = 2$, the task of finding pairs $(\boldsymbol{x}_1, \boldsymbol{x}_2) \in L_1 \times L_2$, s.t. $\|\boldsymbol{x}_1 + \boldsymbol{x}_2\| < \min\{\|\boldsymbol{x}_1\|, \|\boldsymbol{x}_2\|\}$, is at the heart of certain algorithms for the Shortest Vector Problem (SVP). Such algorithms, called *sieving* algorithms ([1], [17]), are asymptotically the fastest SVP solvers known so far.

Sieving algorithms look at pairs of lattice vectors that sum up to a short(er) vector. Once enough such sums are found, repeat the search by combining these shorter vectors into even shorter ones and so on. It is not difficult to see that in order to find even one pair where the sum is shorter than both the summands, we need an exponential number of lattice vectors, so the memory requirement is exponential. In practice, due to the large memory-requirement, sieving algorithms are outperformed by the asymptotically slower Kannan enumeration [10].

Naturally, the question arises whether one can reduce the constant in the exponent of the memory complexity of sieving algorithms at the expense of running time. An affirmative answer is obtained in the recently proposed k-list sieving by Bai, Laarhoven, and Stehlé [4] (BLS, for short). For constant k, they present an algorithm that, given input lists $L_1, \ldots, L_k$ of elements from the n-sphere S^n with radius 1, outputs k-tuples with the property $\|\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_n\| < 1$. They provide the running time and memory-complexities for $k = 3, 4$.

We improve and generalize upon the BLS k-list algorithm. Our results are as follows:

1. We present an algorithm that on input $L_1, \ldots, L_k \subset \mathsf{S}^n$, outputs k-tuples $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k), \in L_1 \times \ldots \times L_k$, s.t. all *pairs* $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ in a tuple satisfy certain inner product constraints. We call this problem the Configuration problem (Def. 3).
2. We give a concentration result on the distribution of scalar products of $\boldsymbol{x}_1, \ldots \boldsymbol{x}_k \in \mathsf{S}^n$ (Thms. 1 and 2), which implies that finding vectors that sum to a shorter vector can be reduced to the above Configuration problem.
3. By working out the properties of the aforementioned distribution, we *prove* the conjectured formula (Eq. (3.2) from [4]) on the input list-sizes (Thm. 3), s.t. we can expect a constant success probability for sieving. We provide closed formulas for the running times for both algorithms: BLS and our Alg. 1 (Thm. 4). Alg. 1 achieves an exponential speed-up compared the BLS algorithm.
4. To further reduce the running time of our algorithm, we introduce the so-called Configuration Extension Algorithm (Alg. 2). It has an effect similar to Locality-Sensitive Hashing as it shrinks the lists in a helpful way. This is a natural generalization of LSH to our framework of configurations. We briefly explain how to combine Alg. 1 and the Configuration Extension in Sect. 7. A complete description can be found in the full version.

Roadmap. Sect. 2 gives basic notations and states the problem we consider in this work. Sect. 3 introduces configurations – a novel tool that aids the analysis in succeeding Sect. 4 and 5 where we present our algorithm for the k-List problem and prove its running time. Our generalization of Locality Sensitive Hashing – Configuration Extension – is described in Sect. 6 and its application to the k-list problem in Sect. 7. We conclude with experimental results confirming our analysis in Sect. 8. We defer some of the proofs and details on the Configuration Extension Algorithm to the appendices as these are not necessary to understand the main part.

2 Preliminaries

Notations. We denote by $\mathsf{S}^n \subset \mathbb{R}^{n+1}$ the n-dimensional unit sphere. We use soft-$\mathcal{O}$ notation to denote running times: $T = \widetilde{\mathcal{O}}(2^{cn})$ means that we suppress subexponential factors. We use sub-indices $\mathcal{O}_k(.)$ in the $\mathcal{O}$-notation to stress that the asymptotic result holds for k fixed. For any set $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ of vectors in some $\mathbb{R}^n$, the *Gram matrix* $C \in \mathbb{R}^{k \times k}$ is given by the set of pairwise scalar products. It is a complete invariant of the $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ up to simultaneous rotation and reflection of all $\boldsymbol{x}_i$'s. For such matrices $C \in \mathbb{R}^{k \times k}$ and $I \subset \{1, \ldots, k\}$, we write $C[I]$ for the appropriate $|I| \times |I|$-submatrix with rows and columns from I.

As we consider distances wrt. the ℓ_2-norm, the approximate k-List problem we consider in this work is the following computational problem:

Definition 1 (Approximate k-List problem). *Let $0 < t < \sqrt{k}$. Assume we are given k lists $L_1, \ldots, L_k$ of equal exponential size, whose entries are iid. uniformly chosen vectors from the n-sphere S^n. The task is to output an $1-o(1)$-fraction of all solutions, where solutions are k-tuples $\boldsymbol{x}_1 \in L_1, \ldots, \boldsymbol{x}_k \in L_k$ satisfying $\|\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_k\|^2 \leq t^2$.*

We consider the case where t, k are constant and the input lists are of size c^n for some constant $\mathsf{c} > 1$. We are interested in the asymptotic complexity for $n \to \infty$. To simplify the exposition, we pretend that we can compute with real numbers; all our algorithms work with sufficiently precise approximations (possibly losing an $o(1)$-fraction of solutions due to rounding). This does not affect the asymptotics. Note that the problem becomes trivial for $t > \sqrt{k}$, since all but an $1 - o(1)$-fraction of k-tuples from $L_1 \times \cdots \times L_k$ satisfy $\|\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_k\|^2 \approx k$ (random $\boldsymbol{x}_i \in \mathsf{S}^n$ are almost orthogonal with high probability, cf. Thm. 1). In the case $t > \sqrt{k}$, we need to ask that $\|\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_k\|^2 \geq t^2$ to get a meaningful problem. Then all our results apply to the case $t > \sqrt{k}$ as well.

In our definition, we allow to drop a $o(1)$-fraction of solutions, which is fine for the sieving applications. In fact, we will propose an algorithm that drops an exponentially small fraction of solutions and our asymptotic improvement compared to BLS crucially relies on dropping more solutions than BLS. For this reason, we are only interested in the case where the expected number of solutions is exponential.

Relation to the approximate Shortest Vector Problem. The main incentive to look at the approximate k-List problem (as in Def. 1) is its straightforward application to the so-called sieving algorithms for the shortest vector problem (SVP) on an n-dimensional lattice (see Sect. 7.2 for a more comprehensive discussion). The complexity of these sieving algorithms is completely determined by the complexity of an approximate k-List solver called as main subroutine. So one can instantiate a lattice sieving algorithm using an approximate k-List solver (the ability to choose k allows a memory-efficient instantiations of such a solver). This is observed and fully explained in [4]. For $k = 3$, the running time for the SVP algorithm presented in [4] is $2^{0.4812n+o(n)}$ requiring $2^{0.1887n+o(n)}$ memory. Running our Alg. 1 instead as a k-List solver within the SVP sieving, one obtains

a running time of $2^{0.3962n+o(n)}$ with the same memory complexity $2^{0.1887n+o(n)}$. As explained in Sect. 7.2, we can reduce the running time even further down to $2^{0.3717n+o(n)}$ with no asymptotic increase in memory by using a combination of Alg. 1 and the LSH-like Configuration Extension Algorithm. This combined algorithm is fully described in the full version of the paper.

In the applications to sieving, we have $t = 1$ and actually look for solutions $\|\pm\boldsymbol{x}_1 \pm \cdots \pm \boldsymbol{x}_k\| \leq 1$ with arbitrary signs. This is clearly equivalent by considering the above problem separately for each of the $2^k = \mathcal{O}(1)$ choices of signs. Further, the lists $L_1, \ldots, L_k$ can actually be equal. Our algorithm works for this case as well. In these settings, some obvious optimizations are possible, but they do not affect the asymptotics.

Our methods are also applicable to lists of different sizes, but we stick to the case of equal list sizes to simplify the formulas for the running times.

3 Configurations

Whether a given k-tuple $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ is a solution to the approximate k-List problem is invariant under simultaneous rotations/reflections of all $\boldsymbol{x}_i$ and we want to look at k-tuples up to such symmetry by what we call configurations of points. As we are concerned with the ℓ_2-norm, a complete invariant of k-tuples up to symmetry is given by the set of pairwise scalar products and we define configurations for this norm:

Definition 2 (Configuration). *The* configuration $C = \mathrm{Conf}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ *of k points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathsf{S}^n$ is defined as the Gram matrix $C_{i,j} = \langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle$.*

Clearly, the configuration of the k-tuple $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ determines the length of the sum $\|\sum_i \boldsymbol{x}_i\|$:

$$\left\|\sum_i \boldsymbol{x}_i\right\|^2 = \sum_{i,j} \langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle = k + 2\sum_{i<j} \langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle. \tag{1}$$

We denote by

$$\begin{aligned} \mathscr{C} &= \{C \in \mathbb{R}^{k \times k} \mid C \text{ symmetric positive semi-definite, } C_{i,i} = 1 \; \forall i\}, \\ \mathscr{C}_{\leq t} &= \{C \in \mathscr{C} \mid \sum\nolimits_{i,j} C_{i,j} \leq t^2\} \subset \mathscr{C} \end{aligned}$$

the spaces of all possible configurations resp. those which give a length of at most t. The spaces $\mathscr{C}$ and $\mathscr{C}_{\leq t}$ are compact and convex. For fixed k, it is helpful from an algorithmic point of view to think of $\mathscr{C}$ as a finite set: for any $\varepsilon > 0$, we can cover $\mathscr{C}$ by finitely many ε-balls, so we can efficiently enumerate $\mathscr{C}$.

In the context of the approximate k-List problem with target length t, a k-tuple $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ is a solution iff $\mathrm{Conf}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \in \mathscr{C}_{\leq t}$. For that reason, we call a configuration in $\mathscr{C}_{\leq t}$ *good*. An obvious way to solve the approximate k-List problem is to enumerate over all good configurations and solve the following k-List configuration problem:

Definition 3 (Configuration problem). *On input k exponentially-sized lists $L_1, \ldots, L_k$ of vectors from S^n, a target configuration $C \in \mathscr{C}$ and some $\varepsilon > 0$, the task is to output all k-tuples $\boldsymbol{x}_1 \in L_1, \ldots, \boldsymbol{x}_k \in L_k$, such that $|\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle - C_{ij}| \leq \varepsilon$ for all i, j. Such k-tuples are called solutions to the problem.*

Remark 1. Due to $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$ taking real values, it does not make sense to ask for exact equality to C, but rather we introduce some $\varepsilon > 0$. We shorthand write $C \approx_\varepsilon C'$ for $|C_{i,j} - C'_{i,j}| \leq \varepsilon$. Formally, our analysis will show that for fixed $\varepsilon > 0$, we obtain running times and list sizes of the form $\widetilde{\mathcal{O}}_\varepsilon(2^{(c+f(\varepsilon))n})$ for some unspecified continuous f with $\lim_{\varepsilon \to 0} f(\varepsilon) = 0$. Letting $\varepsilon \to 0$ sufficiently slowly, we absorb $f(\varepsilon)$ into the $\widetilde{\mathcal{O}}(.)$-notation and omit it.

As opposed to the approximate k-List problem, being a solution to the k-List configuration problem is a locally checkable property[12]: it is a conjunction of conditions involving only *pairs* $\boldsymbol{x}_i, \boldsymbol{x}_j$. It is this and the following observation that we leverage to improve on the results of [4].

It turns out that the configurations attained by the solutions to the approximate k-List problem are concentrated around a single good configuration, which is the good configuration with the highest amount of symmetry. So in fact, we only need to solve the configuration problem for this particular good configuration. The following theorem describes the distribution of configurations:

Theorem 1. *Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathsf{S}^n$ be independent, uniformly distributed on the n-sphere, $n > k$. Then the configuration $C = C(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ follows a distribution $\mu_{\mathscr{C}}$ on $\mathscr{C}$ with density given by*

$$\mu_{\mathscr{C}} = W_{n,k} \cdot \det(C)^{\frac{1}{2}(n-k)} \mathrm{d}\mathscr{C} = \widetilde{\mathcal{O}}_k\Big(\det(C)^{\frac{n}{2}}\Big)\mathrm{d}\mathscr{C},$$

where $W_{n,k} = \pi^{-\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n+1-i}{2})} = \mathcal{O}_k\big(n^{\frac{k(k-1)}{4}}\big)$ is a normalization constant that only depends on n and k. Here, the reference measure $\mathrm{d}\mathscr{C}$ is given by $\mathrm{d}\mathscr{C} = \mathrm{d}C_{1,2} \cdots \mathrm{d}C_{(k-1),k}$ (i.e. the Lebesgue measure in a natural parametrization).

Proof. We derive this by an approximate normalization of the so-called Wishart distribution[20]. Observe that we can sample $C \leftarrow \mu_{\mathscr{C}}$ in the following way:
We sample $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathbb{R}^{n+1}$ iid from spherical $n+1$-dimensional Gaussians, such that the direction of each $\boldsymbol{x}_i$ is uniform over S^n. Note that the lengths of the $\boldsymbol{x}_i$ are not normalized to 1. Then we set $A_{i,j} := \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. Finally, normalize to $C_{i,j} := \frac{A_{i,j}}{\sqrt{A_{i,i} A_{j,j}}}$.

The joint distribution of the $A_{i,j}$ is (by definition) given by the so-called Wishart distribution.[20] Its density for $n + 1 > k - 1$ is known to be

$$\rho_{\text{Wishart}} = \frac{e^{-\frac{1}{2}\operatorname{Tr} A} \cdot \det(A)^{\frac{n+1-k-1}{2}}}{2^{\frac{(n+1)k}{2}} \pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n+1-i}{2})} \mathrm{d}A \tag{2}$$

where the reference density $\mathrm{d}A$ is given by $\mathrm{d}A = \prod_{i\le j} \mathrm{d}A_{i,j}$. We refer to [8] for a relatively simple computation of that density. Consider the change of variables on $\mathbb{R}^{k(k+1)/2}$ given by

$$\begin{aligned}&\Phi\big(A_{1,1}, A_{2,2}, \dots, A_{k,k}, A_{1,2}, \dots, A_{k-1,k}\big)\\ &\quad = \big(A_{1,1}, A_{2,2}, \dots, A_{k,k}, \tfrac{A_{1,2}}{\sqrt{A_{1,1}A_{2,2}}}, \dots, \tfrac{A_{k,k-1}}{\sqrt{A_{k-1,k-1}A_{k,k}}}\big),\end{aligned}$$

i.e. we map the $A_{i,j}$'s to $C_{i,j}$'s while keeping the $A_{i,i}$'s to make the transformation bijective almost everywhere. The Jacobian $D\Phi$ of Φ is a triangular matrix and its determinant is easily seen to be

$$\big|\det\big(D\Phi\big)\big| = \prod_i \frac{1}{\sqrt{A_{i,i}}^{k-1}}.$$

Further, note that $A = TCT$, where T is a diagonal matrix with diagonal $\sqrt{A_{1,1}}, \dots, \sqrt{A_{k,k}}$. In particular, $\det(A) = \det(C)\cdot\prod_i A_{i,i}$. Consequently, we can transform the Wishart density into $\big(A_{1,1}, \dots, A_{k,k}, C_{1,2}, \dots, C_{k-1,k}\big)$-coordinates as

$$\rho_{\text{Wishart}} = \frac{e^{-\frac{1}{2}\sum_i A_{i,i}} \det(C)^{\frac{n-k}{2}} \prod_i A_{i,i}^{\frac{n-k}{2}}}{2^{\frac{(n+1)k}{2}} \pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})} \prod_i \sqrt{A_{i,i}}^{k-1} \prod_i \mathrm{d}A_{i,i} \prod_{i<j} \mathrm{d}C_{i,j}.$$

The desired $\mu_{\mathscr{C}}$ is obtained from ρ_{Wishart} by integrating out $\mathrm{d}A_{1,1}\mathrm{d}A_{2,2}\cdots\mathrm{d}A_{k,k}$. We can immediately see that $\mu_{\mathscr{C}}$ takes the form $\mu_{\mathscr{C}} = W_{n,k} \det(C)^{\frac{n-k}{2}} \mathrm{d}\mathscr{C}$ for some constants $W_{n,k}$. We compute $W_{n,k}$ as

$$\begin{aligned}W_{n,k} &= \int_{A_{1,1}} \cdots \int_{A_{k,k}} \frac{e^{-\frac{1}{2}\sum_i A_{i,i}} \prod_i A_{i,i}^{\frac{n-k}{2}}}{2^{\frac{(n+1)k}{2}} \pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})} \prod_i \sqrt{A_{i,i}}^{k-1} \prod_i \mathrm{d}A_{i,i}\\ &= \frac{1}{2^{\frac{(n+1)k}{2}} \pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})} \Big(\int_{A_{1,1}=0}^{+\infty} A_{1,1}^{\frac{n-1}{2}} e^{-\frac{1}{2}A_{1,1}}\, \mathrm{d}A_{1,1}\Big)^k\\ &= \frac{2^{\frac{(n+1)k}{2}}}{2^{\frac{(n+1)k}{2}} \pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})} \Big(\int_{A_{1,1}=0}^{+\infty} \big(\tfrac{A_{1,1}}{2}\big)^{\frac{n+1}{2}-1} e^{-\frac{1}{2}A_{1,1}}\, \tfrac{1}{2}\mathrm{d}A_{1,1}\Big)^k\\ &= \frac{1}{\pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})} \Big(\int_{x=0}^{+\infty} x^{\frac{n+1}{2}-1} e^{-x}\, \mathrm{d}x\Big)^k\\ &= \frac{\Gamma(\frac{n+1}{2})^k}{\pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})}.\end{aligned}$$

Finally, note that as a consequence of Stirling's formula, we have $\frac{\Gamma(n+z)}{\Gamma(n)} = \mathcal{O}_z(n^z)$ for any fixed z and $n \to \infty$. From this, we get

$$W_{n,k} = \frac{\Gamma(\frac{n+1}{2})^k}{\pi^{\frac{k(k-1)}{4}} \prod_{i=0}^{k-1} \Gamma(\frac{n-i+1}{2})} = \mathcal{O}_k\Big(n^{\sum_{i=0}^{k-1} \frac{i}{2}}\Big) = \mathcal{O}_k\Big(n^{\frac{k(k-1)}{4}}\Big).$$

The configurations C that we care about the most have the highest amount of symmetry. We call a configuration C balanced if $C_{i,j} = C_{i',j'}$ for all $i \neq j$, $i' \neq j'$. To compute the determinant $\det(C)$ for such balanced configurations, we have the following lemma:

Lemma 1.

$$\textit{Let } C = \begin{pmatrix} 1 & a & a & \dots & a \\ a & 1 & a & \dots & a \\ a & a & 1 & \dots & a \\ \vdots & & & \ddots & \vdots \\ a & a & a & \dots & 1 \end{pmatrix} \in \mathbb{R}^{k \times k}.$$

Then $\det(C) = (1-a)^{k-1}(1+(k-1)a)$.

Proof. We have $C = (1-a) \cdot \mathbb{1}_k + a \cdot \mathbf{1} \cdot \mathbf{1}^{\mathsf{t}}$, where $\mathbf{1} \in \mathbb{R}^{k \times 1}$ is an all-ones vector. Sylvester's Determinant Theorem[2] gives

$$\begin{aligned} \det(C) &= (1-a)^k \det\big(\mathbb{1}_k + \tfrac{a}{1-a}\mathbf{1} \cdot \mathbf{1}^{\mathsf{t}}\big) = (1-a)^k \det\big(\mathbb{1}_1 + \tfrac{a}{1-a}\mathbf{1}^{\mathsf{t}} \cdot \mathbf{1}\big) \\ &= (1-a)^k(1 + \tfrac{a}{1-a}k) = (1-a)^{k-1}(1+(k-1)a). \end{aligned}$$

For fixed k and C, the probability density $\widetilde{\mathcal{O}}\big(\det(C)^{\frac{n}{2}}\big)$ of $\mu_{\mathscr{C}}$ is exponential in n. Since $C \in \mathscr{C}$ can only vary in a compact space, taking integrals will asymptotically pick the maximum value: in particular, we have for the probability that a uniformly random k-tuple $\boldsymbol{x}_1, \dots, \boldsymbol{x}_k$ is good:

$$\int_{C \text{ good}} \mu_{\mathscr{C}} = \widetilde{\mathcal{O}}\Big(\max_{C \text{ good}} \det(C)^{\frac{n}{2}}\Big). \tag{3}$$

We now compute this maximum.

Theorem 2. *Let $0 < t < \sqrt{k}$ be some target length and consider the subset $\mathscr{C}_{\leq t} \subset \mathscr{C}$ of good configurations for target length at most t. Then* $\det(C)$ *attains its unique maximum over $\mathscr{C}_{\leq t}$ at the balanced configuration $C_{\mathrm{Bal},t}$, defined by $C_{i,j} = \frac{t^2-k}{k^2-k}$ for all $i \neq j$ with maximal value*

$$\det(C)_{\max} = \det(C_{\mathrm{Bal},t}) = \frac{t^2}{k}\Big(\frac{k^2-t^2}{k^2-k}\Big)^{k-1}.$$

In particular, for $t = 1$, this gives $C_{i,j} = -\frac{1}{k}$ and $\det(C)_{\max} = \frac{(k+1)^{k-1}}{k^k}$. Consequently, for any fixed k and any fixed $\varepsilon > 0$, the probability that a randomly chosen solution to the approximate k-List problem is ε-close to $C_{\mathrm{Bal},t}$ converges exponentially fast to 1 as $n \to \infty$.

Proof. It suffices to show that C is balanced at the maximum, i.e. that all $C_{i,j}$ with $i \neq j$ are equal. Then computing the actual values is straightforward from (1) and Lemma 1. Assume $k \geq 3$, as there is nothing to show otherwise.

For the proof, it is convenient to replace the conditions $C_{i,i} = 1$ for all i by the (weaker) condition $\mathrm{Tr}(C) = k$. Let $\mathscr{C}'_{\leq t}$ denote the set of all symmetric, positive semi-definite $C \in \mathbb{R}^{k \times k}$ with $\mathrm{Tr}(C) = k$ and $\sum_{i,j} C_{i,j} \leq t^2$. We maximize $\det(C)$ over $\mathscr{C}'_{\leq t}$ and our proof will show that $C_{i,i} = 1$ is satisfied at the maximum.

Let $C \in \mathscr{C}'_{\leq t}$. Since C is symmetric, positive semi-definite, there exists an orthonormal basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ of eigenvectors with eigenvalues $0 \leq \lambda_1 \leq \ldots \leq \lambda_k$.

Clearly, $\sum_i \lambda_i = \mathrm{Tr}(C) = k$ and our objective $\det(C)$ is given by $\det(C) = \prod_i \lambda_i$. We can write $\sum_{i,j} C_{i,j}$ as $\mathbf{1}^{\mathsf{t}} C \mathbf{1}$ for an all-ones vector $\mathbf{1}$. We will show that if $\det(C)$ is maximal, then $\mathbf{1}$ is an eigenvector of C. Since

$$t^2 \geq \mathbf{1}^{\mathsf{t}} C \mathbf{1} \geq \lambda_1 \|\mathbf{1}\|^2 = k\lambda_1, \tag{4}$$

for the smallest eigenvalue λ_1 of C, we have $\lambda_1 \leq \frac{t^2}{k} < 1$. For fixed λ_1, maximizing $\det(C) = \lambda_1 \cdot \prod_{i=2}^{k} \lambda_i$ under $\sum_{i=2}^{k} \lambda_i = k - \lambda_1$ gives (via the Arithmetic Mean-Geometric Mean Inequality)

$$\det(C) \leq \lambda_1 \Big(\frac{k-\lambda_1}{k-1}\Big)^{k-1}.$$

The derivative of the right-hand side wrt. λ_1 is $\frac{k(1-\lambda_1)}{k-1}\big(\frac{k-\lambda_1}{k-1}\big)^{k-2} > 0$, so we can bound it by plugging in the maximal $\lambda_1 = \frac{t^2}{k}$:

$$\det(C) \leq \lambda_1 \Big(\frac{k-\lambda_1}{k-1}\Big)^{k-1} \leq \frac{t^2}{k}\Big(\frac{k-\frac{t^2}{k}}{k-1}\Big)^{k-1} = \frac{t^2}{k}\Big(\frac{k^2-t}{k^2-k}\Big)^{k-1} \tag{5}$$

The inequalities (5) are satisfied with equality iff $\lambda_2 = \ldots = \lambda_k$ and $\lambda_1 = \frac{t^2}{k}$. In this case, we can compute the value of λ_2 as $\lambda_2 = \frac{k^2-t^2}{k(k-1)}$ from $\mathrm{Tr}(C) = k$. The condition $\lambda_1 = \frac{t^2}{k}$ means that (4) is satisfied with equality, which implies that $\mathbf{1}$ is an eigenvector with eigenvalue λ_1. So wlog. $\boldsymbol{v}_1 = \frac{1}{\sqrt{k}}\mathbf{1}$. Since the $\boldsymbol{v}_i$'s are orthonormal, we have $\mathbb{1}_k = \sum_i \boldsymbol{v}_i \boldsymbol{v}_i{}^{\mathsf{t}}$, where $\mathbb{1}_k$ is the $k \times k$ identity matrix. Since we can write C as $C = \sum_i \lambda_i \boldsymbol{v}_i \boldsymbol{v}_i{}^{\mathsf{t}}$, we obtain

$$C = \sum_i \lambda_i \boldsymbol{v}_i \boldsymbol{v}_i{}^{\mathsf{t}} = (\lambda_1 - \lambda_2)\boldsymbol{v}_1 \boldsymbol{v}_1{}^{\mathsf{t}} + \lambda_2 \sum_{i=1}^{k} \boldsymbol{v}_i \boldsymbol{v}_i{}^{\mathsf{t}} = \frac{\lambda_1 - \lambda_2}{k}\mathbf{1}\mathbf{1}^{\mathsf{t}} + \lambda_2 \cdot \mathbb{1}_k,$$

for $\det(C)$ maximal. From $C = \frac{\lambda_1-\lambda_2}{k}\mathbf{1}\mathbf{1}^{\mathsf{t}} + \lambda_2 \cdot \mathbb{1}_k$, we see that all diagonal entries of C are equal to $\lambda_2 + \frac{\lambda_1-\lambda_2}{k}$ and the off-diagonal entries are all equal to $\frac{\lambda_1-\lambda_2}{k}$. So all $C_{i,i}$ are equal with $C_{i,i} = 1$, because $\mathrm{Tr}(C) = k$, and C is balanced.

For the case $t > \sqrt{k}$, and $\mathscr{C}_{\leq t}$ replaced by $\mathscr{C}_{\geq t}$, the statement can be proven analogously. Note that we need to consider the largest eigenvalue rather than the smallest in the proof. We remark that for $t = 1$, the condition $\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle = C_{i,j} = -\frac{1}{k}$ for all $i \neq j$ is equivalent to saying that $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ are k points of a regular $k+1$-simplex whose center is the origin. The missing $k+1^{\text{th}}$ point of the simplex is $-\sum_i \boldsymbol{x}_i$, i.e. the negative of the sum (see Fig. (1)).

A corollary of our concentration result is the following formula for the expected size of the output lists in the approximate k-List problem.

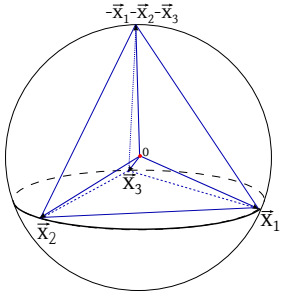


Fig. 1: A regular tetrahedron (3−simplex) represents a balanced configuration for $k = 3$.

Corollary 1. *Let k, t be fixed. Then the expected number of solutions to the approximate k-List problem with input lists of length $|L|$ is*

$$\mathbb{E}[\#solutions] = \widetilde{\mathcal{O}}\left(|L|^k \left(\frac{t^2}{k}\left(\frac{k^2-t^2}{k^2-k}\right)^{k-1}\right)^{\frac{n}{2}}\right). \tag{6}$$

Proof. By Thms. 2 and 1, the probability that any k-tuple is a solution is given by $\widetilde{\mathcal{O}}(\det(C_{\mathrm{Bal},t})^{\frac{n}{2}})$. The claim follows immediately.

In particular, this allows us to prove the following conjecture of [4]:

Theorem 3. *Let k be fixed and $t = 1$. If in the approximate k-List problem, the length $|L|$ of each input list is equal to the expected length of the output list, then $|L| = \widetilde{\mathcal{O}}\left(\left(\frac{k^{\frac{k}{k-1}}}{k+1}\right)^{\frac{n}{2}}\right)$.*

Proof. This follows from simple algebraic manipulation of (6).

Our concentration result shows that it is enough to solve the configuration problem for $C_{\mathrm{Bal},t}$.

Corollary 2. *Let k, t be fixed. Then the approximate k-List problem with target length t can be solved in essentially the same time as the k-List configuration problem with target configuration $C_{\mathrm{Bal},t}$ for any fixed $\varepsilon > 0$.*

Proof. On input $L_1, \ldots, L_k$, solve the k-List configuration problem with target configuration $C_{\mathrm{Bal},t}$. Restrict to those solutions whose sum has length at most t. By Thm. 2, this will find all but an exponentially small fraction of solutions to the approximate k-List problem. Since we only need to output a $1-o(1)$-fraction of the solutions, this solves the problem.

4 Algorithm

In this section we present our algorithm for the Configuration problem (Def. 3). On input it receives k lists $L_1, \ldots, L_k$, a target configuration C in the form of a

Gram matrix $C_{i,j} = \langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle \in \mathbb{R}^{k \times k}$ and a small $\varepsilon > 0$. The algorithm proceeds as follows: it picks an $\boldsymbol{x}_1 \in L_1$ and filters all the remaining lists with respect to the values $\langle \boldsymbol{x}_1 \, , \boldsymbol{x}_i \rangle$ for all $2 \leq i \leq k$. More precisely, $\boldsymbol{x}_i \in L_i$ 'survives' the filter if $|\langle \boldsymbol{x}_1 \, , \boldsymbol{x}_i \rangle - C_{1,i}| \leq \varepsilon$. We put such an $\boldsymbol{x}_i$ into $L_i^{(1)}$ (the superscript indicates how many filters were applied to the original list L_i). On this step, all the k-tuples of the form $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k) \in \{\boldsymbol{x}_1\} \times L_2^{(1)} \times \ldots \times L_k^{(1)}$ with a fixed first component $\boldsymbol{x}_1$ partially match the target configuration: all scalar products involving $\boldsymbol{x}_1$ are as desired. In addition, the lists $L_i^{(1)}$ become much shorter than the original ones.

Next, we choose an $\boldsymbol{x}_2 \in L_2^{(1)}$ and create smaller lists $L_i^{(2)}$ from $L_i^{(1)}$ by filtering out all the $\boldsymbol{x}_i \in L_i^{(1)}$ that do not satisfy $|\langle \boldsymbol{x}_2 \, , \boldsymbol{x}_i \rangle - C_{2,i}| \leq \varepsilon$ for all $3 \leq i \leq k$. A tuple of the form $(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots, \boldsymbol{x}_k) \in \{\boldsymbol{x}_1\} \times \{\boldsymbol{x}_2\} \times L_3^{(2)} \times \ldots \times L_k^{(2)}$ satisfies the target configuration $C_{i,j}$ for $i = 1, 2$. We proceed with this list-filtering strategy until we have fixed all $\boldsymbol{x}_i$ for $1 \leq i \leq k$. We output all such k-tuples. Note that our algorithm becomes the trivial brute-force algorithm once we are down to 2 lists to be processed. As soon as we have fixed $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{k-2}$ and created $L_{k-1}^{(k-2)}, L_k^{(k-2)}$, our algorithm iterates over $L_{k-1}^{(k-2)}$ and checks the scalar product with every element from $L_k^{(k-2)}$.

Our algorithm is detailed in Alg. 1 and illustrated in Fig. (2a).

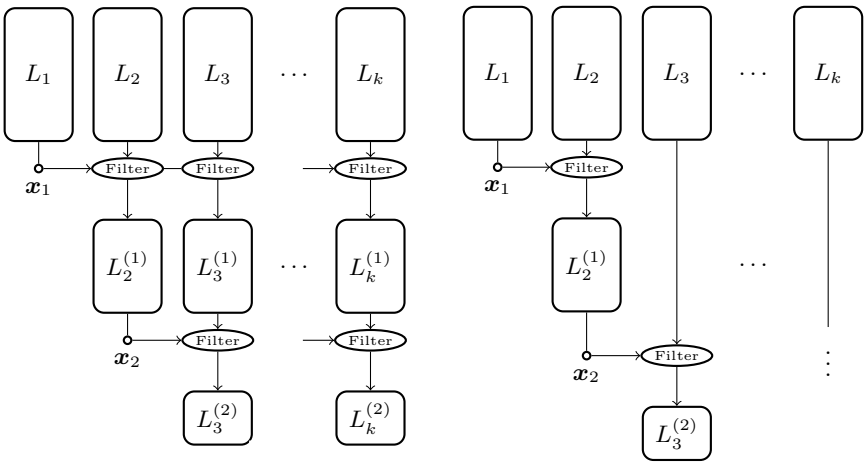
Algorithm 1 k-List for the Configuration Problem

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

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

$L_{\text{out}} \leftarrow \{\}$
for all $\boldsymbol{x}_1 \in L_1$ **do**
  for all $j = 2 \ldots k$ **do**
    $L_j^{(1)} \leftarrow \textsc{Filter}(\boldsymbol{x}_1, L_j, C_{1,j}, \varepsilon)$
  for all $\boldsymbol{x}_2 \in L_2^{(1)}$ **do**
    for all $j = 3 \ldots k$ **do**
      $L_j^{(2)} \leftarrow \textsc{Filter}(\boldsymbol{x}_2, L_j^{(1)}, C_{2,j}, \varepsilon)$
    $\ddots$
      for all $\boldsymbol{x}_k \in L_k^{(k-1)}$ **do**
        $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\boldsymbol{x}_1, \ldots \boldsymbol{x}_k)\}$
return L_{out}

function $\textsc{Filter}(\boldsymbol{x}, L, c, \varepsilon)$
  $L' \leftarrow \{\}$
  for all $\boldsymbol{x}' \in L$ **do**
    if $|\langle \boldsymbol{x} \, , \boldsymbol{x}' \rangle - c| \leq \varepsilon$ **then**
      $L' \leftarrow L' \cup \{\boldsymbol{x}'\}$
  return L'



(a) Pictorial representation of Alg. 1. At level i, a filter receives as input $\boldsymbol{x}_i$ and a vector $\boldsymbol{x}_j$ from $L_j^{(i-1)}$ (for the input lists, $L = L^{(0)}$). $\boldsymbol{x}_j$ passes through the filter if $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle - C_{i,j}| \leq \varepsilon$, in which case it is added to $L_j^{(i)}$. The configuration C is a global parameter.

(b) The k-List algorithm given in [4]. The main difference is that a filter receives as inputs $\boldsymbol{x}_i$ and a vector $\boldsymbol{x}_j \in L_j$, as opposed to $\boldsymbol{x}_j \in L_j^{(i-1)}$. Technically, in [4], $\boldsymbol{x}_i$ survives the filter if $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_1 + \ldots + \boldsymbol{x}_{i-1} \rangle| \geq c_i$ for some predefined c_i. Due to our concentration results, this description is equivalent to the one given in [4] in the sense that the returned solutions are (up to a subexponential fraction) the same.

Fig. 2: k-List Algorithms for the Configuration Problem. Left: Our Alg. 1. Right: k-tuple sieve algorithm of [4]

5 Analysis

In this section we analyze the complexity of Alg. 1 for the Configuration problem. First, we should mention that the memory complexity is completely determined by the input list-sizes $|L_i|$ (remember that we restrict to constant k) and it does not change the asymptotics when we apply k filters. In practice, all intermediate lists $L_i^{(j)}$ can be implemented by storing pointers to the elements of the original lists.

In the following, we compute the expected sizes of filtered lists $L_i^{(j)}$ and establish the expected running time of Alg. 1. Since our algorithm has an exponential running time of 2^{cn} for some $c = \Theta(1)$, we are interested in determining c (which depends on k) and we ignore polynomial factors, e.g. we do not take into account time spent for computing inner products.

Theorem 4. *Let k be fixed. Alg. 1 given as input k lists $L_1, \ldots, L_k \subset \mathsf{S}^n$ of the same size $|L|$, a target balanced configuration $C_{\mathrm{Bal},t} \in \mathbb{R}^{k\times k}$, a target length $0 < t < \sqrt{k}$, and $\varepsilon > 0$, outputs the list L_{out} of solutions to the Configuration problem. The expected running time of Alg. 1 is*

$$T = \widetilde{\mathcal{O}}\Big(|L| \cdot \max_{1\le i\le k-1} |L|^i \cdot \frac{(k^2-t^2)^i}{(k^2-k)^{i+1}} \cdot \Big(\frac{(k^2-k+(i-1)(t^2-k))^2}{k^2-k+(i-2)(t^2-k)}\Big)^{\frac{n}{2}}\Big). \quad (7)$$

In particular, for $t = 1$ and $|L_{\mathrm{out}}| = |L|$ it holds that

$$T = \widetilde{\mathcal{O}}\left(\Big(\frac{k^{\frac{1}{k-1}}}{k+1} \cdot \max_{1\le i\le k-1} k^{\frac{i}{k-1}} \cdot \frac{(k-i+1)^2}{k-i+2}\Big)^{\frac{n}{2}}\right). \quad (8)$$

Remark 2. In the proof below we also show that the expected running time of the k-List algorithm presented in [4] is (see also Fig. (3) for a comparison) for $t = 1, |L_{\mathrm{out}}| = |L|$

$$T_{\mathrm{BLS}} = \widetilde{\mathcal{O}}\Big(\Big(\frac{k^{\frac{k}{k-1}}}{(k+1)^2} \cdot \max_{1\le i\le k-1}\big(k^{\frac{i}{k-1}} \cdot (k-i+1)\big)\Big)^{\frac{n}{2}}\Big). \quad (9)$$

Corollary 3. *For $k = 3$, $t = 1$, and $|L| = |L_{\mathrm{out}}|$ (the most interesting setting for SVP), Alg. 1 has running time*

$$T = 2^{0.3962n+o(n)}, \quad (10)$$

requiring $|L| = 2^{0.1887n+o(n)}$ memory.

Proof (Proof of Thm. 4). The correctness of the algorithm is straightforward: let us associate the lists $L^{(i)}$ with a level i where i indicates the number of filtering steps applied to L (we identify the input lists with the 0^{th} level: $L_i = L_i^{(0)}$). So for executing the filtering for the i^{th} time, we choose an $\boldsymbol{x}_i \in L_i^{(i-1)}$ that satisfies the condition $|\langle \boldsymbol{x}_i\,, \boldsymbol{x}_{i-1}\rangle - C_{i,i-1}| \le \varepsilon$ (for a fixed $\boldsymbol{x}_{i-1}$) and append to

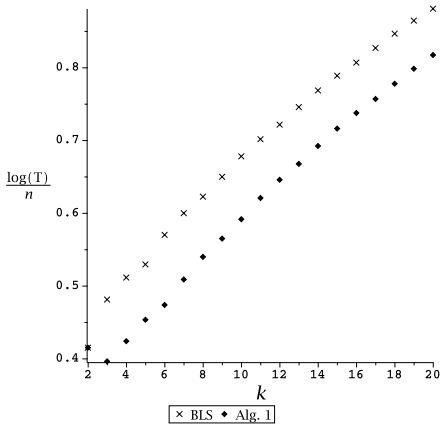


Fig. 3: Running exponents scaled by $1/n$ for the target length $t = 1$. For $k = 2$, both algorithms are the Nguyen-Vidick sieve [18] with $\log(T)/n = 0.415$ (naive brute-force over two lists). For $k = 3$, Algorithm 1 achieves $\log(T)/n = 0.3962$.

a previously obtained $(i-1)$-tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1})$. Thus on the last level, we put into L_{out} a k-tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ that is a solution to the Configuration problem.

Let us first estimate the size of the list $L_i^{(i-1)}$ output by the filtering process applied to the list $L_i^{(i-2)}$ for $i > 1$ (i.e. the left-most lists on Fig. (2a)). Recall that all elements $\boldsymbol{x}_i \in L_i^{(i-1)}$ satisfy $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle - C_{i,j}| \leq \varepsilon, \ 1 \leq j \leq i-1$. Then the *total* number of i-tuples $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_i) \in L_1 \times L_2^{(1)} \times \ldots \times L_i^{(i-1)}$ considered by the algorithm is determined by the probability that in a random i-tuple, all pairs $(\boldsymbol{x}_j, \boldsymbol{x}_{j'}), 1 \leq j, j' \leq i$ satisfy the inner product constraints given by $C_{j,j'}$. This probability is given by Thm. 1 and since the input lists are of the same size $|L|$, we have[1]

$$|L_1| \cdot |L_2^{(1)}| \cdot \ldots |L_i^{(i-1)}| = |L|^i \cdot \det(C[1 \ldots i])^{\frac{n}{2}}, \tag{11}$$

where $\det(C[1 \ldots i])$ denotes the i-th principal minor of C. Using (11) for two consecutive values of i and dividing, we obtain

$$|L_{i+1}^{(i)}| = |L| \cdot \Big(\frac{\det(C[1 \ldots i+1]}{\det(C[1 \ldots i])} \Big)^{\frac{n}{2}}. \tag{12}$$

Note that these expected list sizes can be smaller than 1. This should be thought of as the inverse probability that the list is not empty. Since we target a balanced configuration $C_{\text{Bal},t}$, the entries of the input Gram matrix are specified by Thm. 2 and, hence, we compute the determinants in the above quotient by applying Lemma 1 for $a = \frac{t^k - k}{k^2 - k}$. Again, from the shape of the Gram matrix $C_{\text{Bal},t}$ and the equal-sized input lists, it follows that the filtered list on each level are of the same size: $|L_{i+1}^{(i)}| = |L_{i+2}^{(i)}| = \ldots = |L_k^{(i)}|$. Therefore, for all filtering levels $0 \leq j \leq k-1$ and for all $j+1 \leq i \leq k$,

$$\big|L_i^{(j)}\big| = |L| \cdot \Big(\frac{k^2 - t^2}{k^2 - k} \cdot \frac{k^2 - k + j(t^2 - k)}{k^2 - k + (j-1)(t^2 - k)} \Big)^{\frac{n}{2}}. \tag{13}$$

[1] Throughout this proof, the equations that involve list-sizes $|L|$ and running time T are assumed to have $\widetilde{\mathcal{O}}(\cdot)$ on the right-hand side. We omit it for clarity.

Now let us discuss the running time. Clearly, the running time of Alg. 1 is (up to subexponential factors in n)

$$T = |L_1^{(0)}| \cdot (|L_2^{(0)}| + |L_2^{(1)}| \cdot (|L_3^{(1)}| + |L_3^{(2)}| \cdot (\ldots \cdot (|L_k^{(k-2)}| + |L_k^{(k-1)}|))).$$

Multiplying out and observing that $|L_k^{(k-2)}| > |L_k^{(k-1)}|$, so we may ignore the very last term, we deduce that the total running time is (up to subexponential factors) given by

$$T = |L| \cdot \max_{1 \le i \le k-1} |L^{(i-1)}| \cdot \prod_{j=1}^{i-1} |L^{(j)}|, \tag{14}$$

where $|L^{(j)}|$ is the size of any filtered list on level j (so we omit the subscripts). Consider the value $i_{\max}$ of i where the maximum is attained in the above formula. The meaning of $i_{\max}$ is that the total cost over all loops to create the lists $L_j^{(i_{\max})}$ is dominating the running time. At this level, the lists $L_j^{(i_{\max})}$ become small enough such that iterating over them (i.e. creation of $L_j^{(i_{\max}+1)}$) does not contribute asymptotically. Plugging in Eqns. (11) and (12) into (14), we obtain

$$T = |L| \cdot \max_{1 \le i \le k-1} |L|^i \Big(\frac{(\det C[1 \ldots i])^2}{\det C[1 \ldots (i-1)]} \Big)^{\frac{n}{2}}. \tag{15}$$

Using Lemma 1, we obtain the desired expression for the running time.

For the case $t = 1$ and $|L_{\text{out}}| = |L|$, the result of Thm. 3 on the size of the input lists $|L|$ yields a compact formula for the filtered lists:

$$\big|L_i^{(j)}\big| = \Big(k^{\frac{1}{k-1}} \cdot \frac{k-j}{k-j+1} \Big)^{\frac{n}{2}}. \tag{16}$$

Plugging this into either (14) or (15), the running time stated in (8) easily follows.

It remains to show the complexity of the BLS algorithm [4], claimed in Rmk. 2. We do not give a complete description of the algorithm but illustrate it in Fig. (2b). We change the presentation of the algorithm to our configuration setting: in the original description, a vector $\boldsymbol{x}_i$ survives the filter if it satisfies $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_1 + \ldots + \boldsymbol{x}_{i-1} \rangle| \ge c_i$ for a predefined c_i (a sequence $(c_1, \ldots, c_{k-1}) \in \mathbb{R}^{k-1}$ is given as input to the BLS algorithm). Our concentration result (Thm. 1) also applies here and the condition $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_1 + \ldots + \boldsymbol{x}_{i-1} \rangle| \ge c_i$ is equivalent to a pairwise constraint on the $\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle$ up to losing an exponentially small fraction of solutions. The optimal sequence of c_i's corresponds to the balanced configuration $C_{\text{Bal},t}$ derived in Thm. 2. Indeed, Table 1 in [4] corresponds exactly to $C_{\text{Bal},t}$ for $t = 1$. So we may rephrase their filtering where instead of shrinking the list L_i by taking inner products with the sum $\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_{i-1}$, we filter L_i gradually by considering $\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle$ for $1 \le j \le i-1$.

It follows that the filtered lists $L^{(i)}$ on level i are of the same size (in leading order) for both our and BLS algorithms. In particular, Eq. (12) holds for the expected list-sizes of the BLS algorithm. The crucial difference lies in the

construction of these lists. To construct the list $L_i^{(i-1)}$ in BLS, the filtering procedure is applied not to $L_i^{(i-2)}$, but to a (larger) input-list L_i. Hence, the running time is (cf. (14)), ignoring subexponential factors

$$\begin{aligned} T_{\mathrm{BLS}} &= |L_1| \cdot (|L_2| + |L_2^{(1)}| \cdot (|L_3| + |L_3^{(2)}| \cdot (\ldots \cdot (|L_k| + |L_k^{(k-1)}|))) \\ &= |L|^2 \cdot \max_{1 \le i \le k-1} \cdot \prod_{j=1}^{i-1} |L^{(j)}|. \end{aligned}$$

The result follows after substituting (16) into the above product.

6 Configuration Extension

For $k = 2$, the asymptotically best algorithm with running time $T = \left(\frac{3}{2}\right)^{\frac{n}{2}}$ for $t = 1$ is due to [5], using techniques from Locally Sensitive Hashing. We generalize this to what we call Configuration Extension. To explain the LSH technique, consider the (equivalent) approximate 2-List problem with $t = 1$, where we want to bound the norm of the difference $\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2 \le 1$ rather than the sum, i.e. we want to find points that are close. The basic idea is to choose a family of hash functions $\mathscr{H}$, such that for $h \in \mathscr{H}$, the probability that $h(\boldsymbol{x}_1) = h(\boldsymbol{x}_2)$ is large if $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are close, and small if they are far apart. Using such an $h \in \mathscr{H}$, we can bucket our lists according to h and then only look for pairs $\boldsymbol{x}_1, \boldsymbol{x}_2$ that collide under h. Repeat with several $h \in \mathscr{H}$ as appropriate to find all/most solutions. We may view such an $h \in \mathscr{H}$ as a collection of preimages $D_{h,z} = h^{-1}(z)$ and the algorithm first determines which elements $\boldsymbol{x}_1, \boldsymbol{x}_2$ are in some given $D_{h,z}$ (filtering the list using $D_{h,z}$) and then searches for solutions only among those. Note that, conceptually, we only really need the $D_{h,z}$ and not the functions h. Indeed, there is actually no need for the $D_{h,z}$ to be a partition of S^n for given h, and h need not even exist. Rather, we may have an arbitrary collection of sets $D^{(r)}$, with r belonging to some index set. The existence of functions h would help in efficiency when filtering. However, [5] (and also [16], stated for the ℓ_1-norm) give a technique to efficiently construct and apply filters $D^{(r)}$ without such an h in an amortized way.

The natural choice for $D^{(r)}$ is to choose all points with distance at most d for some $d > 0$ from some reference point $\boldsymbol{v}^{(r)}$ (that is typically not from any L_i). This way, a random pair $\boldsymbol{x}_1, \boldsymbol{x}_2 \in D^{(r)}$ has a higher chance to be close to each other than uniformly random points $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathsf{S}^n$. Notationally, let us call (a description of) $D^{(r)}$ together with the filtered lists an *instance*, where $1 \le r \le R$ and R is the number of instances.

In our situation, we look for small sums rather than small differences. The above translates to asking that $\boldsymbol{x}_1$ is close to $\boldsymbol{v}^{(r)}$ and that $\boldsymbol{x}_2$ is far apart from $\boldsymbol{v}^{(r)}$ (or, equivalently, that $\boldsymbol{x}_2$ is close to $-\boldsymbol{v}^{(r)}$). In general, one may (for $k > 2$) consider not just a single $\boldsymbol{v}^{(r)}$ but rather several *related* $\boldsymbol{v}_1^{(r)}, \ldots, \boldsymbol{v}_m^{(r)}$. So an instance consists of m points $\boldsymbol{v}_1^{(r)}, \ldots, \boldsymbol{v}_m^{(r)}$ and shrunk lists $L_i'^{(r)}$ where $L_i'^{(r)} \subset L_i$ is obtained by taking those $x_i \in L_i$ that have some prescribed distances $d_{i,j}$

to $\boldsymbol{v}_j^{(r)}$. Note that the $d_{i,j}$ may depend on i and so need not treat the lists symmetrically. As a consequence, it does no longer make sense to think of this technique in terms of hash collisions in our setting.

We organize all the distances between $\boldsymbol{v}$'s and $\boldsymbol{x}$'s that occur into a single matrix C (i.e. a configuration) that governs the distances between $\boldsymbol{v}$'s and $\boldsymbol{x}$'s: the $\langle \boldsymbol{v}_j, \boldsymbol{v}_{j'} \rangle$-entries of C describe the relation between the $\boldsymbol{v}$'s and the $\langle \boldsymbol{x}_i, \boldsymbol{v}_j \rangle$-entries of C describe the $d_{i,j}$. The $\langle \boldsymbol{x}_i, \boldsymbol{x}_{i'} \rangle$-entries come from the approximate k-List problem we want to solve. While not relevant for constructing actual $\boldsymbol{v}_j^{(r)}$'s and $L_i'^{(r)}$'s, the $\langle \boldsymbol{x}_i, \boldsymbol{x}_{i'} \rangle$-entries are needed to choose the number R of instances.

For our applications to sieving, the elements from the input list L_i may possibly be not uniform from all of S^n due to previous processing of the lists. Rather, the elements $\boldsymbol{x}_i$ from L_i have some prescribed distance $d_{i,j}$ to (known) $\boldsymbol{v}_j$'s: e.g. in Alg. 1, we fix $\boldsymbol{x}_1 \in L_1$ that we use to filter the remaining $k-1$ lists; we model this by taking $\boldsymbol{x}_1$ as one of the $\boldsymbol{v}_j$'s (and reducing k by 1). Another possibility is that we use configuration extension on lists that are the output of a previous application of configuration extension.

In general, we consider "old" points $\boldsymbol{v}_j$ and wish to create "new" points $\boldsymbol{v}_\ell$, so we have actually three different types of rows/columns in C, corresponding to the list elements, old and new points.

Definition 4 (Configuration Extension). *Consider a configuration matrix C. We consider C as being indexed by disjoint sets $I_{\text{lists}}, I_{\text{old}}, I_{\text{new}}$. Here, $|I_{\text{lists}}| = k$ corresponds to the input lists, $|I_{\text{old}}| = m_{\text{old}}$ corresponds to the "old" points, $|I_{\text{new}}| = m_{\text{new}}$ corresponds to the "new" points. We denote appropriate square submatrices by $C[I_{\text{lists}}]$ etc. By configuration extension, we mean an algorithm* $\mathsf{ConfExt}$ *that takes as input k exponentially large lists $L_i \subset \mathsf{S}^n$ for $i \in I_{\text{lists}}$, m_{old} "old" points $\boldsymbol{v}_j \in \mathsf{S}^n$, $j \in I_{\text{old}}$ and the matrix C. Assume that each input list separately satisfies the given configuration constraints wrt. the old points:* $\mathrm{Conf}(\boldsymbol{x}_i, (\boldsymbol{v}_j)_{j \in I_{\text{old}}}) \approx C[i, I_{\text{old}}]$ *for $i \in I_{\text{lists}}$, $\boldsymbol{x}_i \in L_i$.*

It outputs R instances, where each instance consists of m_{new} points $\boldsymbol{v}_\ell$, $\ell \in I_{\text{new}}$ and shrunk lists $L_i' \subset L_i$, where $\mathrm{Conf}((\boldsymbol{v}_j)_{j \in I_{\text{old}}}, (\boldsymbol{v}_\ell)_{\ell \in I_{\text{new}}}) \approx C[I_{\text{old}}, I_{\text{new}}]$ *and each $\boldsymbol{x}_i' \in L_i'$ satisfies*

$$\mathrm{Conf}(\boldsymbol{x}_i', (\boldsymbol{v}_j)_{j \in I_{\text{old}}}, (\boldsymbol{v}_\ell)_{\ell \in I_{\text{old}}}) \approx C[i, I_{\text{old}}, I_{\text{new}}].$$

The instances are output one-by-one in a streaming fashion. This is important, since the total size of the output usually exceeds the amount of available memory.

The naive way to implement configuration extension is as follows: independently for each instance, sample uniform $\boldsymbol{v}_\ell$'s conditioned on the given constraints and then make a single pass over each input list L_i to construct L_i'. This would require $\widetilde{\mathcal{O}}(\max_i |L_i| \cdot R)$ time. However, using the block coding / stripe techniques of [5,16], one can do much better. The central observation is that if we subdivide the coordinates into blocks, then a configuration constraint on all coordinates is (up to losing a subexponential fraction of solutions) equivalent to independent configuration constraints on each block. The basic idea is then to

construct the $\boldsymbol{v}_\ell$'s in a block-wise fashion such that an exponential number of instances have the same $\boldsymbol{v}_\ell$'s on a block of coordinates. We can then amortize the construction of the L_i''s among such instances, since we can first construct some intermediate $L_i'' \subset L_i$ that is compatible with the $\boldsymbol{v}_\ell$'s on the shared block of coordinates. To actually construct $L_i' \subset L_i''$, we only need to pass over L_i'' rather than L_i. Of course, this foregos independence of the $\boldsymbol{v}_\ell$'s across different instances, but one can show that they are still independent enough to ensure that we will find most solutions if the number of instances is large enough.

Adapting these techniques of [5,16] to our framework is straightforward, but extremely technical. We work out the details in the full version of the paper.

A rough summary of the properties of our Configuration Extension Algorithm ConfExt (see the full version for a proof) is given by the following:

Theorem 5. *Use notation as in Def. 4. Assume that $C, k, m_{\text{old}}, m_{\text{new}}$ do not depend on n. Then our algorithm* ConfExt*, given as input $C, k, m_{\text{old}}, m_{\text{new}}$, old points $\boldsymbol{v}_j$ and exponentially large lists $L_1, \ldots, L_k$ of points from S^n, outputs*

$$R = \widetilde{\mathcal{O}}\left(\frac{\det(C[I_{\text{old}}, I_{\text{new}}]) \cdot \det(C[I_{\text{lists}}, I_{\text{old}}])}{\det(C[I_{\text{lists}}, I_{\text{old}}, I_{\text{new}}]) \cdot \det(C[I_{\text{old}}])}\right)^{\frac{n}{2}} \tag{17}$$

instances, where each output instance consists of m_{new} points $\boldsymbol{v}_\ell$ and sublists $L_i' \subset L_i$. In each such output instance, the new points $(\boldsymbol{v}_\ell)_{\ell \in I_{\text{new}}}$ are chosen uniformly conditioned on the constraints (but not independent across instances). Consider solution k-tuples, i.e. $\boldsymbol{x}_i \in L_i$ with $\text{Conf}((\boldsymbol{x}_i)_{i \in I_{\text{lists}}}) \approx C[I_{\text{lists}}]$. With overwhelming probability, for every solution k-tuple $(\boldsymbol{x}_i)_{i \in I_{\text{lists}}}$, there exists at least one instance such that all $\boldsymbol{x}_i \in L_i'$ for this instance, so we retain all solutions. Assume further that the elements from the input lists L_i, $i \in I_{\text{lists}}$ are iid uniformly distributed conditioned on the configuration $\text{Conf}(\boldsymbol{x}_i, (\boldsymbol{v}_j)_{j \in I_{\text{old}}})$ for $\boldsymbol{x}_i \in L_i$, which is assumed to be compatible with C. Then the expected size of the output lists per instance is given by

$$\mathbb{E}[|L_i'|] = |L_i| \cdot \widetilde{\mathcal{O}}\left(\left(\frac{\det(C[i, I_{\text{old}}, I_{\text{new}}]) \cdot \det(C[I_{\text{old}}])}{\det(C[I_{\text{old}}, I_{\text{new}}]) \cdot \det(C[i, I_{\text{old}}])}\right)^{n/2}\right).$$

Assume that all these expected output list sizes are exponentially increasing in n (rather than decreasing). Then the running time of the algorithm is given by $\widetilde{\mathcal{O}}(R \cdot \max_i \mathbb{E}[|L_i'|])$ (essentially the size of the output) and the memory complexity is given by $\widetilde{\mathcal{O}}(\max_i |L_i|)$ (essentially the size of the input).

7 Improved k-List Algorithm with Configuration Extension

Now we explain how to use the Configuration Extension Algorithmwithin the k-List algorithm Alg. 1 to sp eed-up the search for configurations. In fact, there is a whole family of algorithms obtained by combining Filter from Alg. 1 and the

configuration extension algorithm ConfExt. The combined algorithm is given in Alg. 2.

Recall that Alg. 1 takes as inputs k lists $L_1, \ldots, L_k$ of equal size and processes the lists in several levels (cf. Fig. 2a). The lists $L_j^{(i)}$ for $j \geq i$ at the i^{th} level (where the input lists correspond to the 0^{th} level) are obtained by brute-forcing over $\boldsymbol{x}_i \in L_i^{(i-1)}$ and running Filter on $L_j^{(i-1)}$ and $\boldsymbol{x}_i$.

We can use ConfExt in the following way: before using Filter on $L_j^{(i-1)}$, we run ConfExt to create R instances with smaller sublists $L_j'^{(i-1)} \subset L_j^{(i-1)}$. We then apply Filter to each of these $L_j'^{(i-1)}$ rather than to $L_j^{(i-1)}$. The advantage is that for a given instance, the $L_j'^{(i-1)}$ are dependent (over the choice of j), so we expect a higher chance to find solutions.

In principle, one can use ConfExt on any level, i.e. we alternate between using ConfExt and Filter. Note that the $\boldsymbol{x}_i$'s that we brute-force over in order to apply Filter become "old" $\boldsymbol{v}_j$'s in the context of the following applications of ConfExt.

It turns out that among the variety of potential combinations of Filter and ConfExt, some are more promising than others. From the analysis of Alg. 1, we know that the running time is dominated by the cost of filtering (appropriately multiplied by the number of times we need to filter) to create lists at some level $i_{\max}$. The value of $i_{\max}$ can be deduced from Eq. (14), where the individual contribution $|L| \cdot |L^{(i-1)}| \cdot \prod_{j=1}^{i-1} |L^{(j)}|$ in that formula exactly corresponds to the total cost of creating all lists at the i-th level.

It makes sense to use ConfExt to reduce the cost of filtering at this critical level. This means that we use ConfExt on the lists $L_j^{(i_{\max}-1)}$, $j \geq i_{\max} - 1$. Let us choose $m_{\text{new}} = 1$ new point $\boldsymbol{v}_\ell$. The lists $L_j^{(i_{\max}-1)}$ are already reduced by enforcing configuration constraints with $\boldsymbol{x}_1 \in L_1, \ldots, \boldsymbol{x}_{i_{\max}-1} \in L_{i_{\max}-1}$ from previous applications of Filter. This means that the $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i_{\max}-1}$ take the role of "old" $\boldsymbol{v}_j$'s in ConfExt. The configuration $C^{ext} \in \mathbb{R}^{(k+1)\times(k+1)}$ for ConfExt is obtained as follows: The $C^{ext}[I_{\text{lists}}, I_{\text{old}}]$-part is given by the target configuration. The rest (which means the last row/column corresponding to the single "new" point) can be chosen freely and is subject to optimization. Note that the optimization problem does not depend on n.

This approach is taken in Alg. 2. Note that for levels below $i_{\max}$, it does not matter whether we continue to use our Filter approach or just brute-force: if $i_{\max} = k$, there are no levels below. If $i_{\max} < k$, the lists are small from this level downward and brute-force becomes cheap enough not to affect the asymptotics.

Let us focus on the case where the input list sizes are the same as the output list sizes, which is the relevant case for applications to Shortest Vector sieving. It turns out (numerically) that in this case, the approach taken by Alg. 2 is optimal for most values of k. The reason is as follows: Let T be the contribution to the running time of Alg. 1 from level $i_{\max}$, which is asymptotically the same as the total running time. The second-largest contribution, denoted T' comes from level $i_{\max} - 1$. The improvement in running time from using ConfExt to reduce T decreases with k and is typically not enough to push it below T'. Consequently,

using ConfExt between other levels will not help. We also observed that choosing $m_{\text{new}} = 1$ was usually optimal for k up to 10. Exceptions to these observations occur when T and T' are very close (this happens, e.g. for $k = 6$) or when k is small and the benefit from using ConfExt is large (i.e. $k = 3$).

Since the case $k = 3$ is particularly interesting for the Shortest Vector sieving (see Sect. 7.2), we present the 3-List algorithm separately in Sect. 7.1.

Algorithm 2 k-List with Configuration Extension

Input: $L_1, \ldots, L_k$ – input lists. $C \in \mathbb{R}^{k \times k}$ – target configuration. $\varepsilon > 0$ – measure of closeness.
Output: L_{out} – list of k-tuples $\boldsymbol{x}_1 \in L_1, \ldots, \boldsymbol{x}_k \in L_k$, s.t. $|\langle \boldsymbol{x}_i , \boldsymbol{x}_j \rangle - C_{ij}| \leq \varepsilon$, for all i, j.

$i_{\max}, C^{ext} = \textsc{Preprocess}(k, C_{i,j} \in \mathbb{R}^{k \times k})$
$L_{\text{out}} \leftarrow \{\}$
for all $\boldsymbol{x}_1 \in L_1$ **do**
  $L_j^{(1)} \leftarrow \textsc{Filter}(\boldsymbol{x}_1, L_j, C_{1,j}, \varepsilon)$ ▷ $j = 2, \ldots, k$
$\ddots$
  for all $\boldsymbol{x}_{i_{\max}-1} \in L_{i_{\max}-1}^{(i_{\max}-2)}$ **do**
    $L_j^{(i_{\max}-1)} \leftarrow \textsc{Filter}(\boldsymbol{x}_{i_{\max}-1}, L_j^{i_{\max}-2}, C_{i_{\max}-1,j}, \varepsilon)$ ▷ $j = i_{\max}, \ldots, k$
    $I_{\text{old}} \leftarrow \{1, \ldots, i_{\max} - 1\}$, $I_{\text{lists}} \leftarrow \{i_{\max}, \ldots, k\}$, $I_{\text{new}} \leftarrow \{k + 1\}$.
    $m_{\text{old}} \leftarrow i_{\max} - 1, k' \leftarrow k + 1 - i_{\max}, m_{\text{new}} \leftarrow 1$.
    $\boldsymbol{v}_j \leftarrow \boldsymbol{x}_j$ for $j \in I_{\text{old}}$.
    Call ConfExt$(n, k', m_{\text{old}}, m_{\text{new}}, C^{ext}, L_{i_{\max}}^{(i_{\max}-1)}, \ldots, L_k^{(i_{\max}-1)}, (\boldsymbol{v}_j)_{j \in I_{\text{old}}}, \varepsilon)$
    for all output instances $\boldsymbol{w}$, $L_{i_{\max}}^{\prime(i_{\max}-1)}, \ldots, L_k^{\prime(i_{\max}-1)}$ **do** ▷ Output is streamed
      for all $\boldsymbol{x}_{i_{\max}} \in L_j^{\prime(i_{\max}-1)}$ **do**
        $L_j^{(i_{\max})} \leftarrow \textsc{Filter}(\boldsymbol{x}_{i_{\max}}, L_j^{\prime(i_{\max}-1)}, C_{i_{\max},j}, \varepsilon)$ ▷ $j = i_{\max} + 1 \ldots k$
      Brute-force over $L_j^{(i_{\max})}$ to obtain $\boldsymbol{x}_{i_{\max}+1}, \ldots, \boldsymbol{x}_k$ compatible with C
      $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)\}$
return L_{out}

procedure $\textsc{Preprocess}(k,\ C \in \mathbb{R}^{k \times k})$
  Determine $i_{\max}$ using Eq. (14)
  Set $C^{ext}[\{1, \ldots, k\}] \leftarrow C$.
  Determine optimal $C^{ext}_{i,k+1} = C^{ext}_{k+1,i}$ by numerical optimization.
  return $i_{\max}$, $C^{ext} \in \mathbb{R}^{(k+1)\times(k+1)}$

function $\textsc{Filter}(\boldsymbol{x}, L, c, \varepsilon)$: See Algorithm 1

7.1 Improved 3-List Algorithm

The case $k = 3$ stands out from the above discussion as one can achieve a faster algorithm running the Configuration Extension Algorithm on *two* points $\boldsymbol{v}_1, \boldsymbol{v}_2$. This case is also interesting in applications to lattice sieving, so we detail on it below.

From Eq. (14) we have $i_{\max} = 2$, or more precisely, the running time of the 3-List algorithm (without Configuration Extension) is $T = |L_1| \cdot |L_2^{(1)}| \cdot |L_3^{(1)}|$. So we start shrinking the lists right from the beginning which corresponds to $m_{\text{old}} = 0$. For the balance configuration as the target, we have $C[I_{\text{lists}}] = -1/3$ on the off-diagonals. With the help of an optimization solver, we obtain the optimal values for $\langle \boldsymbol{x}_i \, , \boldsymbol{v}_j \rangle$ for $i = \{1, 2, 3\}$ and $j = \{1, 2\}$, and for $\langle \boldsymbol{v}_1 \, , \boldsymbol{v}_2 \rangle$ (there are 7 values to optimize for), so the input to the Configuration Extension Algorithm is determined. The target configuration is of the form

$$C = \begin{pmatrix} 1 & -1/3 & -1/3 & 0.47 & -0.15 \\ -1/3 & 1 & -1/3 & -0.17 & 0.26 \\ -1/3 & -1/3 & 1 & -0.19 & -0.14 \\ 0.47 & -0.17 & -0.19 & 1 & -0.26 \\ -0.15 & 0.26 & -0.14 & -0.26 & 1 \end{pmatrix} \tag{18}$$

and the number of instances is given by $R = \widetilde{\mathcal{O}}(1.4038^n)$ according to (17). The algorithm runs in a streamed fashion: the lists L_1', L_2', L_3' in line 2 of Alg. 3 are obtained instance by instance and, hence, lines 3 to 9 are repeated R times.

Algorithm 3 3-List with Configuration Extension

Input: L_1, L_2, L_3 – input lists of vectors from S^n, $|L| = 2^{0.1887n+o(n)}$
$C \in \mathbb{R}^{5\times 5}$ as in Eq. (18), $\rho = 1.4038$, $\varepsilon > 0$
Output: $L_{\text{out}} \subset L_1 \times L_2 \times L_3$, s.t. $|\langle \boldsymbol{x}_i \, , \boldsymbol{x}_j \rangle - C_{ij}| \leq \varepsilon$, for all $1 \leq i, j \leq 3$.

1: $L_{\text{out}} \leftarrow \{\}$
2: $L_1', L_2', L_3' \leftarrow \mathsf{ConfExt}(k = 3, m_{\text{old}} = 0, m_{\text{new}} = 2, C \in \mathbb{R}^{5\times 5}, \varepsilon, L_1, L_2, L_3, (n_1, \ldots, n_t))$
3: **for all** $\boldsymbol{x}_1 \in L_1'$ **do**
4: $L_2^{(1)} \leftarrow \textsc{Filter}(\boldsymbol{x}_1, L_2', -1/3, \varepsilon)$
5: $L_3^{(1)} \leftarrow \textsc{Filter}(\boldsymbol{x}_1, L_3', -1/3, \varepsilon)$
6: **for all** $\boldsymbol{x}_2 \in L_2^{(1)}$ **do**
7: **for all** $\boldsymbol{x}_3 \in L_3^{(1)}$ **do**
8: **if** $|\langle \boldsymbol{x}_2 \, , \boldsymbol{x}_3 \rangle + 1/3| \leq \varepsilon$ **then**
9: $L_{\text{out}} \leftarrow (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)$
10: **return** L_{out}
1: **function** $\textsc{Filter}(\boldsymbol{x}, L, c, \varepsilon)$: See Algorithm 1

From Thm. 3, it follows that if the input lists satisfy $|L| = 2^{0.1887n+o(n)}$, then we expect $|L_{\text{out}}| = |L|$. Also from Eq. (8), it follows that the 3-List Algorithm 1 (i.e. without combining with the Configuration Extension Algorithm) has running time of $2^{0.3962n+o(n)}$. The above Alg. 3 brings it down to $2^{0.3717n+o(n)}$.

7.2 Application to the Shortest Vector Problem

In this section we briefly discuss how certain shortest vector algorithms can benefit from our improvement for the approximate k-List problem. We start by stating the approximate shortest vector problem.

On input, we are given a full-rank lattice $\mathcal{L}(B)$ described by a matrix $B \in \mathbb{R}^{n \times n}$ (with polynomially-sized entries) whose columns correspond to basis vectors, and some constant $\mathsf{c} \geq 1$. The task is to output a nonzero lattice vector $\boldsymbol{x} \in \mathcal{L}(B)$, s.t. $\|\boldsymbol{x}\| \leq \mathsf{c}\lambda_1(B)$ where $\lambda_1(B)$ denotes the length of the shortest nonzero vector in $\mathcal{L}(B)$. $\boldsymbol{x}$ is a solution to the approximate shortest vector problem.

The AKS sieving algorithm (introduced by Ajtai, Kumar, and Sivakumar in [1]) is currently the best (heuristic) algorithm for the approximate shortest vector problem: for an n-dimensional lattice, the running time and memory are of order 2^n. Sieving algorithms have two flavours: the Nguyen-Vidick sieve [18] and the Gauss sieve [17]. Both make polynomial in n number of calls to the approximate 2-List solver. Without LSH-techniques, the running time both the Nguyen-Vidick and the Gauss sieve is the running time of the approximate 2-List algorithm: $2^{0.415n+o(n)}$ with $2^{0.208n+o(n)}$ memory. Using our 3-List Algorithm 1 instead, the running time can be reduced to $2^{0.3962n+o(n)}$ (with only $2^{0.1887n+o(n)}$ memory) introducing essentially no polynomial overhead. Using Algorithm 3, we achieve even better asymptotics: $2^{0.3717n+o(n)}$, but it might be too involved for practical speed-ups due very large polynomial overhead for too little exponential gain in realistic dimensions.

Now we describe the Nguyen-Vidick sieve that uses a k-List solver as a main subroutine (see [4] for a more formal description). We start by sampling lattice-vectors $\boldsymbol{x} \in \mathcal{L}(B) \cap \mathsf{B}_n(2^{O(n)} \cdot \lambda_1(B))$, where $\mathsf{B}_n(R)$ denotes an n-dimensional ball of radius R. This can be done using, for example, Klein's nearest plane procedure [11]. In the k-List Nguyen-Vidick for $k > 2$, we sample many such lattice-vectors, put them in a list L, and search for k-tuples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in L \times \ldots \times L$ s.t. $\|\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_k\| \leq \gamma \cdot \max_{1 \leq i \leq k} \boldsymbol{x}_i$ for some $\gamma < 1$. The sum $\boldsymbol{x}_1 + \ldots + \boldsymbol{x}_k$ is put into L_{out}. The size of L is chosen in a way to guarantee that $|L| \approx |L_{\text{out}}|$. The search for short k-tuples is repeated over the list L_{out}. Note that since with each new iteration we obtain vectors that are shorter by a constant factor γ, starting with $2^{\mathcal{O}(n)}$ approximation to the shortest vector (this property is guaranteed by Klein's sampling algorithm applied to an LLL-reduced basis), we need only linear in n iterations to find the desired $\boldsymbol{x} \in \mathcal{L}(B)$.

Naturally, we would like to apply our approximate k-List algorithm to k copies of the list L to implement the search for short sums. Indeed, we can do so by making a commonly used assumption: we assume the lattice-vectors we put into the lists lie uniformly on a spherical shell (on a very thin shell, essentially a sphere). The heuristic here is that it does not affect the behaviour of the algorithm. Intuitively, the discreteness of a lattice should not be "visible" to the algorithm (at least not until we find the approximate shortest vector).

We conclude by noting that our improved k-List Algorithm can as well be used within the Gauss sieve, which is known to perform faster in practice than

the Nguyen-Vidick sieve. An iteration of the original 2-Gauss sieve as described in [17], searches for pairs $(\boldsymbol{p}, \boldsymbol{v})$, s.t. $\|\boldsymbol{p}+\boldsymbol{v}\| < \max\{\|\boldsymbol{p}\|, \|\boldsymbol{v}\|\}$, where $\boldsymbol{p} \in \mathcal{L}(B)$ is *fixed*, $\boldsymbol{v} \in L \subset \mathcal{L}(B)$, and $\boldsymbol{p} \neq \boldsymbol{v}$. Once such a pair is found and $\|\boldsymbol{p}\| > \|\boldsymbol{v}\|$, we set $\boldsymbol{p}' \leftarrow \boldsymbol{p}+\boldsymbol{v}$ and proceed with the search over $(\boldsymbol{p}', \boldsymbol{v})$, otherwise if $\|\boldsymbol{p}\| < \|\boldsymbol{v}\|$, we delete $\boldsymbol{v} \in L$ and store the sum $\boldsymbol{p}+\boldsymbol{v}$ as $\boldsymbol{p}$-input point for the next iteration. Once no pair is found, we add $\boldsymbol{p}'$ to L. On the next iteration, the search is repeated with another $\boldsymbol{p}$ which is obtained either by reducing some deleted $\boldsymbol{v} \in L$ before, or by sampling from $\mathcal{L}(B)$. The idea is to keep only those vectors in L that *cannot* form a pair with a shorter sum. Bai, Laarhoven, and Stehlé in [4], generalize it to k-Gauss sieve by keeping only those vectors in L that do not form a shorter k-sum. In the language of configuration search, we look for configurations $(\boldsymbol{p}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{k-1}) \in \{\boldsymbol{p}\} \times L \times \ldots \times L$ where the first point is fixed, so we apply our Alg. 1 on $k-1$ (identical) lists.

Unfortunately, applying LSH / configuration extension-techniques for the Gauss Sieve is much more involved than for the Nguyen-Vidick Sieve. For $k = 2$, [13] applies LSH techniques, but this requires an exponential increase in memory (which runs counter to our goal). We do not know whether these techniques extend to our setting. At any rate, since the gain from LSH / Configuration Extension techniques decreases with k (with the biggest jump from $k = 2$ to $k = 3$, while the overhead increases, gaining a practical speed-up from LSH / Configuration Extension within the Gauss sieve for $k \geq 3$ seems unrealistic.

Open questions. We present all our algorithms for a *fixed* k, and in the analysis, we suppress all the prefactors (in running time and list-sizes) for fixed k in the $\mathcal{O}_k(.)$ notation. Taking a closer look at how these factors depend on k, we notice (see, for example, the expression for $W_{n,k}$ in Thm. 1) that exponents of the polynomial prefactors depend on k. It prevents us from discussing the case $k \to \infty$, which is an interesting question especially in light of SVP. Another similar question is the optimal choice of ε and how it affects the pre-factors.

8 Experimental results

We implement the 3-Gauss sieve algorithm in collaboration with S. Bai [3]. The implementation is based on the program developed by Bai, Laarhoven, and Stehlé in [4], making the approaches comparable.

Lattice bases are generated by the SVP challenge generator [7]. It produces a lattice generated by the columns of the matrix

$$B = \begin{pmatrix} p & x_1 & \ldots & x_{n-1} \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix},$$

where p is a large prime, and $x_i < p$ for all i. Lattices of this type are random in the sense of Goldstein and Mayer [9].

For all the dimensions except 80, the bases are preprocessed with BKZ reduction of block-size 20. For $n = 80$, the block-size is 30. For our input lattices, we

	2-sieve	BLS 3-sieve	Alg. 1 for $k = 3$			
			$\varepsilon = 0.0$	$\varepsilon = 0.015$	$\varepsilon = 0.3$	$\varepsilon = 0.4$
n	T, $\lvert L\rvert$	T, $\lvert L\rvert$	T, $\lvert L\rvert$	T, $\lvert L\rvert$	T, $\lvert L\rvert$	T, $\lvert L\rvert$
60	1.38e3, 13257	1.02e4, 4936	1.32e3, 7763	1.26e3, 7386	1.26e3, 6751	**1.08e3, 6296**
62	2.88e3, 19193	1.62e4, 6239	2.8e3, 10356	3.1e3, 9386	**1.8e3, 8583**	2.2e3, 8436
64	8.64e3, 24178	5.5e4, 8369	5.7e3, 13573	3.6e3, 12369	**3.36e3, 11142**	4.0e4, 10934
66	1.75e4, 31707	9.66e4, 10853	1.5e4, 17810	1.38e4, 16039	**9.1e3, 14822**	1.2e4, 14428
68	3.95e4, 43160	2.3e5, 14270	2.34e4, 24135	2.0e4, 21327	**1.68e4, 19640**	1.86e4, 18355
70	6.4e4, 58083	6.2e5, 19484	6.21e4, 32168	3.48e5, 26954	**3.3e4, 25307**	3.42e4, 24420
72	2.67e5, 77984	1.2e6, 25034	7.6e4, 40671	7.2e4, 37091	**6.16e4, 34063**	6.35e4, 34032
74	3.45e5, 106654	—	2.28e5, 54198	2.08e5, 47951	**2.02e5, 43661**	2.03e5, 40882
76	4.67e5, 142397	—	3.58e5, 71431	2.92e5, 64620	**2.42e5, 56587**	2.53e5, 54848
78	9.3e5, 188905	—	—	—	**4.6e5, 74610**	4.8e5, 70494
80	—	—	—	—	**9.47e5, 98169**	9.9e5, 98094

Table 1: Experimental results for k-tuple Gauss sieve. The running times T are given in seconds, $|L|$ is the maximal size of the list L. ε is the approximation parameter for the subroutine Filter of Alg. 1. The best running-time per dimension is type-set bold.

do not know their minimum λ_1. The algorithm terminates when it finds many linearly dependent triples $(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3)$. We set a counter for such an event and terminate the algorithm once this counter goes over a pre-defined threshold. The intuition behind this idea is straightforward: at some point the list L will contain very short basis-vectors and the remaining list-vectors will be their linear combinations. Trying to reduced the latter will ultimately produce the zero-vector. The same termination condition was already used in [15], where the authors experimentally determine a threshold of such "zero-sum" triples.

Up to $n = 64$, the experiments are repeated 5 times (i.e. on 5 random lattices), for the dimensions less than 80, 3 times. For the running times and the list-sizes presented in the table below, the average is taken. For $n = 80$, the experiment was performed once.

Our tests confirm a noticeable speed-up of the 3-Gauss sieve when our Configuration Search Algorithm 1 is used. Moreover, as the analysis suggests (see Fig. 3), our algorithm outperforms the naive 2-Gauss sieve while using much less memory. The results can be found in Table 1.

Another interesting aspect of the algorithm is the list-sizes when compared with BLS. Despite the fact that, asymptotically, the size of the list $|L|$ is the same for our and for the BLS algorithms, in practice our algorithm requires a longer list (cf. the right numbers in each column). This is due to the fact that we filter out a larger fraction of solutions. Also notice that increasing ε – the approximation to the target configuration, we achieve an additional speed-up. This becomes obvious once we look at the Filter procedure: allowing for a smaller inner-product throws away less vectors, which in turn results in a shorter list L. For the range of dimensions we consider, we experimentally found $\varepsilon = 0.3$ to be a good choice.

Acknowledgments. We would like to thank the authors of [4], Shi Bai, Damien Stehlé, and Thijs Laarhoven for constructive discussions.

Elena Kirshanova was supported by UbiCrypt, the research training group 1817/1 funded by the DFG. Gottfried Herold was funded by ERC grant 307952 (acronym FSC).

References

1. M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of STOC*, pages 601–610, 2001.
2. A. G. Akritas, E. K. Akritas, and G. I. Malaschonok. Sybolic computation, new trends and developments various proofs of sylvester's (determinant) identity. *Mathematics and Computers in Simulation*, 42(4):585 – 593, 1996.
3. S. Bai, August 2016. Personal Communication.
4. S. Bai, T. Laarhoven, and D. Stehlé. Tuple Lattice Sieving. In *ANTS-XII (to appear)*, 2016.
5. A. Becker, L. Ducas, N. Gama, and T. Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In R. Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 10–24. SIAM, 2016.
6. A. Blum, A. Kalai, and H. Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, pages 506–519, 2003.
7. S. Challenge. SVP challenge generator. `http://latticechallenge.org/svp-challenge`.
8. M. Ghosh and B. K. Sinha. A simple derivation of the wishart distribution. *The American Statistician*, 56(2):100–101, 2002.
9. D. Goldstein and A. Mayer. On the equidistribution of hecke points. *Forum Mathematicum*, 15(3):165–189, 01 2006.
10. R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of STOC*, pages 193–206, 1983.
11. P. Klein. Finding the closest lattice vector when it's unusually close. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 2000, pages 937–941, 2000.
12. O. Kupferman, Y. Lustig, and M. Y. Vardi. *On Locally Checkable Properties*, pages 302–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

13. T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 3–22, Santa Barbara, CA, USA, Aug. 16–20, 2015. Springer, Heidelberg, Germany.
14. V. Lyubashevsky. On random high density subset sums. In *APPROX-RANDOM, LNCS 3624*, pages 378–389, 2005.
15. A. Mariano, T. Laarhoven, and C. Bischof. Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the svp. In *44th International Conference on Parallel Processing (ICPP)*, pages 590–599, September 2015.
16. A. May and I. Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 203–228, Sofia, Bulgaria, Apr. 26–30, 2015. Springer, Heidelberg, Germany.
17. D. Micciancio and P. 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.
18. P. Q. Nguyen and T. Vidick. Sieve algorithms for the shortest vector problem are practical. In *Journal of Mathematical Cryptology*, volume 2, pages 181–207, 2008.
19. D. Wagner. Generalized birthday problem. In *Proceedings of CRYPTO 2002, LNCS 2442*, pages 288–304, 2002.
20. J. Wishart. The generalized product moment distribution in samples from a normal multivariate population. *Biometrika*, 20A(1-2):32–52, 1928.