Quantum Algorithms for the Approximate *k*-List Problem and their Application to Lattice Sieving

Elena Kirshanova[1], Erik Mårtensson[2], Eamonn W. Postlethwaite[3], and Subhayan Roy Moulik[4]

[1] I. Kant Baltic Federal University, Kaliningrad, Russia.
`elenakirshanova@gmail.com`

[2] Department of Electrical and Information Technology, Lund University, Sweden.
`erik.martensson@eit.lth.se`

[3] Information Security Group, Royal Holloway, University of London.
`eamonn.postlethwaite.2016@rhul.ac.uk`

[4] Department of Computer Science, University of Oxford, Oxford, UK.
`subhayan.roy.moulik@cs.ox.ac.uk`

Abstract. The Shortest Vector Problem (SVP) is one of the mathematical foundations of lattice based cryptography. Lattice sieve algorithms are amongst the foremost methods of solving SVP. The asymptotically fastest known classical and quantum sieves solve SVP in a d-dimensional lattice in $2^{cd+o(d)}$ time steps with $2^{c'd+o(d)}$ memory for constants c, c'. In this work, we give various quantum sieving algorithms that trade computational steps for memory.

We first give a quantum analogue of the classical k-Sieve algorithm [Herold–Kirshanova–Laarhoven, PKC'18] in the Quantum Random Access Memory (QRAM) model, achieving an algorithm that heuristically solves SVP in $2^{0.2989d+o(d)}$ time steps using $2^{0.1395d+o(d)}$ memory. This should be compared to the state-of-the-art algorithm [Laarhoven, Ph.D Thesis, 2015] which, in the same model, solves SVP in $2^{0.2653d+o(d)}$ time steps and memory. In the QRAM model these algorithms can be implemented using poly(d) width quantum circuits.

Secondly, we frame the k-Sieve as the problem of k-clique listing in a graph and apply quantum k-clique finding techniques to the k-Sieve.

Finally, we explore the large quantum memory regime by adapting parallel quantum search [Beals et al., Proc. Roy. Soc. A'13] to the 2-Sieve and giving an analysis in the quantum circuit model. We show how to heuristically solve SVP in $2^{0.1037d+o(d)}$ time steps using $2^{0.2075d+o(d)}$ quantum memory.

Keywords:
shortest vector problem (SVP), lattice sieving, Grover's algorithm, approximate k-list problem, nearest neighbour algorithms, distributed computation.

1 Introduction

The Shortest Vector Problem (SVP) is one of the central problems in the theory of lattices. For a given d-dimensional Euclidean lattice, usually described by a basis, to solve SVP one must find a shortest non zero vector in the lattice. This problem gives rise to a variety of efficient, versatile, and (believed to be) quantum resistant cryptographic constructions [AD97,Reg05]. To obtain an estimate for the security of these constructions it is important to understand the complexities of the fastest known algorithms for SVP.

There are two main families of algorithms for SVP, (1) algorithms that require $2^{\omega(d)}$ time and $\mathrm{poly}(d)$ memory; and (2) algorithms that require $2^{\Theta(d)}$ time and memory. The first family includes lattice enumeration algorithms [Kan83,GNR10]. The second contains sieving algorithms [AKS01,NV08,MV10], Voronoi cell based approaches [MV10] and others [ADRSD15,BGJ14]. In practice, it is only enumeration and sieving algorithms that are currently competitive in large dimensions [ADH$^+$19,TKH18]. Practical variants of these algorithms rely on *heuristic* assumptions. For example we may not have a guarantee that the returned vector will solve SVP exactly (e.g. pruning techniques for enumeration [GNR10], lifting techniques for sieving [Duc18]), or that our algorithm will work as expected on arbitrary lattices (e.g. sieving algorithms may fail on orthogonal lattices). Yet these heuristics are natural for lattices often used in cryptographic constructions, and one does not require an exact solution to SVP to progress with cryptanalysis [ADH$^+$19]. Therefore, one usually relies on heuristic variants of SVP solvers for security estimates.

Among the various attractive features of lattice based cryptography is its potential resistance to attacks by quantum computers. In particular, there is no known quantum algorithm that solves SVP on an arbitrary lattice significantly faster than existing classical algorithms.[1] However, some quantum speed-ups for SVP algorithms are possible in general.

It was shown by Aono–Nguyen–Shen [ANS18] that enumeration algorithms for SVP can be sped up using the *quantum backtracking* algorithm of Montanaro [Mon18]. More precisely, with quantum enumeration one solves SVP on a d-dimensional lattice in time $2^{\frac{1}{4e}d\log d+o(d\log d)}$, a square root improvement over classical enumeration. This algorithm requires $\mathrm{poly}(d)$ classical and quantum memory. This bound holds for both provable and heuristic versions of enumeration. Quantum speed-ups for sieving algorithms have been considered by Laarhoven–Mosca–van de Pol [LMvdP15] and later by Laarhoven [Laa15]. The latter result presents various quantum sieving algorithms for SVP. One of them achieves time and classical memory of order $2^{0.2653d+o(d)}$ and requires $\mathrm{poly}(d)$ quantum memory. This is the best known quantum time complexity for heuristic sieving algorithms. Provable single exponential SVP solvers were considered in the quantum setting by Chen–Chang–Lai [CCL17]. Based on [ADRSD15,DRS14], the authors describe a $2^{1.255d+o(d)}$ time, $2^{0.5d+o(d)}$ classical and $\mathrm{poly}(d)$ quantum memory algorithm for SVP. All heuristic and provable results rely on the classical memory being quantumly addressable.

[1] For some families of lattices, like ideal lattices, there exist quantum algorithms that solve a variant of SVP faster than classical algorithms, see [CDW17,PMHS19]. In this work, we consider arbitrary lattices.

A drawback of sieving algorithms is their large memory requirements. Initiated by Bai–Laarhoven–Stehlé, a line of work [BLS16,HK17,HKL18] gave a family of heuristic sieving algorithms, called tuple lattice sieves, or k-Sieves for some fixed constant k, that offer time-memory trade-offs. Such trade-offs have proven important in the current fastest SVP solvers, as the ideas of tuple sieving offer significant speed-ups in practice, [ADH$^+$19]. In this work, we explore various directions for *asymptotic* quantum accelerations of tuple sieves.

Our results.

1. In Section 4 we show how to use a quantum computer to speed up the k-Sieve of Bai–Laarhoven–Stehlé [BLS16] and its improvement due to Herold–Kirshanova–Laarhoven [HKL18] (Algorithms 4.1,4.2). One data point achieves a time complexity of $2^{0.2989d+o(d)}$, while requiring $2^{0.1395d+o(d)}$ classical memory and poly(d) width quantum circuits. In the $\texttt{Area}\times\texttt{Time}$ model this beats the previously best known algorithm [Laa15] of time and memory complexities $2^{0.2653d+o(d)}$; we almost halve the constant in the exponent for memory at the cost of a small increase in the respective constant for time.
2. Borrowing ideas from [Laa15] we give a quantum k-Sieve (Algorithm B.2) that also exploits nearest neighbour techniques. For $k = 2$, we recover Laarhoven's $2^{0.2653d+o(d)}$ time and memory quantum algorithm.
3. In Section 5 the k-Sieve is reduced to listing k-cliques in a graph. By generalising the triangle finding algorithm of [BdWD$^+$01] this approach leads to an algorithm that matches the performance of Algorithm 4.1, when optimised for time, for all k.
4. In Section 6 we specialise to listing 3-cliques (triangles) in a graph. Using the quantum triangle finding algorithm of [LGN17] allows us, in the *query model*,[2] to perform the 3-Sieve using $2^{0.3264d+o(d)}$ *queries.*
5. In Section 7 we describe a quantum circuit consisting only of gates from a universal gate set (e.g. CNOT and single qubit rotations) of depth $2^{0.1038d+o(d)}$ and width $2^{0.2075d+o(d)}$ that implements the 2-Sieve as proposed classically in [NV08]. In particular we consider exponential *quantum* memory to make significant improvements to the number of time steps. Our construction adapts the parallel search procedure of [BBG$^+$13].

Our main results, quantum time-memory trade-offs for sieving algorithms, are summarised in Figure 1. When optimising for time a quantum 2-Sieve with locality sensitive filtering (LSF) remains the best algorithm. For $k \geq 3$ the speed-ups offered by LSF are less impressive, and one can achieve approximately the same asymptotic time complexity by considering quantum k-Sieve algorithms (without LSF) with $k \geq 10$ and far less memory.

All the results presented in this work are asymptotic in nature: our algorithms have time, classical memory, quantum memory complexities of orders $2^{\mathsf{c}d+o(d)}$, $2^{\mathsf{c}'d+o(d)}$, poly(d) or $2^{\mathsf{c}''d+o(d)}$ respectively, for $\mathsf{c},\mathsf{c}',\mathsf{c}'' \in \Theta(1)$, which we aim to minimise. We do not attempt to specify the $o(d)$ or poly(d) terms.

[2] This means that the complexity of the algorithm is measured by the number of oracle calls to the adjacency matrix of a graph.

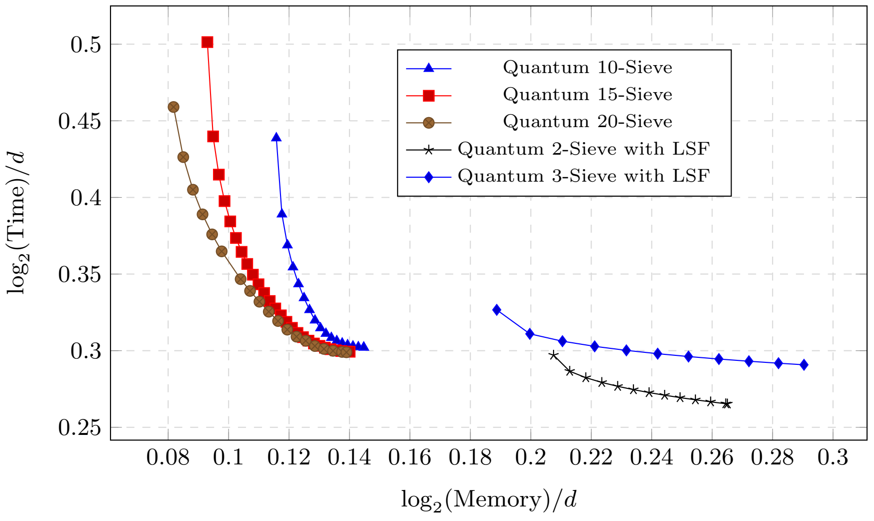


Fig. 1: Time-memory trade-offs for Algorithm 4.1 with $k \in \{10, 15, 20\}$ and Algorithm B.2 with $k \in \{2, 3\}$. Each curve provides time-memory trade-offs for a fixed k, either with nearest neighbour techniques (the right two curves) or without (the left three curves). Each point on a curve (x, y) represents (Memory, Time) values, obtained by numerically optimising for time while fixing available memory. For example, we build the leftmost curve (dotted brown) by computing the memory optimal (Memory, Time) value for the 20-Sieve and then repeatedly increase the available memory (decreasing time) until we reach the time optimal (Memory, Time) value. Increasing memory further than the rightmost point on each curve does not decrease time. The figures were obtained using optimisation package provided by Maple™ [Map].

Our techniques. We now briefly describe the main ingredients of our results.

1. A useful abstraction of the k-Sieve is the *configuration problem*, first described in [HK17]. It consists of finding k elements that satisfy certain pairwise inner product constraints from k exponentially large lists of vectors. Assuming $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ is a solution tuple, the i^{th} element $\mathbf{x}_i$ can be obtained via a brute force search either over the i^{th} input list [BLS16], or over a certain sublist of the i^{th} list [HK17], see Figure 2b. We replace the brute force searches with calls to Grover's algorithm and reanalyse the configuration problem.
2. An alternative way to find the i^{th} element of a solution tuple for the configuration problem is to apply nearest neighbour techniques [Laa15,BLS16]. This method sorts the i^{th} list into a specially crafted data structure that, for a given $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})$, allows one to find the satisfying $\mathbf{x}_i$ faster than via brute force. The search for $\mathbf{x}_i$ within such a data structure can itself be sped up by Grover's algorithm.

3. The configuration problem can be reduced to the k-clique problem in a graph with vertices representing elements from the lists given by the configuration problem. Vertices are connected by an edge if and only if the corresponding list elements satisfy some inner product constraint. Classically, this interpretation yields no improvements to configuration problem algorithms. However we achieve quantum speed-ups by generalising the triangle finding algorithm of Buhrman et al. [BdWD$^+$01] and applying it to k-cliques.
4. We apply the triangle finding algorithm of Le Gall–Nakajima [LGN17] and exploit the structure of our graph instance. In particular we form many graphs from unions of sublists of our lists, allowing us to alter the sparsity of said graphs.
5. To make use of more quantum memory we run Grover searches in parallel. The idea is to allow simultaneous queries by several processors to a large, shared, quantum memory. Instead of looking for a "good" $\mathbf{x}_i$ for one *fixed* tuple $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})$, one could think of parallel searches aiming to find a "good" $\mathbf{x}_i$ for several tuples $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})$. The possibility of running several Grover's algorithms concurrently was shown in the work of Beals et al. [BBG$^+$13]. Based on this result we specify all the subroutines needed to solve the shortest vector problem using large quantum memory.

Open questions.

1. The classical configuration search algorithms of [HKL18] offer time-memory trade-offs for SVP by varying k (larger k requires less memory but more time). We observe in Section 3 that time optimal classical algorithms for the configuration problem hit a certain point on the time-memory trade-off curve once k becomes large enough, see Table 1. The same behaviour is observed for our quantum algorithms for the configuration problem, see Table 2. Although we provide some explanation of this, we do not rigorously prove that the trade-off curve indeed stops on its time optimal side. We leave it as an open problem to determine the shape of the configuration problem for these time optimal instances of the algorithm. Another open problem originating from [HK17] is to extend the analysis to non constant k.
2. We do not give time complexities in Section 6, instead reporting the query complexity for listing triangles. We leave open the question of determining, e.g. the complexity of forming auxiliary databases used by the quantum random walks on Johnson graphs of [LGN17], which is not captured in the query model, as well as giving the (quantum) memory requirements of these methods in our setting. If the asymptotic time complexity does not increase (much) above the query complexity then the $2^{0.3264d+o(d)}$ achieved by the algorithm in Section 6 represents both an improvement over the best classical algorithms for the relevant configuration problem [HKL18] and an improvement over Algorithms 4.1, 4.2 for $k = 3$ in low memory regimes, see Table 2.
3. In Section 7 we present a parallel quantum version of a 2-Sieve. We believe that it should be possible to extend the techniques to k-Sieve for $k > 2$.

2 Preliminaries

We denote by $\mathsf{S}^d \subset \mathbb{R}^{d+1}$ the d-dimensional unit sphere. We use soft-$\mathcal{O}$ notation to denote running times, that is $T = \widetilde{\mathcal{O}}(2^{\mathsf{c}d})$ suppresses subexponential factors in d. By $[n]$ we denote the set $\{1, \ldots, n\}$. The norm considered in this work is Euclidean and is denoted by $\|\cdot\|$.

For any set $\mathbf{x}_1, \ldots, \mathbf{x}_k$ of vectors in $\mathbb{R}^d$, the *Gram matrix* $C \in \mathbb{R}^{k\times k}$ is given by $C_{i,j} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$, the set of pairwise scalar products. For $I \subset [k]$, we denote by $C[I]$ the $|I| \times |I|$ submatrix of C obtained by restricting C to the rows and columns indexed by I. For a vector $\mathbf{x}$ and $i \in [k]$, $\mathbf{x}[i]$ denotes the ith entry. For a function f, by O_f we denote a unitary matrix that implements f.

Lattices. Given a basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\} \subset \mathbb{R}^d$ of linearly independent vectors $\mathbf{b}_i$, the lattice generated by B is defined as $\mathcal{L}(B) = \{\sum_{i=1}^m z_i \mathbf{b}_i \colon z_i \in \mathbb{Z}\}$. For simplicity we work with lattices of full rank ($d = m$). The Shortest Vector Problem (SVP) is to find, for a given B, a shortest non zero vector of $\mathcal{L}(B)$. Minkowski's theorem for the Euclidean norm states that a shortest vector of $\mathcal{L}(B)$ is bounded from above by $\sqrt{d} \cdot \det(B)^{1/d}$.

Quantum Search. Our results rely on Grover's quantum search algorithm [Gro96] which finds "good" elements in a (large) list. The analysis of the success probability of this algorithm can be found in [BBHT98]. We also rely on the generalisation of Grover's algorithm, called Amplitude Amplification, due to Brassard–Høyer–Mosca–Tapp [BHMT02] and a result on parallel quantum search [BBG+13].

Theorem 1 (Grover's algorithm [Gro96,BBHT98]). *Given quantum access to a list L that contains t marked elements (the value t is not necessarily known) and a function $f \colon L \to \{0,1\}$, described by a unitary O_f, which determines whether an element is "good" or not, we wish to find a solution $i \in [|L|]$, such that for $f(x_i) = 1, x_i \in L$. There exists a quantum algorithm, called Grover's algorithm, that with probability greater than $1 - t/|L|$ outputs one "good" element using $\mathcal{O}(\sqrt{|L|/t})$ calls to O_f.*

Theorem 2 (Amplitude Amplification [BHMT02, Theorem 2]). *Let $\mathcal{A}$ be any quantum algorithm that makes no measurements and let $\mathcal{A}|0\rangle = |\Psi_0\rangle + |\Psi_1\rangle$, where $|\Psi_0\rangle$ and $|\Psi_1\rangle$ are spanned by "bad" and "good" states respectively. Let further $a = \langle \Psi_1 | \Psi_1 \rangle$ be the success probability of $\mathcal{A}$. Given access to a function f that flips the sign of the amplitudes of good states, i.e. $f \colon |x\rangle \mapsto -|x\rangle$ for "good" $|x\rangle$ and leaves the amplitudes of "bad" $|x\rangle$ unchanged, the amplitude amplification algorithm constructs the unitary $Q = -\mathcal{A}R\mathcal{A}^{-1}O_f$, where R is the reflection about $|0\rangle$, and applies Q^m to the state $\mathcal{A}|0\rangle$, where $m = \lfloor \frac{\pi}{4} \arcsin(\sqrt{a}) \rfloor$. Upon measurement of the system, a "good" state is obtained with probability at least $\max\{a, 1-a\}$.*

Theorem 3 (Quantum Parallel Search [BBG+13]). *Given a list L, with each element of bit length d, and $|L|$ functions that take list elements as input $f_i \colon L \to \{0,1\}$ for $i \in [|L|]$, we wish to find solution vectors $\mathbf{s} \in [|L|]^{|L|}$.*

A solution has $f_i(\mathbf{x}_{\mathbf{s}[i]}) = 1$ for all $i \in [|L|]$. Given unitaries $U_{f_i} \colon |\mathbf{x}\rangle |b\rangle \to |\mathbf{x}\rangle |b \oplus f_i(\mathbf{x})\rangle$ there exists a quantum algorithm that, for each $i \in [|L|]$, either returns a solution $\mathbf{s}[i]$ or if there is no such solution, returns no solution. *The algorithm succeeds with probability $\Theta(1)$ and, given that each U_{f_i} has depth and width $\operatorname{poly}\log(|L|, d)$, can be implemented using a quantum circuit of width $\widetilde{\mathcal{O}}(|L|)$ and depth $\widetilde{\mathcal{O}}(\sqrt{|L|})$.*

Computational Models. Our algorithms are analysed in the quantum circuit model [KLM07]. We set each wire to represent a qubit, i.e. a vector in a two dimensional complex Hilbert space, and assert that we have a set of universal gates. We work in the noiseless quantum theory, i.e. we assume there is no (or negligible) decoherence or other sources of noise in the computational procedures.

The algorithms given in Sections 4 and 5 are in the QRAM model and assume quantumly accessible classical memory [GLM08]. More concretely in this model we store all data, e.g. the list of vectors, in classical memory and only demand that this memory is quantumly accessible, i.e. elements in the list can be efficiently accessed in coherent superposition. This enables us to design algorithms that, in principle, do not require large quantum memories and can be implemented with only $\operatorname{poly}(d)$ qubits and with the $2^{\Theta(d)}$ sized list stored in classical memory. Several works [BHT97,Kup13] suggest that this memory model is potentially easier to achieve than a full quantum memory.

In Section 6 we study the algorithms in the query model, which is the typical model for quantum triangle or k-clique finding algorithms. Namely, the complexity of our algorithm is measured in the number of oracle calls to the adjacency matrix of a graph associated to a list of vectors.

Acknowledging the arguments against the feasibility of QRAM and whether it can be meaningfully cheaper than quantum memory [AGJO$^+$15], in Section 7 we consider algorithms that use exponential quantum memory in the quantum circuit model without assuming QRAM.

3 Sieving as Configuration Search

In this section we describe previously known *classical* sieving algorithms. We will not go into detail or give proofs, which can be found in the relevant references.

Sieving algorithms receive on input a basis $B \in \mathbb{R}^{d \times d}$ and start by sampling an exponentially large list L of (long) lattice vectors from $\mathcal{L}(B)$. There are efficient algorithms for sampling lattice vectors, e.g. [Kle00]. The elements of L are then iteratively combined to form shorter lattice vectors, $\mathbf{x}_{\text{new}} = \mathbf{x}_1 \pm \mathbf{x}_2 \pm \ldots \pm \mathbf{x}_k$ such that $\|\mathbf{x}_{\text{new}}\| \leq \max_{i \leq k}\{\|\mathbf{x}_i\|\}$, for some $k \geq 2$. Newly obtained vectors $\mathbf{x}_{\text{new}}$ are stored in a new list and the process is repeated with this new list of shorter vectors. It can be shown [NV08,Reg09] that after $\operatorname{poly}(d)$ such iterations we obtain a list that contains a shortest vector. Therefore, the asymptotic complexity of sieving is determined by the cost of finding k-tuples whose combination produces shorter vectors. Under certain heuristics, specified below, finding such k-tuples can be formulated as the approximate k-List problem.

Definition 1 (Approximate k-List problem). *Assume we are given k lists $L_1, \ldots, L_k$ of equal exponential (in d) size $|L|$ and whose elements are i.i.d. uniformly chosen vectors from S^{d-1}. The approximate k-List problem is to find $|L|$ solutions, where a solution is a k-tuple $(x_1, \ldots, x_k) \in L_1 \times \ldots \times L_k$ satisfying $\|\mathbf{x}_1 + \ldots + \mathbf{x}_k\| \leq 1$.*

The assumption made in analyses of heuristic sieving algorithms [NV08] is that the lattice vectors in the new list after an iteration are thought of as i.i.d. uniform vectors on a thin spherical shell (essentially, a sphere), and, once normalised, on S^{d-1}. Hence sieves do not "see" the discrete structure of the lattice from the vectors operated on. The heuristic becomes invalid when the vectors become short. In this case we assume we have solved SVP. Thus, we may not find a *shortest* vector, but an approximation to it, which is enough for most cryptanalytic purposes.

We consider k to be constant. The lists $L_1, \ldots, L_k$ in Definition 1 may be identical. The algorithms described below are applicable to this case as well. Furthermore, the approximate k-List problem only looks for solutions with $+$ signs, i.e. $\|\mathbf{x}_1 + \ldots + \mathbf{x}_k\| \leq 1$, while sieving looks for arbitrary signs. This is not an issue, as we may repeat an algorithm for the approximate k-List problem $2^k = \mathcal{O}(1)$ times in order to obtain all solutions.

Configuration Search. Using a concentration result on the distribution of scalar products of $\mathbf{x}_1, \ldots, \mathbf{x}_k \in \mathsf{S}^{d-1}$ shown in [HK17], the approximate k-List problem can be reduced to the configuration problem. In order to state this problem, we need a notion of configurations.

Definition 2 (Configuration). *The configuration $C = \mathrm{Conf}(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ of k points $\mathbf{x}_1, \ldots, \mathbf{x}_k \in \mathsf{S}^{d-1}$ is the Gram matrix of the $\mathbf{x}_i$, i.e. $C_{i,j} = \langle \mathbf{x}_i \,, \mathbf{x}_j \rangle$.*

A configuration $C \in \mathbb{R}^{k \times k}$ is a positive semidefinite matrix. Rewriting the solution condition $\|\mathbf{x}_1 + \ldots + \mathbf{x}_k\|^2 \leq 1$, one can check that a configuration C for a solution tuple satisfies $\mathbf{1}^{\mathsf{t}} C \mathbf{1} \leq 1$. We denote the set of such "good" configurations by

$$\mathscr{C} = \{C \in \mathbb{R}^{k \times k} \colon C \text{ is positive semidefinite and } \mathbf{1}^{\mathsf{t}} C \mathbf{1} \leq 1\}.$$

It has been shown [HK17] that rather than looking for k-tuples that form a solution for the approximate k-List problem, we may look for k-tuples that satisfy a constraint on their configuration. It gives rise to the following problem.

Definition 3 (Configuration problem). *Let $k \in \mathbb{N}$ and $\varepsilon > 0$. Suppose we are given a target configuration $C \in \mathscr{C}$. Given k lists $L_1, \ldots, L_k$ all of exponential (in d) size $|L|$, whose elements are i.i.d. uniform from S^{d-1}, the configuration problem consists of finding a $1 - o(1)$ fraction of all solutions, where a solution is a k-tuple $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ with $\mathbf{x}_i \in L_i$ such that $|\langle \mathbf{x}_i \,, \mathbf{x}_j \rangle - C_{i,j}| \leq \varepsilon$ for all i, j.*

Solving the configuration problem for a $C \in \mathscr{C}$ gives solutions to the approximate k-List problem. For a given $C \in \mathbb{R}^{k \times k}$ the number of expected solutions to the configuration problem is given by $\det(C)$ as the following theorem shows.

Theorem 4 (Distribution of configurations [HK17, Theorem 1]). *If $\mathbf{x}_1, \ldots, \mathbf{x}_k$ are i.i.d. from S^{d-1}, $d > k$, then their configuration $C = \mathrm{Conf}(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ follows a distribution with density function*

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

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

This theorem tells us that the expected number of solutions to the configuration problem for C is given by $\prod_i |L_i| \cdot (\det C)^{d/2}$. If we want to apply an algorithm for the configuration problem to the approximate k-List problem (and to sieving), we require that the expected number of output solutions to the configuration problem is equal to the size of the input lists. Namely, C and the input lists L_i of size $|L|$ should (up to polynomial factors) satisfy $|L|^k \cdot (\det C)^{d/2} = |L|$. This condition gives a lower bound on the size of the input lists. Using Chernoff bounds, one can show (see [HKL18, Lemma 2]) that increasing this bound by a poly(d) factor gives a sufficient condition for the size of input lists, namely

$$|L| = \widetilde{\mathcal{O}}\left(\left(\frac{1}{\det(C)}\right)^{\frac{d}{2(k-1)}}\right). \tag{2}$$

Classical algorithms for the configuration problem. The first classical algorithm for the configuration problem for $k \geq 2$ was given by Bai–Laarhoven–Stehlé [BLS16]. It is depicted in Figure 2a. It was later improved by Herold–Kirshanova [HK17] and by Herold–Kirshanova–Laarhoven [HKL18] (Figure 2b). These results present a family of algorithms for the configuration problem that offer time-memory trade-offs. In Section 4 we present quantum versions of these algorithms.

Both algorithms [BLS16,HKL18] process the lists from left to right but in a different manner. For each $\mathbf{x}_1 \in L_1$ the algorithm from [BLS16] applies a filtering procedure to L_2 and creates the "filtered" list $L_2(\mathbf{x}_1)$. This filtering procedure takes as input an element $\mathbf{x}_2 \in L_2$ and adds it to $L_2(\mathbf{x}_1)$ iff $|\langle \mathbf{x}_1 , \mathbf{x}_2 \rangle - C_{1,2}| \leq \varepsilon$. Having constructed the list $L_2(\mathbf{x}_1)$, the algorithm then iterates over it: for each $\mathbf{x}_2 \in L_2(\mathbf{x}_1)$ it applies the filtering procedure to L_3 with respect to $C_{2,3}$ and obtains $L_3(\mathbf{x}_1, \mathbf{x}_2)$. Throughout, vectors in brackets indicate fixed elements with respect to which the list has been filtered. Filtering of the top level lists $(L_1, \ldots, L_k)$ continues in this fashion until we have constructed $L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1})$ for fixed values $\mathbf{x}_1, \ldots, \mathbf{x}_{k-1}$. The tuples of the form $(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1}, \mathbf{x}_k)$ for all $\mathbf{x}_k \in L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1})$ form solutions to the configuration problem.

The algorithms from [HK17,HKL18] apply more filtering steps. For a fixed $\mathbf{x}_1 \in L_1$, they not only create $L_2(\mathbf{x}_1)$, but also $L_3(\mathbf{x}_1), \ldots, L_k(\mathbf{x}_1)$. This speeds up the next iteration over all $\mathbf{x}_2 \in L_2(\mathbf{x}_1)$, where now the filtering step with respect to $C_{2,3}$ is applied not to L_3, but to $L_3(\mathbf{x}_1)$, as well as to $L_4(\mathbf{x}_1), \ldots, L_k(\mathbf{x}_1)$, each of which is smaller than L_i. This speeds up the construction of $L_3(\mathbf{x}_1, \mathbf{x}_2)$. The algorithm continues with this filtering process until the last inner product check with respect to $C_{k-1,k}$ is applied to all the elements from $L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})$

and the list $L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1})$ is constructed. This gives solutions of the form $(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1}, \mathbf{x}_k)$ for all $\mathbf{x}_k \in L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1})$. The concentration result, Theorem 4, implies the outputs of algorithms from [BLS16] and [HK17,HKL18] are (up to a subexponential fraction) the same. Pseudocode for [HK17] can be found in Appendix A.

Important for our analysis in Section 4 will be the the result of [HKL18] that describes the sizes of all the intermediate lists that appear during the configuration search algorithms via the determinants of submatrices of the target configuration C. The next theorem gives the expected sizes of these lists and the time complexity of the algorithm from [HKL18].

Theorem 5 (Intermediate list sizes [HKL18, Lemma 1] and time compleixty of configuration search algorithm). *During a run of the configuration search algorithms described in Figures 2a, 2b, given an input configuration $C \in \mathbb{R}^{k \times k}$ and lists $L_1, \ldots, L_k \subset \mathsf{S}^{d-1}$ each of size $|L|$, the intermediate lists for $1 \leq i < j \leq k$ are of expected sizes*

$$\mathbb{E}[|L_j(\mathbf{x}_1, \ldots, \mathbf{x}_i)|] = |L| \cdot \left(\frac{\det(C[1, \ldots, i, j])}{\det(C[1 \ldots i])} \right)^{d/2}. \tag{3}$$

The expected running time of the algorithm described in Figure 2b is

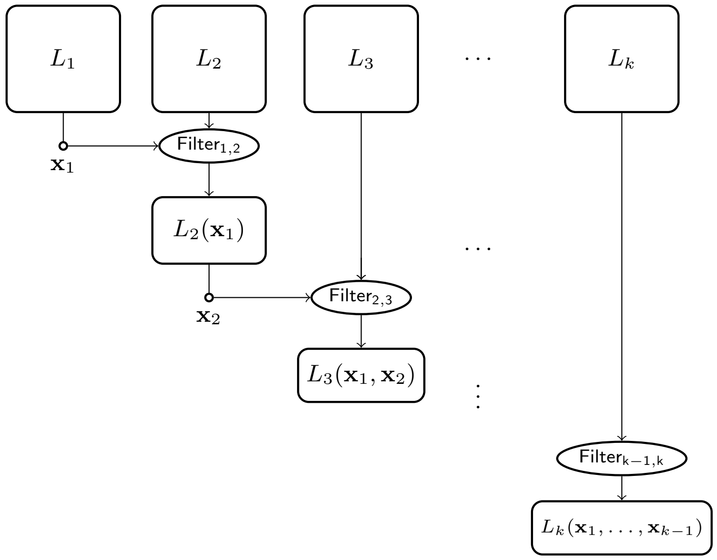
$$T^{\mathrm{C}}_{\text{k-Conf}} = \max_{1 \leq i \leq k} \left[\prod_{r=1}^{i} |L_r(\mathbf{x}_1, \ldots, \mathbf{x}_{r-1})| \cdot \max_{i+1 \leq j \leq k} |L_j(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})| \right]. \tag{4}$$

Finding a configuration for optimal runtime. For a given i the square bracketed term in Eq. (4) represents the expected time required to create all filtered lists on a given "level". Here "level" refers to all lists filtered with respect to the same fixed $\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}$, i.e. a row of lists in Figure 2b. In order to find an optimal configuration C that minimises Eq. (4), we perform numerical optimisations using the Maple™ package [Map].[3] In particular, we search for $C \in \mathscr{C}$ that minimises Eq. (4) under the condition that Eq. (2) is satisfied (so that we actually obtain enough solutions for the k-List problem). Figures for the optimal runtime and the corresponding memory are given in Table 1. The memory is determined by the size of the input lists computed from the optimal C using Eq. (2). Since the k-List routine determines the asymptotic cost of k-Sieve, the figures in Table 1 are also the constants in the exponents for complexities of k-Sieves.

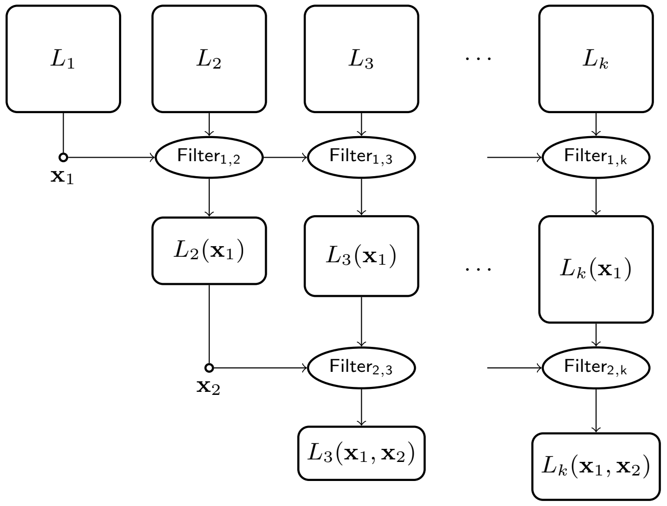
Interestingly, the optimal runtime constant turns out to be equal for large enough k. This can be explained as follows. The optimal C achieves the situation where all the expressions in the outer max in Eq. (4) are equal. This implies that creating all the filtered lists on level i asymptotically costs the same as creating all the filtered lists on level $i+1$ for $2 \leq i \leq k-1$. The cost of creating filtered lists $L_i(\mathbf{x}_1)$ on the second level (assuming that the first level consists of the input lists) is of order $|L|^2$. This value, $|L|^2$, becomes (up to poly(d) factors) the running time of the whole algorithm (compare the Time and Space constants

[3] The code is available at `https://github.com/ElenaKirshanova/QuantumSieve`

Fig. 2: Algorithms for the configuration problem. Procedures $\mathsf{Filter}_{i,j}$ receive as input a vector (e.g. $\mathbf{x}_1$), 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. $L_2(\mathbf{x}_1)$) 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.



(a) The algorithm of Bai et al. [BLS16] for the configuration problem.



(b) The algorithm of Herold et al. [HKL18] for the configuration problem.

k	2	3	4	5	6	...	16	17	18
Time	0.4150	0.3789	0.3702	0.3707	0.3716		0.3728	0.37281	0.37281
Space	0.2075	0.1895	0.1851	0.1853	0.1858		0.1864	0.18640	0.18640

Table 1: Asymptotic complexity exponents for the approximate k-List problem, base 2. The table gives optimised runtime and the corresponding memory exponents for the classical algorithm from [HKL18], see Figure 2b and Algorithm A.1.

for $k = 16, 17, 18$ in Table 1). The precise shape of $C \in \mathscr{C}$ that balances the costs per level can be obtained by equating all the terms in the max of Eq. (4) and minimising the value $|L|^2$ under these constraints. Even for small k these computations become rather tedious and we do not attempt to express $C_{i,j}$ as a function of k, which is, in principal, possible.

Finding a configuration for optimal memory. If we want to optimise for memory, the optimal configuration C has all its off diagonal elements $C_{i,j} = -1/k$. It is shown in [HK17] that such C maximises $\det(C)$ among all $C \in \mathscr{C}$, which, in turn, minimises the sizes of the input lists (but does not lead to optimal running time as the costs per level are not balanced).

4 Quantum Configuration Search

In this section we present several quantum algorithms for the configuration problem (Definition 3). As explained in Section 3, this directly translates to quantum sieving algorithms for SVP. We start with a quantum version of the BLS style configuration search [BLS16], then we show how to improve this algorithm by constructing intermediate lists. In Appendix B we show how nearest neighbour methods in the quantum setting speed up the latter algorithm.

Recall the configuration problem: as input we receive k lists $L_i, i \in [k]$ each of size a power of two,[4] a configuration matrix $C \in \mathbb{R}^{k \times k}$ and $\varepsilon \geq 0$. To describe our first algorithm we denote by $f_{[i],j}$ a function that takes as input $(i+1)$ many d-dimensional vectors and is defined as

$$f_{[i],j}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \mathbf{x}) = \begin{cases} 1, & |\langle \mathbf{x}_\ell \,, \mathbf{x} \rangle - C_{\ell,j}| \leq \varepsilon, \quad \ell \in [i] \\ 0, & \text{else.} \end{cases}$$

A reversible embedding of $f_{[i],j}$ is denoted by $O_{f_{[i],j}}$. Using these functions we perform a check for "good" elements and construct the lists $L_j(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_i)$. Furthermore, we assume that any vector encountered by the algorithm fits into $\bar{d}$ qubits. We denote by $|\mathbf{0}\rangle$ the $\bar{d}$-tensor of 0 qubits, i.e. $|\mathbf{0}\rangle = |0^{\otimes \bar{d}}\rangle$.

[4] This is not necessary but it enables us to efficiently create superpositions $|\Psi_{L_i}\rangle$ using Hadamard gates. Since our lists L_i are of sizes $2^{\mathsf{c}d+o(d)}$ for a large d and a constant $\mathsf{c} < 1$, this condition is easy to satisfy by rounding $\mathsf{c}d$.

The input lists, $L_i, i \in [k]$, are stored classically and are assumed to be quantumly accessible. In particular, we assume that we can efficiently construct a uniform superposition over all elements from a given list by first applying Hadamards to $|\mathbf{0}\rangle$ to create a superposition over all indices, and then by querying $L[i]$ for each i in the superposition. That is, we assume an efficient circuit for $\frac{1}{\sqrt{|L|}} \sum_i |i\rangle |\mathbf{0}\rangle \to \frac{1}{\sqrt{|L|}} \sum_i |i\rangle |L[i]\rangle$. For simplicity, we ignore the first qubit that stores indices and we denote by $|\Psi_L\rangle$ a uniform superposition over all the elements in L, i.e. $|\Psi_L\rangle = \frac{1}{\sqrt{|L|}} \sum_{\mathbf{x} \in L} |\mathbf{x}\rangle$.

The idea of our algorithm for the configuration problem is the following. We have a global *classical* loop over $\mathbf{x}_1 \in L_1$ inside which we run our quantum algorithm to find a $(k-1)$ tuple $(\mathbf{x}_2, \ldots, \mathbf{x}_k)$ that together with $\mathbf{x}_1$ gives a solution to the configuration problem. We expect to have $\mathcal{O}(1)$ such $(k-1)$ tuples per $\mathbf{x}_1$.[5] At the end of the algorithm we expect to obtain such a solution by means of amplitude amplification (Theorem 2). In Theorem 6 we argue that this procedure succeeds in finding a solution with probability at least $1 - 2^{-\Omega(d)}$.

Inside the classical loop over $\mathbf{x}_1$ we prepare $(k-1)\bar{d}$ qubits, which we arrange into $k-1$ registers, so that each register will store (a superposition of) input vectors, see Figure 3. Each such register is set in uniform superposition over the elements of the input lists: $|\Psi_{L_2}\rangle \otimes |\Psi_{L_3}\rangle \otimes \cdots \otimes |\Psi_{L_k}\rangle$. We apply Grover's algorithm on $|\Psi_{L_2}\rangle$. Each Grover's iteration is defined by the unitary $Q_{1,2} = -H^{\otimes \bar{d}} R H^{\otimes \bar{d}} O_{f_{[1],2}}$. Here H is the Hadamard gate and R is the rotation around $|\mathbf{0}\rangle$. We have $|L_2(\mathbf{x}_1)|$ "good" states out of $|L_2|$ possible states in $|\Psi_{L_2}\rangle$, so after $\mathcal{O}\left(\sqrt{\frac{|L_2|}{|L_2(\mathbf{x}_1)|}}\right)$ applications of $Q_{1,2}$ we obtain the state

$$|\Psi_{L_2(\mathbf{x}_1)}\rangle = \frac{1}{\sqrt{|L_2(\mathbf{x}_1)|}} \sum_{\mathbf{x}_2 \in L_2(\mathbf{x}_1)} |\mathbf{x}_2\rangle . \tag{5}$$

In fact, what we create is a state close to Eq. (5) as we do not perform any measurement. For now, we drop the expression "close to" for all the states in this description, and argue about the failure probability in Theorem 6.

Now consider the state $|\Psi_{L_2(\mathbf{x}_1)}\rangle \otimes |\Psi_{L_3}\rangle$ and the function $f_{[2],3}$ that uses the first and second registers and a fixed $\mathbf{x}_1$ as inputs. We apply the unitary $Q_{2,3}$ to $|\Psi_{L_3}\rangle$, where $Q_{2,3} = -H^{\otimes \bar{d}} R H^{\otimes \bar{d}} O_{f_{[2],3}}$. In other words, for all vectors from L_3, we check if they satisfy the inner product constraints with respect to $\mathbf{x}_1$ and $\mathbf{x}_2$. In this setting there are $|L_3(\mathbf{x}_1, \mathbf{x}_2)|$ "good" states in $|\Psi_{L_3}\rangle$ whose amplitudes we aim to amplify. Applying Grover's iteration unitary $Q_{2,3}$ the order of $\mathcal{O}\left(\sqrt{\frac{|L_3|}{|L_3(\mathbf{x}_1,\mathbf{x}_2)|}}\right)$ times, we obtain the state

$$|\Psi_{L_2(\mathbf{x}_1)}\rangle |\Psi_{L_3(\mathbf{x}_1,\mathbf{x}_2)}\rangle = \frac{1}{\sqrt{|L_2(\mathbf{x}_1)|}} \sum_{\mathbf{x}_2 \in L_2(\mathbf{x}_1)} |\mathbf{x}_2\rangle \left(\frac{1}{\sqrt{|L_3(\mathbf{x}_1, \mathbf{x}_2)|}} \sum_{\mathbf{x}_3 \in L_3(\mathbf{x}_1,\mathbf{x}_2)} |\mathbf{x}_3\rangle \right) .$$

[5] This follows by multiplying the sizes of the lists $L_i(\mathbf{x}_1, \ldots \mathbf{x}_{i-1})$ for all $2 \leq i \leq k$.

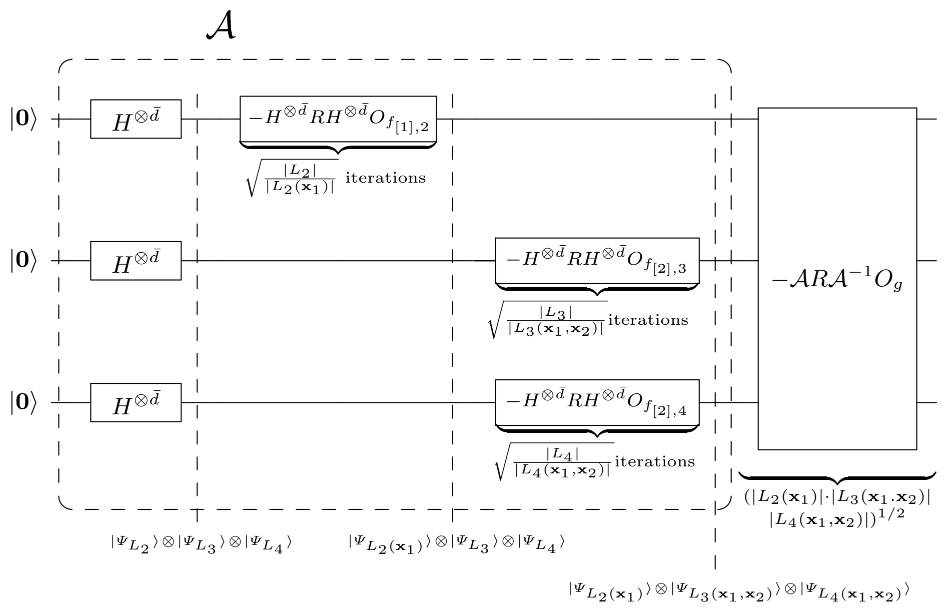

$\mathcal{A}$
$|\mathbf{0}\rangle$
$|\mathbf{0}\rangle$
$|\mathbf{0}\rangle$
$H^{\otimes \bar{d}}$
$H^{\otimes \bar{d}}$
$H^{\otimes \bar{d}}$
$-H^{\otimes \bar{d}} R H^{\otimes \bar{d}} O_{f_{[1],2}}$
$\sqrt{\frac{|L_2|}{|L_2(\mathbf{x}_1)|}}$ iterations
$-H^{\otimes \bar{d}} R H^{\otimes \bar{d}} O_{f_{[2],3}}$
$\sqrt{\frac{|L_3|}{|L_3(\mathbf{x}_1,\mathbf{x}_2)|}}$ iterations
$-H^{\otimes \bar{d}} R H^{\otimes \bar{d}} O_{f_{[2],4}}$
$\sqrt{\frac{|L_4|}{|L_4(\mathbf{x}_1,\mathbf{x}_2)|}}$ iterations
$-\mathcal{A} R \mathcal{A}^{-1} O_g$
$(|L_2(\mathbf{x}_1)| \cdot |L_3(\mathbf{x}_1 . \mathbf{x}_2)| \, |L_4(\mathbf{x}_1,\mathbf{x}_2)|)^{1/2}$
$|\Psi_{L_2}\rangle \otimes |\Psi_{L_3}\rangle \otimes |\Psi_{L_4}\rangle$
$|\Psi_{L_2(\mathbf{x}_1)}\rangle \otimes |\Psi_{L_3}\rangle \otimes |\Psi_{L_4}\rangle$
$|\Psi_{L_2(\mathbf{x}_1)}\rangle \otimes |\Psi_{L_3(\mathbf{x}_1,\mathbf{x}_2)}\rangle \otimes |\Psi_{L_4(\mathbf{x}_1,\mathbf{x}_2)}\rangle$

Fig. 3: Quantum circuit representing the quantum part of Algorithm 4.1 with $k = 4$, i.e. this circuit is executed inside the loop over $\mathbf{x}_1 \in L_1$. The Hadamard gates create the superposition $|\Psi_{L_2}\rangle \otimes |\Psi_{L_3}\rangle \otimes |\Psi_{L_4}\rangle$. We apply $\sqrt{\frac{|L_2|}{|L_2(\mathbf{x}_1)|}}$ Grover iterations to $|\Psi_{L_2}\rangle$ to obtain the state $|\Psi_{L_2(\mathbf{x}_2)}(\mathbf{x}_1)\rangle \otimes |\Psi_{L_3}\rangle \otimes |\Psi_{L_4}\rangle$. We then apply (sequentially) $\mathcal{O}\left(\sqrt{\frac{|L_3|}{|L_3(\mathbf{x}_1,\mathbf{x}_2)|}}\right)$ *resp.* $\mathcal{O}\left(\sqrt{\frac{|L_4|}{|L_4(\mathbf{x}_1,\mathbf{x}_2)|}}\right)$ Grover iterations to the second *resp.* third registers, where the checking function takes as input the first and second *resp.* the first and third registers. This whole process is $\mathcal{A}$ and is repeated $\mathcal{O}(\sqrt{|L_2(\mathbf{x}_1)| \cdot |L_3(\mathbf{x}_1, \mathbf{x}_2)| \, |L_4(\mathbf{x}_1, \mathbf{x}_2)|})$ times inside the amplitude amplification. Final measurement gives a triple $(\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ which, together with a fixed $\mathbf{x}_1$, forms a solution to the configuration problem.

We continue creating the lists $L_{i+1}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_i)$ by filtering the *initial* list L_{i+1} with respect to $\mathbf{x}_1$ (fixed by the outer classical loop), and with respect to $\mathbf{x}_2, \ldots, \mathbf{x}_i$ (given in a superposition) using the function $f_{[i],i+1}$. At level $k-1$ we obtain the state $|\Psi_{L_2(\mathbf{x}_1)}\rangle \otimes |\Psi_{L_3(\mathbf{x}_1,\mathbf{x}_2)}\rangle \otimes \ldots \otimes |\Psi_{L_{k-1}(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle$. For the last list L_k we filter with respect to $\mathbf{x}_1, \ldots, \mathbf{x}_{k-2}$ as for the list L_{k-1}. Finally, for a fixed $\mathbf{x}_1$, the "filtered" state we obtained is of the form

$$|\Psi_F\rangle = |\Psi_{L_2(\mathbf{x}_1)}\rangle \otimes |\Psi_{L_3(\mathbf{x}_1,\mathbf{x}_2)}\rangle \otimes \ldots \otimes |\Psi_{L_{k-1}(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle \otimes |\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle . \tag{6}$$

The state is expected to contain $\mathcal{O}(1)$ many $(k-1)$-tuples $(\mathbf{x}_2, \ldots, \mathbf{x}_k)$ which together with $\mathbf{x}_1$ give a solution to the configuration problem. To prepare the state $|\Psi_F\rangle$ for a fixed $\mathbf{x}_1$, we need

$$T_{\text{InGrover}} = \mathcal{O}\left(\sqrt{\left(\frac{|L_2|}{|L_2(\mathbf{x}_1)|}\right)} + \ldots + \sqrt{\left(\frac{|L_k|}{|L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|}\right)}\right) \tag{7}$$

unitary operations of the form $(-H^{\otimes \bar{d}})RH^{\otimes \bar{d}}O_{f_{[i],j}}$. This is what we call the "inner" Grover procedure.

Let us denote by $\mathcal{A}$ an algorithm that creates $|\Psi_F\rangle$ from $|\mathbf{0}\rangle \otimes \ldots \otimes |\mathbf{0}\rangle$ in time T_{InGrover}. In order to obtain a solution tuple $(\mathbf{x}_2, \ldots, \mathbf{x}_k)$ we apply amplitude amplification using the unitary $Q_{\text{Outer}} = -\mathcal{A}R\mathcal{A}^{-1}O_g$, where g is the function that operates on the last two registers and is defined as

$$g(\mathbf{x}, \mathbf{x}') = \begin{cases} 1, & |\langle \mathbf{x} \,, \mathbf{x}' \rangle - C_{k-1,k}| \leq \varepsilon \\ 0, & \text{else.} \end{cases} \tag{8}$$

Notice that in the state $|\Psi_F\rangle$ it is only the last two registers storing $\mathbf{x}_{k-1}$ and $\mathbf{x}_k$ that are left to be checked against the target configuration. This is precisely what we use O_g to check. Let $|\mathbf{z}\rangle = |\mathbf{x}_2, \ldots, \mathbf{x}_k\rangle$ be a solution tuple. The state $|\mathbf{z}\rangle$ appears in $|\Psi_F\rangle$ with amplitude

$$\langle \mathbf{z} | \Psi_F \rangle = \mathcal{O}\left(\left(\sqrt{|L_2(\mathbf{x}_1)| \cdot \ldots \cdot |L_{k-1}(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})| \cdot |L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|} \right)^{-1} \right) .$$

This value is the inverse of the number of iteration steps Q_{Outer} which we repeat in order to obtain $\mathbf{z}$ when measuring $|\Psi_F\rangle$. The overall complexity of the algorithm for the configuration problem becomes

$$\begin{aligned} T^{\text{Q}}_{\text{BLS}} = \mathcal{O}\Bigg(|L_1| \left(\sqrt{\left(\frac{|L_2|}{|L_2(\mathbf{x}_1)|}\right)} + \ldots + \sqrt{\left(\frac{|L_k|}{|L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|}\right)} \right) \\ \cdot \sqrt{|L_2(\mathbf{x}_1)| \cdot |L_3(\mathbf{x}_1, \mathbf{x}_2)| \cdot \ldots \cdot |L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|} \Bigg) , \end{aligned} \tag{9}$$

where all the filtered lists in the above expression are assumed to be of expected size greater than or equal to 1. For certain target configurations intermediate lists are of sizes less than 1 in expectation (see Eq. (1)), which should be understood as the expected number of times we need to construct these lists to obtain 1 element in them. So there will exist elements in the superposition for which a solution does not exist. Still, for the elements, for which a solution does exist (we expect $\mathcal{O}(1)$ of these), we perform $\mathcal{O}(\sqrt{|L|})$ Grover iterations during the "inner" Grover procedure, and during the "outer" procedure these "good" elements contribute a $\mathcal{O}(1)$ factor to the running time. Therefore formally, each $|L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})|$ in Eq. (9) should be changed to $\max\{1, |L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})|\}$. Alternatively, one can enforce that intermediate lists are of size greater than 1 by choosing the target configuration appropriately.

Algorithm 4.1 Quantum algorithm for the Configuration Problem

Input: $L_1, \ldots, L_k-$ lists of vectors from S^{d-1}, target configuration $C_{i,j} = \langle \mathbf{x}_i \,, \mathbf{x}_j \rangle \in \mathbb{R}^{k\times k}-$ a Gram matrix, $\varepsilon > 0$.
Output: $L_{\text{out}}-$ list of k-tuples $(\mathbf{x}_1, \ldots, \mathbf{x}_k) \in L_1 \times \cdots \times L_k$, s.t. $|\langle \mathbf{x}_i \,, \mathbf{x}_j \rangle - C_{ij}| \leq \varepsilon$ for all i, j.

```
 1: $L_{\text{out}} \leftarrow \emptyset$
 2: for all $\mathbf{x}_1 \in L_1$ do
 3:     Prepare the state $|\Psi_{L_2}\rangle \otimes \ldots \otimes |\Psi_{L_k}\rangle$
 4:     for all $i = 2 \ldots k-1$ do
 5:         Run Grover's on the $i^{\text{th}}$ register with the checking function $f_{[i-1],i}$ to transform the state $|\Psi_{L_i}\rangle$ to the state $|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})}\rangle$.
 6:     Run Grover's on the $k^{\text{th}}$ register with the checking function $f_{[k-2],k}$ to transform the state $|\Psi_{L_k}\rangle$ to the state $|\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle$.
 7:     Let $\mathcal{A}$ be unitary that implements steps 3–6, i.e.
            $\mathcal{A}\,|\mathbf{0}^{\otimes k}\rangle \rightarrow |\Psi_F\rangle$.
 8:     Run amplitude amplification using the unitary $-\mathcal{A}R\mathcal{A}^{-1}O_g$, where $g$ is defined in Eq. (8).
 9:     Measure all the registers, obtain a tuple $(\mathbf{x}_2, \ldots, \mathbf{x}_k)$.
10:     if $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ satisfies $C$ then
11:         $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\mathbf{x}_1, \ldots, \mathbf{x}_k)\}$.
```

The procedure we have just described is summarised in Algorithm 4.1. If we want to use this algorithm to solve the Approximate k-List problem (Definition 1), we additionally require that the number of output solutions is equal to the size of the input lists. Using the results of Theorem 4, we can express the complexity of Algorithm 4.1 for the Approximate k-List problem via the determinant of the target configuration C and its minors.

Theorem 6. *Given input $L_1, \ldots, L_k \subset \mathsf{S}^{d-1}$ and a configuration $C \in \mathscr{C}$, such that Eq. (2) holds, Algorithm 4.1 solves the Approximate k-List problem in time*

$$T_{\text{k-List}} = \widetilde{\mathcal{O}}\left(\left(\left(\frac{1}{\det(C)}\right)^{\frac{k+1}{2(k-1)}} \cdot \sqrt{\det(C[1\ldots k-1])}\right)^{d/2}\right) \tag{10}$$

using $M_{\text{k-List}} = \widetilde{\mathcal{O}}\left(\left(\frac{1}{\det(C)}\right)^{\frac{d}{2(k-1)}}\right)$ *classical memory and* $\operatorname{poly}(d)$ *quantum memory with success probability at least* $1 - 2^{-\Omega(d)}$.

Proof. From Theorem 4, the input lists $L_1, \ldots, L_k$ should be of sizes $|L| = \widetilde{\mathcal{O}}\left(\left(\frac{1}{\det(C)}\right)^{\frac{d}{2(k-1)}}\right)$ to guarantee a sufficient number of solutions. This determines the requirement for classical memory. Furthermore, since all intermediate lists are stored in the superposition, we require quantum registers of size $\operatorname{poly}(d)$.

Next, we can simplify the expression for $T^{\text{Q}}_{\text{BLS}}$ given in Eq. (9) by noting that $|L_2(\mathbf{x}_1)| \geq |L_3(\mathbf{x}_1, \mathbf{x}_2)| \geq \ldots \geq |L_{k-1}(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})| = |L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|$. The dominant term in the sum appearing in Eq. (9) is $\sqrt{\left(\frac{|L_k|}{|L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})|}\right)}$.

From Theorem 5, the product $\sqrt{|L_2(\mathbf{x}_1)| \cdot \ldots \cdot |L_{k-1}(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|}$ in Eq. (9) can be simplified to $|L|^{\frac{k-2}{2}} \left(\sqrt{\det(C[1 \ldots k-1])}\right)^{d/2}$, from where we arrive at the expression for $T_{\text{k-List}}$ as in the statement.

The success probability of Algorithm 4.1 is determined by the success probability of the amplitude amplification run in Step 8. For this we consider the precise form of the state $|\Psi_F\rangle$ given in Eq. (6). This state is obtained by running $k-1$ (sequential) Grover algorithms. Each tensor $|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})}\rangle$ in this state is a superposition

$$|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})}\rangle = \sqrt{\frac{1-\epsilon_i}{|L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})|}} \sum_{\mathbf{x} \in L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})} |\mathbf{x}\rangle + \sqrt{\frac{\epsilon_i}{|L_i \setminus L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})|}} \sum_{\mathbf{x} \in L_i \setminus L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})} |\mathbf{x}\rangle,$$

where $\epsilon_i < \frac{|L_i(\mathbf{x}_1,\ldots,\mathbf{x}_i)|}{|L_i|} \leq 2^{-\Omega(d)}$. The first inequality comes from the success probability of Grover's algorithm, Theorem 1, the second inequality is due to the fact that all lists on a "lower" level are exponentially smaller than lists on a "higher" level, see Theorem 5. Therefore, the success probability of the amplitude amplification is given by $\prod_{i=2}^{k-1} \frac{1-\epsilon_i}{|L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})|} \cdot \frac{1-\epsilon_k}{|L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})|} \geq (1 - 2^{-\Omega(d)}) \prod_{i=2}^{k-1} |L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})|^{-1}$. According to Theorem 2, after performing $\mathcal{O}\left(\prod_{i=2}^{k} |L_i(\mathbf{x}_1, \ldots, \mathbf{x}_i)| \, |L_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-2})|\right)$ amplitude amplification iterations, in Step 9 we measure a "good" $(\mathbf{x}_2, \ldots, \mathbf{x}_k)$ with probability at least $1 - 2^{-\Omega(d)}$. □

4.1 Quantum version of the Configuration search algorithm from [HKL18]

The main difference between the two algorithms for the configuration problem – the algorithm due to Bai–Laarhoven–Stehlé [BLS16] and due to Herold–Kirshanova–Laarhoven [HKL18] – is that the latter constructs intermediate filtered lists, Figure 2. We use quantum enumeration to construct and classically store these lists.

For a fixed $\mathbf{x}$, quantum enumeration repeatedly applies Grover's algorithm to an input list L_i, where each application returns a random vector from the filtered list $L_i(\mathbf{x})$ with probability greater than $1-2^{-\Omega(d)}$. The quantum complexity of obtaining one vector from $L_i(\mathbf{x})$ is $\mathcal{O}\left(\sqrt{\frac{|L_i|}{|L_i(\mathbf{x})|}}\right)$. We can also check that the returned vector belongs to $L_i(\mathbf{x})$ by checking its inner product with $\mathbf{x}$. Repeating this process $\widetilde{\mathcal{O}}(|L_i(\mathbf{x})|)$ times, we obtain the list $L_i(\mathbf{x})$ stored classically in time $\widetilde{\mathcal{O}}(\sqrt{|L_i| \cdot |L_i(\mathbf{x})|})$. The advantage of constructing the lists $L_i(\mathbf{x})$ is that we can now efficiently prepare the state $|\Psi_{L_2(\mathbf{x})}\rangle \otimes \ldots \otimes |\Psi_{L_k(\mathbf{x})}\rangle$ (cf. Line 3 in Algorithm 4.1) and run amplitude amplification on the states $|\Psi_{L_i(\mathbf{x})}\rangle$ rather than on $|\Psi_{L_i}\rangle$. This may give a speed up if the complexity of the Steps 3–11 of Algorithm 4.1, which is of order $\widetilde{\mathcal{O}}(T^{\mathrm{Q}}_{\mathrm{BLS}}/\,|L_1|)$, dominates the cost of quantum enumeration, which is of order $\widetilde{\mathcal{O}}(\sqrt{|L_i| \cdot |L_i(\mathbf{x})|})$. In general, we can continue creating the "levels" as in [HKL18] (see Figure 2b) using quantum enumeration and at some level switch to the quantum BLS style algorithm. For example, for some level $1 < j \leq k-1$, we apply quantum enumeration to obtain $L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{j-1})$ for all $i > j$. Then for all $(j-1)$-tuples $(\mathbf{x}_1, \ldots, \mathbf{x}_{j-1}) \in L_1 \times \ldots \times L_{j-1}(\mathbf{x}_1, \ldots, \mathbf{x}_{j-2})$, apply Grover's algorithm as in steps 3–11 of Algorithm 4.1 but now to the states $|\Psi_{L_j(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle \otimes \ldots \otimes |\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$. Note that since we have these lists stored in memory, we can efficiently create this superposition. In this way we obtain a quantum "hybrid" between the HKL and the BLS algorithms: until some level j, we construct the intermediate lists using quantum enumeration, create superpositions over all the filtered lists at level j for some fixed values $\mathbf{x}_1, \ldots, \mathbf{x}_{j-1}$, and apply Grover's algorothm to find (if it exists) the $(k-j+1)$ tuple $(\mathbf{x}_j, \ldots, \mathbf{x}_k)$. Pseudocode for this approach is given in Algorithm 4.2.

Let us now analyse Algorithm 4.2. To simplify notation, we denote $L_i^{(j)} = L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{j-1})$ for all $i \geq j$, letting $L_i^{(1)}$ be the input lists L_i (so the upper index denotes the level of the list). All $\mathcal{O}$ notations are omitted. Each quantum enumeration of $L_i^{(j)}$ from $L_i^{(j-1)}$ costs $\sqrt{\left|L_i^{(j-1)}\right| \left|L_i^{(j)}\right|}$. On level $1 \leq \ell \leq j-1$, we repeat such an enumeration $\prod_{r=1}^{\ell-1} \left|L_r^{(r)}\right|$ times to create the intermediate lists, once for each $(\mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1})$. Once the lists $L_i^{(j)}$, $i \geq j$, are constructed, Grover's algorithm gives the state $|\Psi_{L_j^{(j)}}\rangle \ldots |\Psi_{L_{k-1}^{(k-1)}}\rangle\, |\Psi_{L_k^{(k-1)}}\rangle$ in time $\left(\sqrt{\frac{\left|L_{j+1}^{(j)}\right|}{\left|L_{j+1}^{(j+1)}\right|}} + \ldots + \sqrt{\frac{\left|L_{k-1}^{(j)}\right|}{\left|L_{k-1}^{(k-1)}\right|}} + \sqrt{\frac{\left|L_k^{(j)}\right|}{\left|L_k^{(k-1)}\right|}}\right)$ (Steps 11–12 in Algorithm 4.2). On Step 14 the unitary $\mathcal{A}$ must be executed $\sqrt{\left|L_j^{(j)}\right| \cdot \ldots \cdot \left|L_{k-1}^{(k-1)}\right| \cdot \left|L_k^{(k-1)}\right|}$ times to ensure that the measurement of the system gives the "good" tuple $(\mathbf{x}_j, \ldots, \mathbf{x}_k)$. Such tuples may not exist: for $j \geq 3$, i.e. for *fixed* $\mathbf{x}_1, \mathbf{x}_2$, we expect to have less than 1 such tuples. So most of the time, the measurement will return a random $(k-j+1)$-tuple, which we classically check against the target configuration C.

Algorithm 4.2 Hybrid quantum algorithm for the Configuration Problem

```
Input: $L_1, \ldots, L_k$, lists of vectors from $\mathsf{S}^{d-1}$, target configuration $C_{i,j} = \langle \mathbf{x}_i , \mathbf{x}_j \rangle \in \mathbb{R}^{k \times k}$, $\varepsilon > 0$, $2 \leq j \leq k-1$, level we construct the intermediate filtered lists until.
Output: $L_{\text{out}}$– list of $k$-tuples $(\mathbf{x}_1, \ldots, \mathbf{x}_k) \in L_1 \times \cdots \times L_k$, s.t. $|\langle \mathbf{x}_i , \mathbf{x}_j \rangle - C_{ij}| \leq \varepsilon$ for all $i, j$.
 1: $L_{\text{out}} \leftarrow \emptyset$
 2: for all $\mathbf{x}_1 \in L_1$ do
 3:     Use quantum enumeration to construct $L_i(\mathbf{x}_1)$ for $\forall i \geq 2$
 4:     for all $\mathbf{x}_2 \in L_2(\mathbf{x}_1)$ do
 5:         Use quantum enumeration to construct $L_i(\mathbf{x}_1, \mathbf{x}_2)$, $\forall i \geq 3$
 6:             $\ddots$
 7:             for all $\mathbf{x}_{j-1} \in L_{j-1}(\mathbf{x}_1, \ldots, \mathbf{x}_{j-2})$ do
 8:                 Use quantum enumeration to construct $L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{j-1})$, $\forall i \geq j$
 9:             Prepare the state $|\Psi_{L_j(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle \otimes \ldots \otimes |\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$
10:             for all $i = j+1 \ldots k-1$ do
11:                 Run Grover's on the $i^{\text{th}}$ register with the checking function $f_{[i-1],i}$ to transform the state $|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$ to the state $|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})}\rangle$.
12:             Run Grover's on the $k^{\text{th}}$ register with the checking function $f_{[k-2],k}$ to transform the state $|\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$ to the state $|\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle$.
13:             Let $\mathcal{A}$ be unitary that implements Steps 9–12, i.e.
                    $\mathcal{A}\,|\mathbf{0}^{\otimes(k-j+1)}\rangle \rightarrow |\Psi_{L_j(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle \otimes |\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle$
14:             Run amplitude amplification using the unitary $-\mathcal{A}R\mathcal{A}^{-1}O_g$, where $g$ is defined in Eq. (8).
15:             Measure all the registers, obtain a tuple $(\mathbf{x}_j, \ldots, \mathbf{x}_k)$.
16:             if $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ satisfies $C$ then
17:                 $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\mathbf{x}_1, \ldots, \mathbf{x}_k)\}$.
```

Overall, given on input a level j, the runtime of Algorithm 4.2 is

$$T^{\mathrm{Q}}_{\text{Hybrid}}(j) = \max_{1 \leq \ell \leq j-1} \left\{ \prod_{r=1}^{\ell-1} \left| L_r^{(r)} \right| \cdot \max_{\ell \leq i \leq k} \left\{ \sqrt{\left| L_i^{(\ell)} \right| \left| L_i^{(\ell+1)} \right|} \right\}, \right.$$
$$\prod_{r=1}^{j-1} \left| L_r^{(r)} \right| \left(\sqrt{\frac{\left| L_{j+1}^{(j)} \right|}{\left| L_{j+1}^{(j+1)} \right|}} + \ldots + \sqrt{\frac{\left| L_{k-1}^{(j)} \right|}{\left| L_{k-1}^{(k-1)} \right|}} + \sqrt{\frac{\left| L_k^{(j)} \right|}{\left| L_k^{(k-1)} \right|}} \right)$$
$$\left. \cdot \sqrt{\left| L_j^{(j)} \right| \cdot \ldots \left| L_{k-1}^{(k-1)} \right| \cdot \left| L_k^{(k-1)} \right|} \right\}. \tag{11}$$

Similar to Eq. (9), all the list sizes in the above formula are assumed to be greater than or equal to 1. If, for a certain configuration it happens that the expected size of a list is less than 1, it should be replaced with 1 in this expression. The above complexity can be expressed via the determinant and subdeterminants of the target configuration C using Theorem 5. An optimal value of j for a given

C can be found using numerical optimisations by looking for j that minimises Eq. (11).

Speed-ups with nearest neighbour techniques. We can further speed up the creation of filtered lists in both Algorithms 4.1 and 4.2 with a quantum version of nearest neighbour search. In particular, in Appendix B we describe a locality sensitive filtering (LSF) technique (first introduced in [BDGL16]) in the quantum setting, extending the idea of Laarhoven [Laa15] to $k > 2$.

Numerical optimisations. We performed numerical optimisations for the target configuration C which minimises the runtime of the two algorithms for the configuration problem given in this section. The upper part of Table 2 gives time optimal c for Eq. (10) and the c' of the corresponding memory requirements for various k. These constants decrease with k and, eventually, those for time become close to the value 0.2989. The explanation for this behaviour is the following: looking at Eq. (9) the expression decreases when the lists $L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})$ under the square root become smaller. When k is large enough, in particular, once $k \geq 6$, there is a target configuration that ensures that $|L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})|$ are of expected size 1 for levels $i \geq 4$. So for $k \geq 6$, under the observation that the maximal value in the sum appearing in Eq. (9) is attained by the last summand, the runtime of Algorithm 4.1 becomes $T^{\mathrm{Q}}_{\mathrm{BLS}} = |L_1|^{3/2} \cdot \sqrt{|L_2(\mathbf{x}_1)|\,|L_3(\mathbf{x}_1, \mathbf{x}_2)|}$. The list sizes can be made explicit using Eq. (3) when a configuration C is such that $|L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})|$ are of expected size 1. Namely, for $k \geq 6$ and for configuration C that minimises the runtime exponent, Eq. (9) with the help of Eq. (3) simplifies to $\left(\left(\frac{1}{\det C}\right)^{\frac{5}{2(k-1)}} \sqrt{\det C[1,2,3]}\right)^{d/2}$.

The optimal runtime exponents for the hybrid, Algorithm 4.2, with $j = 2$ are given in the middle part of Table 2. Experimentally, we establish that $j = 2$ is optimal for small values of k and that this algorithm has the same behaviour for large values of k as Algorithm 4.1. The reason is the following: for the runtime optimal configuration C the intermediate lists on the same level increase in size "from left to right", i.e. $|L_2(\mathbf{x}_1)| \leq |L_3(\mathbf{x}_1)| \leq \ldots, \leq |L_k(\mathbf{x}_1)|$. It turns out that $|L_k(\mathbf{x}_1)|$ becomes almost $|L_k|$ (i.e. the target inner product is very close to 0), so quantumly enumerating this list brings no advantage over Algorithm 4.1 where we use the initial list L_k, of essentially the same size, in Grover's algorithm.

5 Quantum Configuration Search via k-Clique Listing

In this section we introduce a distinct approach to finding solutions of the configuration problem, Definition 3, via k-clique listing in graphs. We achieve this by repeatedly applying k-clique finding algorithms to the graphs. Throughout this section we assume that $L_1 = \cdots = L_k = L$. We first solve the configuration problem with $k = 3$, C the balanced configuration with all off diagonals equal to $-1/3$ and the size of L determined by Eq. (2). We then adapt the idea to the case for general k. In Appendix C.1 we give the $k = 4$ balanced case to elucidate

k	2	3	4	5	6	...	28	29	30
Quantum version of [BLS16] Algorithm 4.1									
Time	0.3112	0.3306	0.3289	0.3219	0.3147	...	0.29893	0.29893	0.29893
Space	0.2075	0.1907	0.1796	0.1685	0.1596	...	0.1395	0.1395	0.1395
Quantum Hybrid version of [BLS16,HKL18] Algorithm 4.2									
Time	0.3112	0.3306	0.3197	0.3088	0.3059	...	0.29893	0.29893	0.29893
Space	0.2075	0.1907	0.1731	0.1638	0.1595	...	0.1395	0.1395	0.1395
Low memory Quantum Hybrid version of [BLS16,HKL18] Algorithm 4.2									
Time	0.3112	0.3349	0.3215	0.3305	0.3655	...	0.6352	0.6423	0.6490
Space	0.2075	0.1887	0.1724	0.1587	0.1473	...	0.0637	0.0623	0.0609

Table 2: Asymptotic complexity exponents for the approximate k-List problem, base 2. The top part gives optimised runtime exponents and the corresponding memory exponents for Algorithm 4.1. These are the results of the optimisation (minimisation) of the runtime expression given in Eq. (10). The middle part gives the runtime and memory exponents for Algorithm 4.2, again optimising for time, with $j = 2$, i.e. when we use quantum enumeration to create the second level lists $L_i(\mathbf{x}_1)$, $i \geq 2$. The bottom part gives the exponents for Algorithm 4.2 with $j = 2$ in the memory optimal setting.

the jump to the general k case, and in Appendix C.2 the case for general k with unbalanced configurations.

Let $G = (V, E)$ be an undirected graph with known vertices and an oracle $O_G \colon V^2 \to \{\texttt{True}, \texttt{False}\}$. On input $(\mathbf{x}_1, \mathbf{x}_2) \in V^2$, O_G returns `True` if $(\mathbf{x}_1, \mathbf{x}_2) \in E$ and `False` otherwise. A k-clique is $\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ such that $O_G(\mathbf{x}_i, \mathbf{x}_j) = \texttt{True}$ for $i \neq j$. Given k in the balanced case, $(\mathbf{x}_i, \mathbf{x}_j) \in E \iff |\langle \mathbf{x}_i , \mathbf{x}_j \rangle + 1/k| \leq \varepsilon$ for some $\varepsilon > 0$. In the unbalanced case $(\mathbf{x}_i, \mathbf{x}_j) \in E \iff |\langle \mathbf{x}_i, \mathbf{x}_j \rangle - C_{i,j}| \leq \varepsilon$ (considered in Appendix C.2). In both cases, the oracle computes a d dimensional inner product and compares the result against the target configuration. Throughout we let $|V| = n$ and $|E| = m$.

5.1 The Triangle Case

We start with the simple triangle finding algorithm of [BdWD+01]. A triangle is a 3-clique. Given the balanced configuration and $k = 3$ on S^{d-1}, we have

$$n = |L| = \widetilde{\mathcal{O}}\left((3\sqrt{3}/4)^{d/2} \right), \; m = |L|\,|L(\mathbf{x}_1)| = \widetilde{\mathcal{O}}\left(n^2 (8/9)^{d/2} \right) \tag{12}$$

by Eq. (2) and Theorem 5 respectively,[6] We expect $\Theta(n)$ triangles to be found [HKL18]. The algorithm of [BdWD+01] consists of three steps:

[6] As we are in the balanced configuration case, and our input lists are identical, Theorem 5 has no dependence on j.

1. Use Grover's algorithm to find any edge $(\mathbf{x}_1, \mathbf{x}_2) \in E$ among all potential $\mathcal{O}(n^2)$ edges.
2. Given an edge $(\mathbf{x}_1, \mathbf{x}_2)$ from Step 1, use Grover's algorithm to find a vertex $\mathbf{x}_3 \in V$, such that $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ is a triangle.
3. Apply amplitude amplification on Steps 1–2.

Note that the algorithm searches for any triangle in the graph, not a fixed one. To be more explicit about the use of the oracle O_G, below we describe a circuit that returns a triangle. Step 1 takes the state $\frac{1}{n} \sum_{(\mathbf{x}_1,\mathbf{x}_2) \in V^2} |\mathbf{x}_1\rangle \otimes |\mathbf{x}_2\rangle$ and applies $\mathcal{O}(\sqrt{n^2/m})$ times the Grover iteration given by $-H^{\otimes 2\bar{d}} R H^{\otimes 2\bar{d}} O_G$. The output is the state $\sqrt{\frac{\epsilon}{n^2-m}} \sum_{(\mathbf{x}_1,\mathbf{x}_2) \notin E} |\mathbf{x}_1\rangle \otimes |\mathbf{x}_2\rangle + \sqrt{\frac{1-\epsilon}{m}} \sum_{(\mathbf{x}_1,\mathbf{x}_2) \in E} |\mathbf{x}_1\rangle \otimes |\mathbf{x}_2\rangle$, where ϵ represents the probability of failure. We disregard this as in the proof of Theorem 6. We then join with a uniform superposition over the vertices to create the state $\frac{1}{\sqrt{m}} \sum_{(\mathbf{x}_1,\mathbf{x}_2) \in E} |\mathbf{x}_1\rangle \otimes |\mathbf{x}_2\rangle \otimes \frac{1}{\sqrt{n}} \sum_{\mathbf{x}_3 \in V} |\mathbf{x}_3\rangle$ and apply $-H^{\otimes 3\bar{d}} R H^{\otimes 3\bar{d}} O_G^{\Delta}$ $\mathcal{O}(\sqrt{n})$ times. This oracle O_G^{Δ} outputs `True` on a triple from V^3 if each pair of vertices has an edge. We call the final state $|\Psi_F\rangle$. Let $\mathcal{A} |\mathbf{0}^{\otimes 3}\rangle \rightarrow |\Psi_F\rangle$, then we apply amplitude amplification with $\mathcal{A}$ repeated some number of times determined by the success probability of $\mathcal{A}$ calculated below.

Given that oracle queries O_G or O_G^{Δ} have some poly(d) cost, we may calculate the time complexity of this method directly from the query complexity. The cost of the first step is $\mathcal{O}(\sqrt{n^2/m})$ and the second step $\mathcal{O}(\sqrt{n})$. From Eq. (12), and that the costs of Step 1 and Step 2 are additive, we see that $\mathcal{O}(\sqrt{n})$ dominates, therefore Steps 1–2 cost $\mathcal{O}(\sqrt{n})$. The probability that Step 2 finds a triangle is the probability that Step 1 finds an edge of a triangle. Given that there are $\Theta(n)$ triangles, this probability is $\Theta(n/m)$, therefore by applying the amplitude amplification in Step 3, the cost of finding a triangle is $\mathcal{O}(\sqrt{m})$.[7]

The algorithm finds one of the n triangles uniformly at random. By the coupon collector's problem we must repeat the algorithm $\widetilde{\mathcal{O}}(n)$ times to find all the triangles. Therefore the total cost of finding all triangles is $\widetilde{\mathcal{O}}(n\sqrt{m}) = \widetilde{\mathcal{O}}(|L|^{3/2}|L(\mathbf{x}_1)|^{1/2}) \approx 2^{0.3349d+o(d)}$ using $2^{0.1887d+o(d)}$ memory. This matches the complexity of Algorithm 4.1 for $k = 3$ in the balanced setting (see Table 2).

5.2 The General *k*-Clique Case

The algorithm generalises to arbitrary constant k. We have a graph with $|L|$ vertices, $|L||L(\mathbf{x}_1)|$ edges, $\ldots$, $|L||L(\mathbf{x}_1)| \ldots |L(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1})|$ i-cliques for $i \in \{3, \ldots, k-1\}$, and $\Theta(|L|)$ k-cliques. The following algorithm finds a k-clique, with $2 \leq i \leq k-1$

[7] Note that this differs from [BdWD+01] as in general either of Step 1 or 2 may dominate and we also make use of the existence of $\Theta(n)$ triangles.

1. Use Grover's algorithm to find an edge $(\mathbf{x}_1, \mathbf{x}_2) \in E$ among all potential $\mathcal{O}(|L|^2)$ edges.

$$\vdots$$

i. Given an i-clique $(\mathbf{x}_1, \ldots, \mathbf{x}_i)$ from step $i-1$, use Grover's algorithm to find a vertex $\mathbf{x}_{i+1} \in V$, such that $(\mathbf{x}_1, \ldots, \mathbf{x}_{i+1})$ is an $(i+1)$-clique.

$$\vdots$$

k. Apply amplitude amplification on Steps 1–$(k-1)$.

The costs of Steps 1–$(k-1)$ are additive. The dominant term is from Step $k-1$, a Grover search over $|L|$, equal to $\mathcal{O}(\sqrt{|L|})$. To determine the cost of finding one k-clique, we need the probability that Steps 1–$(k-1)$ find a k-clique. We calculate the following probabilities, with $2 \le i \le k-2$

1. The probability that Step 1 finds a good edge, that is, an edge belonging to a k-clique.

i. The probability that Step i finds a good $(i+1)$-clique given that Step $i-1$ finds a good i-clique.

In Step 1 there are $\mathcal{O}(|L||L(\mathbf{x}_1)|)$ edges to choose from, $\Theta(|L|)$ of which belong to a k-clique. Thus the success probability of this Step is $\Theta(1/|L(\mathbf{x}_1)|)$. Thereafter, in Step i, given an i-clique $(\mathbf{x}_1, \ldots, \mathbf{x}_i)$ there are $\mathcal{O}(\max\{|L(\mathbf{x}_1, \ldots, \mathbf{x}_i)|, 1\})$ $(i+1)$-cliques on the form $(\mathbf{x}_1, \ldots, \mathbf{x}_i, \mathbf{x}_{i+1})$, $\Theta(1)$ of which are good. The success probability of Steps 1–$(k-1)$ is equal to $\Theta\left(\prod_{i=1}^{k-2} \max\{|L(\mathbf{x}_1, \ldots, \mathbf{x}_i)|, 1\}^{-1}\right)$. By applying amplitude amplification at Step k, we get the cost

$$\mathcal{O}\left(\sqrt{|L|}\sqrt{\prod_{i=1}^{k-2} \max\{|L(\mathbf{x}_1, \ldots, \mathbf{x}_i)|, 1\}}\right),$$

for finding one k-clique. Multiplying the above expression by $\widetilde{\mathcal{O}}(|L|)$ gives the total complexity for finding $\Theta(|L|)$ k-cliques. This matches the complexity of Algorithm 4.1, Eq. (9), for balanced configurations for all k.

In Appendix C.2 we show how to solve the configuration problem with unbalanced configurations using a graph approach, again achieving the same complexity as Algorithm 4.1.

6 Quantum Configuration Search via Triangle Listing

Given the phrasing of the configuration problem as a clique listing problem in graphs, we restrict our attention to the balanced $k=3$ case and appeal to the wide body of recent work on triangle finding in graphs. Let the notation be as in Section 5, and in particular recall Eq. (12) then a triangle represents a solution to the configuration problem.

We note that the operations counted in the works discussed here are queries to an oracle that returns whether an edge exists between two vertices in our graph. While, in the case of [BdWD+01], it is simple to translate this cost into a time complexity, for the algorithms which use more complex quantum data structures [LGN17] it is not. In particular, the costs of computing various auxiliary databases from certain sets is not captured in the total query cost.

The quantum triangle finding works we consider are [BdWD+01,Gal14,LGN17]. In [BdWD+01] a simple algorithm based on nested Grover search and quantum amplitude amplification is given which finds a triangle in $\mathcal{O}(n+\sqrt{nm})$ queries to O_G. For sufficiently sparse graphs G, with sparsity measured as $m=\mathcal{O}(n^c)$ and G becoming more sparse as c decreases, this complexity attains the optimal $\Omega(n)$. This is the algorithm extended in Section 5 for the k-configuration problem. In [Gal14] an algorithm is given that finds a triangle in $\widetilde{\mathcal{O}}(n^{5/4})$ queries to O_G. This complexity has no dependence on sparsity and is the currently best known result for generic graphs. Finally in [LGN17] an interpolation between the two previous results is given as the sparsity of the graph increases.

Theorem 7 ([LGN17, Theorem 1]). *There exists a quantum algorithm that solves, with high probability, the triangle finding problem over graphs of n vertices and m edges with query complexity*

$$\begin{cases} \mathcal{O}(n+\sqrt{nm}) & \text{if } 0 \le m \le n^{7/6} \\ \widetilde{\mathcal{O}}(nm^{1/14}) & \text{if } n^{7/6} \le m \le n^{7/5} \\ \widetilde{\mathcal{O}}(n^{1/6}m^{1/3}) & \text{if } n^{7/5} \le m \le n^{3/2} \\ \widetilde{\mathcal{O}}(n^{23/30}m^{4/15}) & \text{if } n^{3/2} \le m \le n^{13/8} \\ \widetilde{\mathcal{O}}(n^{59/60}m^{2/15}) & \text{if } n^{13/8} \le m \le n^{2}. \end{cases}$$

More specifically it is shown that for $c \in (7/6, 2)$ a better complexity can be achieved than shown in [BdWD+01,Gal14]. Moreover at the end points the two previous algorithms are recovered; [BdWD+01] for $c \le 7/6$ and [Gal14] for $c = 2$. We recall that these costs are in the query model, and that for $c > 7/6$, where we do not recover [BdWD+01], we do not convert them into time complexity.

We explore two directions that follow from the above embedding of the configuration problem into a graph. The first is the most naïve, we simply calculate the sparsity regime (as per [LGN17]) that the graph, constructed as above, lies in and calculate a lower bound on the cost of listing all triangles.

The second splits our list into triples of distinct sublists and considers graphs formed from the union of said triples of sublists. The sublists are parameterised such that the sparsity and the expected number of triangles in these new graphs can be altered.

6.1 Naïve Triangle Finding

With $G=(V,E)$ and n,m as in (12), we expect to have

$$m=\mathcal{O}\left(n^{2+\delta}\right)=\mathcal{O}\left(n^{1.5500}\right),\ \delta=\log(8/9)/\log(3\sqrt{3}/4).$$

Therefore finding a single triangle takes $\widetilde{\mathcal{O}}(n^{23/30}m^{4/15})=\widetilde{\mathcal{O}}\left(n^{1.1799}\right)$ queries to O_G [LGN17]. If, to list the expected $\Theta(n)$ triangles, we have to repeat this algorithm $\widetilde{\mathcal{O}}(n)$ times this leads to a total O_G query complexity of $\widetilde{\mathcal{O}}(n^{2.1799}) = 2^{0.4114d+o(d)}$ which is not competitive with classical algorithms [HK17] or the approach of Section 5.

6.2 Altering the Sparsity

Let n remain as in Eq.(12) and $\gamma \in (0,1)$ be such that we consider $\Gamma = n^{1-\gamma}$ disjoint sublists of L, $\ell_1, \dots, \ell_\Gamma$, each with $n' = n^\gamma$ elements. There are $\mathcal{O}(n^{3(1-\gamma)})$ triples of such sublists, (ℓ_i, ℓ_j, ℓ_k), with i, j, k pairwise not equal and the union of the sublists within one triple, $\ell_{ijk} = \ell_i \cup \ell_j \cup \ell_k$, has size $\mathcal{O}(n')$. Let $G_{ijk} = (\ell_{ijk}, E_{ijk})$ with $(\mathbf{x}_1, \mathbf{x}_2)$ in $\ell_{ijk} \times \ell_{ijk}$, $(\mathbf{x}_1, \mathbf{x}_2) \in E_{ijk} \iff |\langle \mathbf{x}_1, \mathbf{x}_2 \rangle + 1/3| \leq \varepsilon$ as before. Using Theorem 5, each G_{ijk} is expected to have

$$m' = \mathcal{O}\left(|\ell_{ijk}|\,|\ell_{ijk}(x_1)|\right) = \mathcal{O}\left((n')^2 (8/9)^{d/2}\right) = \mathcal{O}\left(n^{2\gamma} (8/9)^{d/2}\right)$$

edges. By listing all triangles in all G_{ijk} we list all triangles in G, and as n is chosen to expect $\Theta(n)$ triangles in G, we have sufficiently many solutions for the underlying k-List problem. We expect, by Theorem 5

$$\begin{aligned}|\ell_{ijk}||\ell_{ijk}(\mathbf{x}_1)||\ell_{ijk}(\mathbf{x}_1, \mathbf{x}_2)| &= |\ell_{ijk}| \left(|\ell_{ijk}|(8/9)^{d/2}\right)\left(|\ell_{ijk}|(2/3)^{d/2}\right) \\ &= \mathcal{O}(n^{3\gamma})(16/27)^{d/2} = \mathcal{O}(n^{3\gamma-2})\end{aligned}$$

triangles per ℓ_{ijk}. We must at least test each ℓ_{ijk} once, even if $\mathcal{O}(n^{3\gamma-2})$ is subconstant. The sparsity of ℓ_{ijk} given γ is calculated as

$$m' = \mathcal{O}\left((n')^{2+\beta(\gamma)}\right), \ \beta(\gamma) = \frac{\log(8/9)}{\gamma \log(3\sqrt{3}/4)}.$$

For given γ the number of ℓ_{ijk} to test is $\mathcal{O}(n^{3(1-\gamma)})$, the number of triangles to list per ℓ_{ijk} is $\mathcal{O}(n^{3\gamma-2})$ – we always perform at least one triangle finding attempt and assume listing them all takes $\widetilde{\mathcal{O}}(n^{3\gamma-2})$ repeats – and we are in the sparsity regime $c(\gamma) = 2 + \beta(\gamma)$ [LGN17]. Let a, b represent the exponents of n', m' respectively[8] in Theorem 7 given by $m' = (n')^{c(\gamma)}$. We therefore minimise, for $\gamma \in (0,1)$, the exponent of n in $\mathcal{O}(n^{3(1-\gamma)}) \cdot \widetilde{\mathcal{O}}(n^{3\gamma-2}) \cdot \widetilde{\mathcal{O}}((n')^a (m')^b)$,

$$3(1-\gamma) + \max\{0, 3\gamma - 2\} + a\gamma + \left(2\gamma + \frac{\log(8/9)}{\log(3\sqrt{3}/4)}\right) b.$$

The minimal query complexity of $n^{1.7298+o(d)} = 2^{0.326d+o(d)}$ is achieved at $\gamma = \frac{2}{3}$.

The above method leaves open the possibility of finding the same triangle multiple times. In particular if a triangle exists in $G_{ij} = (\ell_{ij}, E_{ij})$, with ℓ_{ij} and E_{ij} defined analogously to ℓ_{ijk} and E_{ijk}, then it will be found in G_{ijk} for all k, that is $\mathcal{O}(n^{1-\gamma})$ many times. Worse yet is the case where a triangle exists in $G_i = (\ell_i, E_i)$ where it will be found $\mathcal{O}(n^{2(1-\gamma)})$ times. However, in both cases the total number of rediscoveries of the same triangle does not affect the asymptotic

[8] Note that we are considering G_{ijk} rather than G here, hence the $n \leftrightarrow n', m \leftrightarrow m'$ notation change.

complexity of this approach. Indeed in the ℓ_{ij} case this number is the product $\mathcal{O}(n^{2(1-\gamma)}) \cdot \mathcal{O}(n^{3\gamma} \cdot (8/9)^{d/2}) \cdot \mathcal{O}(n^{1-\gamma}) = \mathcal{O}(n)$, the product of the number of ℓ_{ij}, the number of triangles[9] per ℓ_{ij} and the number of rediscoveries per triangle in ℓ_{ij} respectively. Similarly, this value remains $\mathcal{O}(n)$ in the ℓ_i case and as we are required to list $\mathcal{O}(n)$ triangles the asymptotic complexity remains $\mathcal{O}(n)$.

7 Parallelising Quantum Configuration Search

In this section we deviate slightly from the k-List problem and the configuration framework and target SVP directly. On input we receive $\{\mathbf{b}_1, \ldots, \mathbf{b}_d\} \subset \mathbb{R}^d$, a basis of $\mathcal{L}(B)$. Our algorithm finds and outputs a short vector from $\mathcal{L}(B)$. As in all the algorithms described above, we will be satisfied with an approximation to the shortest vector and with heuristic analysis.

We describe an algorithm that can be implemented using a quantum circuit of width $\widetilde{\mathcal{O}}(N)$ and depth $\widetilde{\mathcal{O}}(\sqrt{N})$, where $N = 2^{0.2075d+o(d)}$. We therefore require our input and output to be less than $\widetilde{\mathcal{O}}(\sqrt{N})$, and if we were to phrase the 2-Sieve algorithm as a 2-List problem we would not be able to read in and write out the data. Our algorithm uses poly(d) classical memory. For the analysis, we make the same heuristic assumptions as in the original 2-Sieve work of Nguyen–Vidick [NV08].

All the vectors encountered by the algorithm (except for the final measurement) are kept in quantum memory. Recall that for a pair of normalised vectors $\mathbf{x}_1, \mathbf{x}_2$ to form a "good" pair, i.e. to satisfy $\|\mathbf{x}_1 \pm \mathbf{x}_2\| \leq 1$, it must hold that $|\langle \mathbf{x}_1 , \mathbf{x}_2 \rangle| \geq \frac{1}{2}$. The algorithm described below is the quantum parallel version of 2-Sieve. Each step is analysed in the subsequent lemmas.

Algorithm 7.1 A parallel quantum algorithm for 2-Sieve

```
Input: $\{\mathbf{b}_1, \ldots, \mathbf{b}_d\} \subset \mathbb{R}^d$ a lattice basis
Output: $\mathbf{v} \in \mathcal{L}(B)$, a short vector from $\mathcal{L}(B)$
1: Set $N \leftarrow 2^{0.2075d+o(d)}$ and set $\lambda = \Theta(\sqrt{d} \cdot \det(B)^{1/d})$ the target length.
2: Generate a list $L_1 \leftarrow \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ of normalised lattice vectors using an efficient
   lattice sampling procedure, e.g. [Kle00].
3: Construct a list $L_2 \leftarrow \{\mathbf{x}'_1, \ldots, \mathbf{x}'_N\}$ such that $|\langle \mathbf{x}_i , \mathbf{x}'_i \rangle| \geq 1/2$ for $\mathbf{x}'_i \in L_1$. If no
   such $\mathbf{x}'_i \in L_1$ exists, set $\mathbf{x}'_i \leftarrow \mathbf{0}$.
4: Construct a list $L_3 \leftarrow \{\mathbf{y}_i : \mathbf{y}_i \leftarrow \min\{\|\mathbf{x}_i \pm \mathbf{x}'_i\|\}$ for all $i \leq N\}$ and normalise its
   elements except for the last iteration.
5: Swap the labels $L_1, L_3$. Reinitialise $L_2$ and $L_3$ to the zero state by transferring
   their contents to auxiliary memory.
6: Repeat Steps 3–5 poly$(d)$ times.
7: Output a vector from $L_1$ of Euclidean norm less than $\lambda$.
```

Several remarks about Algorithm 7.1.

[9] Given that $|\ell_i| = n^\gamma, |\ell_{ij}| = 2n^\gamma, |\ell_{ijk}| = 3n^\gamma$ the expected numbers of triangles differ only by a constant.

1. The bound on the repetition factor on Step 6 is, as in classical 2-Sieve algorithms, appropriately set to achieve the desired norm of the returned vectors. In particular, it suffices to repeat Steps 2–5 poly(d) times [NV08].
2. In classical 2-Sieve algorithms, if $\mathbf{x}_i$ does not have a match $\mathbf{x}'_i$, it is simply discarded. Quantumly we cannot just discard an element from the system, so we keep it as the zero vector. This is why, as opposed to the classical setting, we keep our lists of exactly the same size throughout all the iterations.
3. The target norm λ is appropriately set to the desired length. The algorithm can be easily adapted to output several, say T, short vectors of $\mathcal{L}(B)$ by repeating Step 7 T times.

Theorem 8. *Given on input a lattice basis $\mathcal{L}(B) = \{\mathbf{b}_1, \dots, \mathbf{b}_d\} \subset \mathbb{R}^d$, Algorithm 7.1 heuristically solves the shortest vector problem on $\mathcal{L}(B)$ with constant success probability. The algorithm can be implemented using a uniform family of quantum circuits of width $\widetilde{\mathcal{O}}(N)$ and depth $\widetilde{\mathcal{O}}(\sqrt{N})$, where $N = 2^{0.2075d+o(d)}$.*

We prove the above theorem in several lemmas. Here we only give proof sketches for these lemmas, and defer more detailed proofs to Appendix D. In the first lemma we explain the process of generating a database of vectors of size N having N processors. The main routines, Steps 3–5, are analysed in Lemma 2. Finally, in Step 7 we use Grover's algorithm to amplify the amplitudes of small norm vectors.

Lemma 1. *Step (2) of Algorithm 7.1 can be implemented using a uniform family of quantum circuits of width $\widetilde{\mathcal{O}}(N)$ and depth* $\operatorname{poly}\log(N)$.

Lemma 2. *Steps (3–5) of Algorithm 7.1 can be implemented using a uniform family of quantum circuits of width $\widetilde{\mathcal{O}}(N)$ and depth $\widetilde{\mathcal{O}}(\sqrt{N})$.*

Lemma 3. *Step (7) of the Algorithm 7.1 can be implemented using a uniform family of quantum circuits of width $\widetilde{\mathcal{O}}(N)$ and depth $\widetilde{\mathcal{O}}(\sqrt{N})$.*

Before we present our proofs for the above lemmas, we briefly explain our computational model. We assume that each input vector $\mathbf{b}_i$ is encoded in $\bar{d} = poly(d)$ qubits and we say that it is stored in a single register. We also consider the circuit model and assume we have at our disposal a set of elementary gates – Toffoli, and all 1-qubit unitary gates (including the Hadamard and Pauli X), i.e. a universal gate set that can be implemented efficiently. We further assume that any parallel composition of unitaries can be implemented simultaneously. For brevity, we will often want to interpret (computations consisting of) parallel processes to be running on parallel processors. We emphasise that this is inconsequential to the computation and our analysis. However, thinking this way greatly helps to understand the physical motivation and convey the intuition behind the computation.

Proof sketch of Lemma 1. The idea is to copy the *cell* of *registers*, $|B\rangle$, encoding the basis $B = \{\mathbf{b}_1, \dots, \mathbf{b}_d\}$ to N processors, where each processor is equipped with $\operatorname{poly}\log(N)$ qubits. The state $|B\rangle$ itself is a classical (diagonal) state made of $\bar{d}^2 = \mathcal{O}(\log^2(N))$ qubits. To copy B to all N processors, it takes $\lceil \log(N) \rceil$ steps each consisting of a cascade of CNOT operations.

Each of the processors samples a single $\mathbf{x}_i$ using a randomised sampling algorithm, e.g. [Kle00]. This is an efficient classical procedure that can be implemented by a reversible circuit of $\operatorname{poly}(d)$ depth and width. The exact same circuit can be used to realise the sampling on a quantum processor.

Each processor i, having computed the $\mathbf{x}_i$, now keeps $\mathbf{x}_i$ locally and also copies it to a distinguished cell $\mathtt{L1}$. The state of the system now can be described as

$$|\mathbf{x}_1\rangle_{\mathtt{P}_1} |\mathbf{x}_2\rangle_{\mathtt{P}_2} \ldots |\mathbf{x}_N\rangle_{\mathtt{P}_N} |\mathbf{x}_1, \mathbf{x}_2 \ldots \mathbf{x}_N\rangle_{\mathtt{L1}} |\text{ancilla}\rangle$$

where $\mathtt{P}_i$ is the register in possession of processor i. The total depth of the circuit is $\mathcal{O}(\log(N))$ to copy plus $\operatorname{poly}\log(N)$ to sample plus $\mathcal{O}(1)$ to copy to the list L_1. Each operation is carried out by N processors and uses $\operatorname{poly}\log(N)$ qubits. Thus the total depth of a quantum circuit implementing Step (2) is $\operatorname{poly}\log(N)$ and its width is $\widetilde{\mathcal{O}}(N)$. $\square$

Proof sketch of Lemma 2. The key idea to construct the list L_2 is to let each processor $\mathtt{P}_i$, which already has a copy of $|\mathbf{x}_i\rangle, \mathbf{x}_i \in L_1$, search through L_1 (now stored in the distinguished cell $\mathtt{L1}$) to find a vector $\mathbf{x}_i'$ such that $|\langle \mathbf{x}_i \, , \mathbf{x}_i' \rangle| \geq 1/2$ (if no such $\mathbf{x}_i' \in L_1$, set $\mathbf{x}_i' = 0$). The key ingredient is to parallelise this search, i.e. let all processors do the search at the same time. The notion of parallelisation is however only a (correct) interpretation of the operational meaning of the unitary transformations. It is important to stress that we make no assumptions about how data structures are stored, accessed and processed, beyond what is allowed by the axioms of quantum theory and the framework of the circuit model.

For each processor i, we define a function $f_i(\mathbf{y}) = 1$ if $|\langle \mathbf{x}_i \, , \mathbf{y} \rangle| \geq 1/2$ and 0 otherwise; and let W_f and D_f be the maximal width and depth of a unitary implementing any f_i. It is possible to implement a quantum circuit of $\widetilde{\mathcal{O}}(N \cdot W_f)$ width and $\widetilde{\mathcal{O}}(\sqrt{N} D_f)$ depth that can in parallel find solutions to all $f_i, 1 \leq i \leq N$ [BBG+13]. This quantum circuit searches through the list in parallel, i.e. each processor can simultaneously access the memory and search. Note, f_i is really a reduced transformation. The "purification" of f_i is a two parameter function $f \colon X \times X \to \{0, 1\}$. However, in each processor i, one of the inputs is "fixed and hardcoded" to be $\mathbf{x}_i$. The function f itself admits an efficient implementation in the size of the inputs, since this is the inner product function and also has a classical reversible circuit consisting of Toffoli and NOT gates. Once the search is done, it is expected with probability greater than $1 - 2^{-\Omega(d)}$ that each processor i will have found an index j_i, s.t. $|\langle \mathbf{x}_i \, , \mathbf{x}_{j_i} \rangle| \geq 1/2$, $\mathbf{x}_i, \mathbf{x}_{j_i} \in L_1$. One can always check if the processor found a solution, otherwise the search can be repeated a constant number of times. If none of the searches found a "good" j_i, we set $\mathbf{x}_{j_i} = \mathbf{0}$. Else, if any of the searches succeed, we keep that index j_i.

At this point we have a virtual list L_2, which consists of all indices j_i. We create a list L_3 in another distinguished cell, by asking each processor to compute $\mathbf{y}_i^+ = \mathbf{x}_i + \mathbf{x}_{j_i}$ and $\mathbf{y}_i^- = \mathbf{x}_i - \mathbf{x}_{j_i}$ and copy into the i^{th} register the shorter of

$\mathbf{y}_i^+$ and $\mathbf{y}_i^-$, in the Euclidean length. The state of the system now is,

$$|\mathbf{x}_1\rangle_{P1}\dots|\mathbf{x}_N\rangle_{PN}\,|\mathbf{y}_1\rangle_{P1}\dots|\mathbf{y}_L\rangle_{PN}\,|\mathbf{x}_1\dots\mathbf{x}_N\rangle_{L1}\,|\mathbf{y}_1\dots\mathbf{y}_N\rangle_{L3}\,|\text{ancilla}\rangle\,.$$

A swap between qubits say, S and R, is just $CNOT_{SR}\circ CNOT_{RS}\circ CNOT_{SR}$, and thus the Swap in Step 5 between L_1 and L_2 can be done with a depth 3 circuit. Finally reinitialise the lists L_2 and L_3 by swapping them with two registers of equal size that are all initialised to zero. This unloads the data from the main memories ($\texttt{L2}, \texttt{L3}$) and enables processors to reuse them for the next iteration.

The total depth of the circuit is $\widetilde{\mathcal{O}}(\sqrt{N})$ (to perform the parallel search for "good" indices j_i), $\operatorname{poly}\log N$ (to compute the elements of the new list L_3 and copy them), and $\mathcal{O}(1)$ (to swap the content in memory registers). Thus, in total we have constructed a circuit of $\widetilde{\mathcal{O}}(\sqrt{N})$ depth and $\widetilde{\mathcal{O}}(N)$ width. □

Proof sketch of Lemma 3. Given a database of vectors of size N and a norm threshold λ, finding a vector from the database of Euclidean norm less than λ amounts to Grover's search over the database. It can be done with a quantum circuit of depth $\widetilde{\mathcal{O}}(\sqrt{N})$. It could happen that the threshold λ is set to be too small, in which case Grover's search returns a random element form the database. In that case, we repeat the whole algorithm with an increased value for λ. After $\Theta(1)$ repetitions, we heuristically obtain a short vector from $\mathcal{L}(B)$. □

Proof sketch of Theorem 8. As established from the lemmas above, each of Step 2, Steps 3–5 and Step 7 can be realised using a family of quantum circuits of depth and width (at most) $\widetilde{\mathcal{O}}(\sqrt{N})$ and $\widetilde{\mathcal{O}}(N)$ respectively. However, Steps 3–5 run $\mathcal{O}(\text{poly}(d))$ times, thus the total depth of the circuit now goes up by at most a multiplicative factor of $\mathcal{O}(\text{poly}(d)) = \mathcal{O}(\operatorname{poly}\log(N))$. The total depth and width of a circuit implementing Algorithm 7.1 remains as $\widetilde{\mathcal{O}}(\sqrt{N})$ and $\widetilde{\mathcal{O}}(N)$ respectively as $\widetilde{\mathcal{O}}$ notation suppresses subexponential factors. This concludes the proof. □

7.1 Distributed Configuration Search: Classical Analogue

Algorithm 7.1 should be compared with a classical model where there are $N = 2^{0.2075d+o(d)}$ computing nodes, each equipped with poly(d) memory. It suffices for these nodes to have a nearest neighbour architecture, where node i is connected to nodes $i-1$ and $i+1$, and arranged like beads in a necklace. We cost one time unit for poly(d) bits sent from any node to an adjacent node. A comparable distributed classical algorithm would be where each node, i, receives the basis B and samples a vector $\mathbf{v}_i$. In any given round, node i sends $\tilde{\mathbf{v}}_i$ to node $i+1$ and receives $\tilde{\mathbf{v}}_{i-1}$ from node $i-1$ (in the first round $\tilde{\mathbf{v}}_i := \mathbf{v}_i$). Then each node checks if the vector pair $(\mathbf{v}_i, \tilde{\mathbf{v}}_{i-1})$ gives a shorter sum or difference. If yes, it computes $\mathbf{v}_i^{(2)} = \min\{\mathbf{v}_i \pm \tilde{\mathbf{v}}_{i-1}\}$ and sets $\tilde{\mathbf{v}}_i := \mathbf{v}_{i-1}$. After N rounds every node i has compared their vector $\mathbf{v}_i$ with all N vectors sampled. The vectors $\mathbf{v}_i$ can be discarded and the new round begins with $\mathbf{v}_i^{(2)}$ being the new vector. The process is repeated poly(d) many times leading to $\mathcal{O}(N) \cdot \text{poly}(d)$ time steps. Thus this distributed algorithm needs $\widetilde{\mathcal{O}}(N) = 2^{0.2075d+o(d)}$ time.

Acknowledgements. Most of this work was done while EK was at ENS de Lyon, supported by ERC Starting Grant ERC-2013-StG-335086-LATTAC and by the European Union PROMETHEUS project (Horizon 2020 Research and Innovation Program, grant 780701). EM is supported by the Swedish Research Counsel (grant 2015-04528) and the Swedish Foundation for Strategic Research (grant RIT17-0005). EWP is supported by the EPSRC and the UK government (grant EP/P009301/1). SRM is supported by the Clarendon Scholarship, Google-DeepMind Scholarship and Keble Sloane–Robinson Award.

We are grateful to the organisers of the Oxford Post-Quantum Cryptography Workshop held at the Mathematical Institute, University of Oxford, March 18–22, 2019, for arranging the session on Quantum Cryptanalysis, where this work began. We would like to acknowledge the fruitful discussions we had with Gottfried Herold during this session.

Finally, we would like to thank the AsiaCrypt'19 reviewers, whose constructive comments helped to improve the quality of this paper.

References

AD97. Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 284–293, 1997.

ADH$^+$19. Martin Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *Advances in Cryptology – EUROCRYPT 2019*, pages 717–746, 2019.

ADRSD15. Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in 2n time using discrete gaussian sampling: Extended abstract. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 733–742, New York, NY, USA, 2015. ACM.

AGJO$^+$15. Srinivasan Arunachalam, Vlad Gheorghiu, Tomas Jochym-O'Connor, Michele Mosca, and Priyaa Varshinee Srinivasan. On the robustness of bucket brigade quantum RAM. *New Journal of Physics*, 17(12):123010, dec 2015.

AKS01. Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 601–610, 2001.

ANS18. Yoshinori Aono, Phong Q. Nguyen, and Yixin Shen. Quantum lattice enumeration and tweaking discrete pruning. In *Advances in Cryptology – ASIACRYPT 2018*, pages 405–434, 2018.

BBG$^+$13. Robert Beals, Stephen Brierley, Oliver Gray, Aram W Harrow, Samuel Kutin, Noah Linden, Dan Shepherd, and Mark Stather. Efficient distributed quantum computing. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 469(2153):20120686, 2013.

BBHT98. Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(4–5):493–505, 1998.

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 10–24, 2016.

BdWD$^+$01. Harry Buhrman, Ronald de Wolf, Christoph Dürr, Mark Heiligman, Peter Høyer, Frédéric Magniez, and Miklos Santha. Quantum algorithms for element distinctness. In *Proceedings of the 16th Annual Conference on Computational Complexity*, CCC '01, pages 131–137, Washington, DC, USA, 2001. IEEE Computer Society.

BGJ14. Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. *LMS Journal of Computation and Mathematics*, 17(A):49–70, 2014.

BHMT02. Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Quantum Computation and Quantum Information: A Millennium Volume*, 305:53–74, 2002. Earlier version in arxiv:quant-ph/0005055.

BHT97. Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum algorithm for the collision problem. *ACM SIGACT News (Cryptology Column)*, 28, 1997.

BLS16. Shi Bai, Thijs Laarhoven, and Damien Stehlé. Tuple lattice sieving. *LMS Journal of Computation and Mathematics*, 19:146—162, 2016.

CCL17. Yanlin Chen, Kai-Min Chung, and Ching-Yi Lai. Space-efficient classical and quantum algorithms for the shortest vector problem. *arXiv e-prints*, Aug 2017.

CDW17. Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-svp. In *Advances in Cryptology - EUROCRYPT 2017*, pages 324–348, 2017.

DRS14. D. Dadush, O. Regev, and N. Stephens-Davidowitz. On the closest vector problem with a distance guarantee. In *2014 IEEE 29th Conference on Computational Complexity (CCC)*, pages 98–109, June 2014.

Duc18. Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In *Advances in Cryptology – EUROCRYPT 2018*, pages 125–145, 2018.

Gal14. F. L. Gall. Improved quantum algorithm for triangle finding via combinatorial arguments. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 216–225, Oct 2014.

GLM08. Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Phys. Rev. Lett.*, 100:160501, Apr 2008.

GNR10. Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *Advances in Cryptology – EUROCRYPT 2010*, pages 257–278, 2010.

Gro96. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, 1996.

HK17. Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in Euclidean norm. In *Public-Key Cryptography – PKC 2017*, pages 16–40, 2017.

HKL18. Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time–memory trade-offs for tuple lattice sieving. In *Public-Key Cryptography – PKC 2018*, pages 407–436, 2018.

Kan83. Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 193–206, 1983.

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

KLM07. Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An introduction to quantum computing*. Oxford University Press, 2007.

Kup13. Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *8th Conference on the Theory of Quantum Computation, Communication and Cryptography, TQC*, pages 20–34, 2013.

Laa15. Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2015.

LGN17. François Le Gall and Shogo Nakajima. Quantum algorithm for triangle finding in sparse graphs. *Algorithmica*, 79(3):941–959, Nov 2017.

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, Dec 2015.

Map. Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. Standard Worksheet Interface, Maple 2016.0, February 17 2016.

Mon18. Ashley Montanaro. Quantum-walk speedup of backtracking algorithms. *Theory of Computing*, 14(15):1–24, 2018.

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 '10, pages 1468–1480, 2010.

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

PMHS19. Alice Pellet-Mary, Guillaume Hanrot, and Damien Stehlé. Approx-svp in ideal lattices with pre-processing. In *Advances in Cryptology – EUROCRYPT 2019*, pages 685–716, 2019.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, 2005.

Reg09. Oded Regev. Lecture notes: Lattices in computer science. `http://www.cims.nyu.edu/~regev/teaching/lattices_fall_2009/index.html`, 2009. Accessed: 30-04-2019.

TKH18. Tadanori Teruya, Kenji Kashiwabara, and Goichiro Hanaoka. Fast lattice basis reduction suitable forÂ massive parallelization and its application to the shortest vector problem. In *Public-Key Cryptography – PKC 2018*, pages 437–460, 2018.

A Configuration search algorithm

The pseudocode of the algorithm for the configuration problem from [HK17] is given in Algorithm A.1.

Algorithm A.1 Algorithm for the Configuration Problem

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

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

```
 1: $L_{\text{out}} \leftarrow \{\}$
 2: for all $\mathbf{x}_1 \in L_1$ do
 3:     for all $j = 2 \dots k$ do
 4:         $L_j^{(1)} \leftarrow$ FILTER($(\mathbf{x}_1, L_j, C_{1,j}, \varepsilon)$)
 5:     for all $\mathbf{x}_2 \in L_2^{(1)}$ do
 6:         for all $j = 3 \dots k$ do
 7:             $L_j^{(2)} \leftarrow$ FILTER($(\mathbf{x}_2, L_j^{(1)}, C_{2,j}, \varepsilon)$)
 8:         $\ddots$
 9:             for all $\mathbf{x}_k \in L_k^{(k-1)}$ do
10:                 $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\mathbf{x}_1, \dots \mathbf{x}_k)\}$
    return $L_{\text{out}}$

 1: function FILTER($()\mathbf{x}, L, c, \varepsilon$)
 2:     $L' \leftarrow \{\}$
 3:     for all $\mathbf{x}' \in L$ do
 4:         if $|\langle \mathbf{x} \, , \mathbf{x}' \rangle - c| \leq \varepsilon$ then
 5:             $L' \leftarrow L' \cup \{\mathbf{x}'\}$
        return $L'$
```

B Quantum algorithms for Locality Sensitive Filters

We employed quantum enumeration (Grover's algorithm) to quantumly speed up the creation of filtered lists. Recall how these filtered lists are created classically (see Section 3): given a point $\mathbf{x}$, a list L and a target inner product $C_{1,i}$, we iterate over all $\mathbf{x}_i \in L$ to find those that satisfy $\langle \mathbf{x} , \mathbf{x}_i \rangle \approx C_{1,i}$. This is precisely the problem addressed by neariest neighbour techniques. A nearest neighbour algorithm of particular interest to us is the the Locality Sensitive Filtering (LSF) method [BLS16]. In this subsection we briefly explain the method, and then we show how it can be sped up with a quantum computer. For $k = 2$ this idea was proposed by Laarhoven in [Laa15].

Spherical Locality Sensitive Filters The idea of LSF is to create a data structure $\mathcal{D}$ of *buckets* $B_{\mathbf{v}}$ of the form $\mathcal{D} = \cup_{\mathbf{v}} B_{\mathbf{v}}$. Each bucket $B_{\mathbf{v}}$, indexed by some *filter vector* $\mathbf{v} \in \mathsf{S}^{d-1}$, contains all vectors from L that are α-close to $\mathbf{v}$, i.e, $\langle \mathbf{x}, \mathbf{v} \rangle \geq \alpha$. Let us denote by $\mathcal{V}$ the set of all filter vectors $\mathbf{v}$. In application, $\mathcal{V}$ is large set: later we set $|\mathcal{V}|$ to be exponential in d.

For a given set of buckets and a list L, building a data structure $\mathcal{D}$ for L means that for each $\mathbf{x} \in L$, we find all $\mathbf{v} \in \mathcal{V}$ that are α-close to $\mathbf{x}$, and, for all such $\mathbf{v}$, we put $\mathbf{x}$ into the bucket $B_{\mathbf{v}}$. Now if we want to find all points from $\mathbf{x} \in L$ that are c-close to a query point $\mathbf{q}$, we first find all β-close $\mathbf{v} \in \mathcal{V}$ to $\mathbf{q}$, and then for all these $\mathbf{v}$ search inside the relevant buckets $B_{\mathbf{v}}$ for $\mathbf{x}$'s, which are c-close to $\mathbf{q}$. The main observation is that quantum search over the relevant buckets can be done using Grover's algorithm. First we create a superposition over the content of the relevant buckets (the bucket labels are known) and then apply Grover's search over the vectors stored in these buckets to find a c-close to $\mathbf{q}$. As we want to find all c-close vectors, we repeat this process until all these vectors are found.

To solve the task of finding relevant filter vectors $\mathbf{v}$ for a given $\mathbf{x}$ (or $\mathbf{q}$) efficiently, vectors $\mathbf{v}$ are chosen with some structure. On the one hand, it enables us to find filter vectors relevant for a given $\mathbf{x}$ in time (up to lower order terms) proportional to the number of such filter vectors. On the other hand, even with such structure, we can argue that the joint distribution of $\mathbf{v}$'s is sufficiently close to $|\mathcal{V}|$ independent vectors from S^{d-1}. For further details on this technique, we refer the reader to [BDGL16]. Here we simply assume that $\mathbf{v}$'s are independent and that we can find all relevant $\mathbf{v}$'s for a given query point in time essentially equal to the size of the output. Note that because we can find all the relevant filters in time asymptotically equal to the output size, applying Grover's search will not speed up this part of the algorithm. The main routines INSERT($\mathbf{x}$) – putting $\mathbf{x}$ into all α-close buckets, and QUERY($\mathbf{q}$) – finding all c-close points for $\mathbf{q}$, are listed in Algorithm B.1. The only difference between the classical and quantum routines are in the algorithm QUERY($\mathbf{q}$), where the search inside a relevant bucket is done using quantum enumeration.

The three parameters α, β, c govern the complexity of LSF. First, they determine the size of $\mathcal{V}$ necessary to find all vectors from a list, which are c-close to a given query point. The insertion parameter α determines the number of buckets a vector is put into and thus, controls the size of buckets. The query parameter β determines the number of relevant buckets for a query $\mathbf{q}$. In the next theorem, adapted from [HKL18, Theorem 3], we give closed formulas for these costs.

Algorithm B.1 Algorithms for spherical locality-sensitive filtering

```
   Parameters: $\alpha, \beta, c, \mathcal{V}$
   $\mathcal{D} = \cup_{\mathbf{v} \in \mathcal{V}} B_{\mathbf{v}}$.
1: function Insert($\mathbf{x}$)
2:     Find all $\mathbf{v} \in \mathcal{V}$ s.t. $\langle \mathbf{v}, \mathbf{x} \rangle \approx \alpha$
3:     for all $\mathbf{v} \in \mathcal{V}$ s.t. $\langle \mathbf{v}, \mathbf{x} \rangle \approx \alpha$ do
4:         $B_{\mathbf{v}} \leftarrow B_{\mathbf{v}} \cup \{\mathbf{x}\}$

1: function Query($\mathbf{q}$)                    ▷ Find $\mathbf{x} \in L$ with $\langle \mathbf{x}, \mathbf{q} \rangle \geq c$
2:     PointsFound $\leftarrow \emptyset$
3:     Find all $\mathbf{v} \in \mathcal{V}$ s.t. $\langle \mathbf{v}, \mathbf{x} \rangle \approx \beta$
4:     Create a superposition over all relevant buckets $\mathbf{v}$, query all vectors in these
   buckets
       $$|\Psi\rangle = \sqrt{\frac{1}{|\mathcal{V}|\,(1-\beta^2)^{d/2} \cdot |L|\,(1-\alpha^2)^{d/2}}} \sum_{\mathbf{v} \text{ relevant}} |\mathbf{v}\rangle \sum_{\mathbf{x} \in B_{\mathbf{v}}} |\mathbf{x}\rangle$$
5:     Run Grover's algorithm on $|\Psi\rangle$ to output an $\mathbf{x}$ using the checking function
       $$f(\mathbf{x}, \mathbf{x}') = \begin{cases} 1, & \langle \mathbf{x}, \mathbf{q} \rangle \approx c \\ 0, & \text{else.} \end{cases}$$
6:     PointsFound $\leftarrow$ PointsFound $\cup \{\mathbf{x}\}$
7:     Repeat Steps 4–6 $\widetilde{\mathcal{O}}(|L|\,(1-c^2)^{d/2})$ times until all the $c$-close vectors are found
8:     return PointsFound
```

Theorem 9 (Complexity of spherical LSF, adapted from [HKL18, Theorem 3]). *Assume we are given a list L of iid. uniform points from S^{d-1} of size exponential in d, a target $0 \leq c \leq 1$, and fixed $0 \leq \alpha, \beta \leq 1$. Then the costs of LSF procedures given in Algorithm B.1 determined by:*

– ***Number of buckets (filter vectors)*** *used in creating the data structure $\mathcal{D} = \cup_{\mathbf{v} \in \mathcal{V}} B_{\mathbf{v}}$:*

$$|\mathcal{V}| = \frac{\left(\det\left(\begin{smallmatrix} 1 & c \\ c & 1 \end{smallmatrix}\right)\right)^{d/2}}{\left(\det\left(\begin{smallmatrix} 1 & \alpha & \beta \\ \alpha & 1 & c \\ \beta & c & 1 \end{smallmatrix}\right)\right)^{d/2}};$$

– ***Update cost*** *per $\mathbf{x} \in L$ to find all α-close $\mathbf{v} \in \mathcal{V}$: $T_{\text{Update}} = |\mathcal{V}| \cdot (1-\alpha^2)^{d/2}$;*

– ***Preprocessing time*** *to build the data structure $\mathcal{D} = \cup_{\mathbf{v} \in \mathcal{V}} B_{\mathbf{v}}$:*

$$T_{\text{Prep}} = |L| \cdot |\mathcal{V}| \cdot (1-\alpha^2)^{d/2};$$

– ***Memory*** *to store the the data structure* $\mathcal{D}$ *is* $|L| \cdot |\mathcal{V}| \cdot (1-\alpha^2)^{d/2}$*;*
– ***Quantum query cost*** *per* $\mathbf{q}$ *to find all relevant* β*-close* $\mathbf{v} \in \mathcal{V}$ *and to find all* c*-close* $\mathbf{x}$*'s:*

$$T_{\text{Query}} = \; |\mathcal{V}| \cdot (1-\beta^2)^{d/2} + \sqrt{|\mathcal{V}|\,(1-\beta^2)^{d/2} \cdot |L|\,(1-\alpha^2)^{d/2} \cdot |L|\,(1-c^2)^{d/2}}.$$

A proof of this theorem can be found in [HKL18, Appendix D]. The intuition behind all these complexities is the following: for a point $\mathbf{x} \in L \subset \mathsf{S}^{d-1}$ and a fixed vector $\mathbf{v}$, the probability that $\langle \mathbf{v}\,,\mathbf{x}\rangle \approx \alpha$ is $(1-\alpha^2)^{d/2}$. Different to the classical case, we do not assume that the buckets are of expected size 1 (intuitively, this is because iterating over the buckets becomes cheaper in the quantum setting). The number of buckets is determined by the probability that the triple $(\mathbf{x},\mathbf{v},\mathbf{q})$, conditioned on the fact that $\langle \mathbf{x}\,,\mathbf{q}\rangle \approx c$, satisfies all the inner product constraints: $\langle \mathbf{x}\,,\mathbf{q}\rangle \approx c$, $\langle \mathbf{x}\,,\mathbf{v}\rangle \approx \alpha$, $\langle \mathbf{v}\,,\mathbf{q}\rangle \approx \beta$. The quantum query cost consists of first, finding all the relevant buckets in $\mathcal{V}$ and second, enumerating the expected number of $|L|\,(1-c^2)^{d/2}$ many c-close vectors. The expected size of each bucket is $|L|\,(1-\alpha^2)^{d/2}$.

Using LSF for the configuration problem. We make use of locality sensitive filters for the configuration problem as follows: given a target configuration C and the lists L_i, preprocess the lists L_i for $i \geq 2$ using the data structures D_i and the LSF parameters α_i, β_i. Once the preprocessing is done, we can construct the intermediate lists $L_i(\mathbf{x}_1)$ using quantum queries into D_i's for $\mathbf{x}_1 \in L_1$. When $L_i(\mathbf{x}_1)$ are constructed we can either run Grover's algorithm to find the $(k-1)$-tuple analogously to Algorithm 4.1, or we can construct new LSF data structures now for $L_i(\mathbf{x}_1)$'s and "delay" Grover's search for deeper levels. In general, to create all the filtered lists on level $j \geq 2$, we first prepare the LSF data structures $D_i^{(j-1)}$ for the lists $L_i^{(j-1)}$ on level $j-1$, for all $i \geq j$. This is done classically. Then for each $\mathbf{x}_{j-1} \in L_{j-1}^{(j-1)}$, we create the filtered lists $L_i^{(j)}$ by calling $\textsc{Query}(\mathbf{x}_{j-1})$ into the corresponding $D_i^{(j-1)}$. Contrary to Algorithm 4.2, here we run quantum enumeration over the relevant buckets rather than over $L_i^{(j-1)}$, which brings a speed-up provided the construction of $D_i^{(j-1)}$'s was not too costly.

Pseudocode of the algorithm for the configuration problem with LSF is given in Algorithm B.2. Let us again denote $L_i^{(j)} := L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})$ for all $i \geq j$, assuming $L_i^{(1)}$ are the input lists. Then the time needed to construct an intermediate list $L_i^{(j)}, i \geq j$ is

$$T(i,j) = \prod_{r=1}^{j-2} \left|L_r^{(r)}\right| \left(T_{\text{Prep}}(i,j-1) + \left|L_{j-1}^{(j-1)}\right| T_{\text{Query}}(j-1,j-1)\right),$$

where $T_{\text{Prep}}(i,j-1)$ is the preprocessing cost for $L_i^{(j-1)}$ and $T_{\text{Query}}(j-1,j-1)$ is the query costs for the elements from $L_{j-1}^{(j-1)}$. Therefore, to create all the intermediate lists on level j, we need time

$$T(j) = \max_{j\leq i\leq k} T(i,j).$$

These costs can be established from Theorem 9. Once all the lists on level j are constructed, we run Grover search for a good $(k-j+1)$ tuple analogous to Algorithm 4.2. Overall complexity of Algorithm B.2 (cf. Eq. (11)) is:

$$T^{\mathrm{Q}}_{\mathrm{LSF}}(j) = \max \left\{ T(j), \prod_{r=1}^{j-1} \left| L_r^{(r)} \right| \left(\sqrt{\frac{\left|L_{j+1}^{(j)}\right|}{\left|L_{j+1}^{(j+1)}\right|}} + \ldots + \sqrt{\frac{\left|L_{k-1}^{(j)}\right|}{\left|L_{k-1}^{(k-1)}\right|}} + \sqrt{\frac{\left|L_{k}^{(j)}\right|}{\left|L_{k}^{(k-1)}\right|}} \right) \right.$$
$$\left. \cdot \sqrt{\left|L_j^{(j)}\right| \cdot \ldots \left|L_{k-1}^{(k-1)}\right| \cdot \left|L_k^{(k-1)}\right|} \right\}. \tag{13}$$

The best runtime exponents are obtained when the locality sensitive techniques are used to construct the intermediate lists, Algorithm B.2. It is again optimal to set $j = 2$ (as in the case of Algorithm 4.2), i.e., when the LSF data structure is used to quantumly construct the second level lists. For $k = 2$ the exponent was established in [Laa15] and is currently the best known when we seek for optimal time. Locality sensitive filters also speed up the configuration search algorithm for $k = 3, 4$. However, the advantage of near neighbour techniques decreases when the lists become smaller, which is what we observe when k increases. Furthermore, the LSF technique (and near neighbour methods, in general) become more expensive when the target inner product c becomes smaller: one requires a larger data structure to allocate c-close vectors, which is why we did not run numerical optimisations for large k's.

k	2	3	4	5
	Quantum k-List with LSF			
Time	0.2653	0.2908	0.3013	0.3048
Space	0.2653	0.2908	0.3013	0.3048
	Quantum k-List with LSF, low-memory regime			
Time	0.2925	0.3266	0.3178	0.3281
Space	0.2075	0.1887	0.1724	0.1587

Table 3: Asymptotic complexity exponents for the approximate k-List problem using the LSF techniques, Algorithm B.2 with $j = 2$.

.

Algorithm B.2 Quantum algorithm for the Configuration problem with LSF

Input: $L_1, \ldots, L_k-$ lists of vectors from S^{d-1}, target configuration $C_{i,j} = \langle \mathbf{x}_i , \mathbf{x}_j \rangle \in \mathbb{R}^{k \times k}-$ a Gram matrix, $\varepsilon > 0$, $2 \le j \le k-1-$ level until we construct the intermediate filtered lists; LSF parameters $\{\alpha_{i,r}, \beta_{i,r}\}_{1 \le i \le k, 1 \le r \le j-1}$
Output: $L_{\text{out}}-$ list of k-tuples $(\mathbf{x}_1, \ldots, \mathbf{x}_k) \in L_1 \times \cdots \times L_k$, s.t. $|\langle \mathbf{x}_i , \mathbf{x}_j \rangle - C_{ij}| \le \varepsilon$ for all i, j.

```
 1: $L_{\text{out}} \leftarrow \emptyset$
 2: Create LSF data structures $\mathcal{D}_i^{(1)}$ with parameters $\{\alpha_{i,1}, \beta_{i,1}\}$ $\forall L_i, i \ge 2$
 3: for all $\mathbf{x}_1 \in L_1$ do
 4:     Call Query($\mathbf{x}_1$) using $\mathcal{D}_i^{(1)}$ to construct $L_i(\mathbf{x}_1)$ $\forall i \ge 2$
 5:     Create LSF data structures $\mathcal{D}_i^{(2)}$ with parameters $\{\alpha_{i,2}, \beta_{i,2}\}$ $\forall L_i, i \ge 3$
 6:     for all $\mathbf{x}_2 \in L_2(\mathbf{x}_1)$ do
 7:         Call Query($\mathbf{x}_1$) using $\mathcal{D}_i^{(2)}$ to construct $L_i(\mathbf{x}_1, \mathbf{x}_2)$, $\forall i \ge 3$
 8:             $\ddots$
 9:             for all $\mathbf{x}_{j-1} \in L_{j-1}(\mathbf{x}_1, \ldots, \mathbf{x}_{j-2})$ do
10:                 Call Query($\mathbf{x}_1$) using $\mathcal{D}_i^{(j-1)}$ to construct $L_i(\mathbf{x}_1, \ldots, \mathbf{x}_{j-1})$, $\forall i \ge j$
11:             Prepare the state $|\Psi_{L_j(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle \otimes \ldots \otimes |\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$
12:             for all $i = j+1 \ldots k-1$ do
13:                 Run Grover's on the $i^{\text{th}}$ register with the checking function $f_{[i-1],i}$
                    to transform the state $|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$ to the state $|\Psi_{L_i(\mathbf{x}_1,\ldots,\mathbf{x}_{i-1})}\rangle$.
14:             Run Grover's on the $k^{\text{th}}$ register with the checking function $f_{[k-2],k}$ to
                transform the state $|\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle$ to the state $|\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-2})}\rangle$.
15:             Let $\mathcal{A}$ be unitary that implements Steps 9–12, i.e.
                    $\mathcal{A}\,|\mathbf{0}\rangle \to |\Psi_{L_j(\mathbf{x}_1,\ldots,\mathbf{x}_{j-1})}\rangle \otimes |\Psi_{L_k(\mathbf{x}_1,\ldots,\mathbf{x}_{k-1})}\rangle$
16:             Run amplitude amplification using the unitary $-\mathcal{A}R\mathcal{A}^{-1}O_g$, where $g$ is
                defined in Eq. (8).
17:             Measure all the registers, obtain a tuple $(\mathbf{x}_j, \ldots, \mathbf{x}_k)$.
18:             if $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ satisfies $C$ then
19:                 $L_{\text{out}} \leftarrow L_{\text{out}} \cup \{(\mathbf{x}_1, \ldots, \mathbf{x}_k)\}$.
```

C Some More k-clique Cases

C.1 The $k = 4$ Case

Here we modify the idea of applying quantum triangle listing to the 3-List problem developed in Section 5, to the $k = 4$ case. We use the notations from Section 5. The general case is covered in Section 5.2. For $k = 4$, we obtain a graph with $n = |L| = \mathcal{O}\left(\left(4\sqrt[3]{4}/5\right)^{d/2}\right)$ edges, $m = |L||L(\mathbf{x}_1)|$ vertices, $t = |L||L(\mathbf{x}_1)||L(\mathbf{x}_1, \mathbf{x}_2)|$ triangles and $\Theta(n)$ 4-cliques (cf. Eq. (12)). The algorithm from Section 5.1 can be modified to give an algorithm for finding 4-cliques as follows.

1. Use Grover's algorithm to find an edge $(\mathbf{x}_1, \mathbf{x}_2) \in E$ among all potential $\mathcal{O}(n^2)$ edges.
2. Given an edge $(\mathbf{x}_1, \mathbf{x}_2)$, use Grover's algorithm to find a vertex $\mathbf{x}_3 \in V$, such that $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ is a triangle.
3. Given a triangle $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$, use Grover's algorithm to find a vertex $\mathbf{x}_4 \in V$, such that $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ is a 4-clique.
4. Apply amplitude amplification on steps 1–3.

As in the $k = 3$ case, the algorithm returns any 4-clique in the graph, rather than a fixed one. Step 1 searches for any edge, and given that edge Step 2 searches for any triangle containing it.

Step 1 costs $\mathcal{O}(\sqrt{n^2/m})$ and Step 3 costs $\mathcal{O}(\sqrt{n})$. As an edge is expected to be in many triangles, $t/m = \alpha^{d/2}, \alpha > 1$, the cost of Step 2 is below $\mathcal{O}(\sqrt{n})$. As steps 1–3 are additive, step 3 dominates and their cost is $\mathcal{O}(\sqrt{n})$, as before.

For steps 1–3 to find a 4-clique, two things must happen. First, Step 1 needs to pick a good edge, that is, an edge in a 4-clique. Given there are $\Theta(n)$ 4-cliques, this happens with probability $\Theta(n/m)$. Next, given that Step 1 picks a good edge, Step 2 must pick a triangle, including the good edge, which itself is in a 4-clique. This happens with probability $\Theta(m/t)$. The success probability is therefore $\Theta(n/t)$. Using amplitude amplification the total cost of finding a 4-clique is therefore $\mathcal{O}(\sqrt{t})$.

By the coupon collector's problem, the total cost of finding all 4-cliques is $\widetilde{\mathcal{O}}(n\sqrt{t}) = \widetilde{\mathcal{O}}\left(|L|^{3/2}(|L(\mathbf{x}_1)||L(\mathbf{x}_1, \mathbf{x}_2)|)^{1/2}\right) \approx 2^{0.3418d+o(d)}$ using $2^{0.1724d+o(d)}$ memory. This also matches the complexity of Algorithm 4.1 in the $k = 4$ balanced setting.

C.2 The General k-clique Case for Unbalanced Configurations

The graph approach of Section 5 can also be applied to unbalanced configurations. Given the matrix C of a good configuration we form the following coloured, undirected graph, $G = (V, E)$. We have lists $L = L_1 = \cdots = L_k$ and vertices are elements of L. Let there be an edge of colour $c_{i,j}$ between vertices $\mathbf{x}_1$ and $\mathbf{x}_2$ if and only if $|\langle \mathbf{x}_1 , \mathbf{x}_2 \rangle - C_{i,j}| \leq \varepsilon$, with ε as before. Define oracles $O_{G_{i,j}} \colon V^2 \to \{\texttt{True}, \texttt{False}\}$ such that $O_{G_{i,j}}(\mathbf{x}_1, \mathbf{x}_2) = \texttt{True}$ if and only if there is an edge of colour $c_{i,j}$ between $\mathbf{x}_1, \mathbf{x}_2$. That is, we have $\binom{k}{2}$ coloured edges and for each colour we have an oracle that checks whether there is an edge of this colour between two vertices.

In this setting we search for a coloured k-clique. That is, a set of vertices $\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ such that there is an edge of colour $c_{i,j}$ between the vertices $\mathbf{x}_i$ and $\mathbf{x}_j$, for all $i \neq j$.

With small changes, we can apply the same k-clique finding algorithm in this setting. The algorithm is the following, with $2 \leq i \leq k-1$

1. Use Grover's algorithm to find an edge $(\mathbf{x}_1, \mathbf{x}_2)$ of colour $c_{1,2}$ among all potential $\mathcal{O}(|L|^2)$ edges of colour $c_{1,2}$.

$$\vdots$$

i. Given a coloured i-clique $(\mathbf{x}_1, \ldots, \mathbf{x}_i)$, use Grover's algorithm to find a vertex $\mathbf{x}_{i+1} \in V$, such that $(\mathbf{x}_1, \mathbf{x}_{i+1})$, …, $(\mathbf{x}_i, \mathbf{x}_{i+1})$ are edges of colours $c_{1,i+1}$, …, $c_{i,i+1}$ respectively, to form a coloured $(i+1)$-clique $(\mathbf{x}_1, \ldots, \mathbf{x}_{i+1})$.

$$\vdots$$

k. Apply amplitude amplification on steps $1-(k{-}1)$.

The complexity analysis is similar to the one in Section 5.2. The dominant cost of Steps $1-(k{-}1)$ is a Grover search over the $|L_i| = |L|$ vertices. What differs slightly is the calculation of the success probability.

Let us first look at the triangle case. An unbalanced configuration implies that the number of edges of colours $c_{1,2}$, $c_{1,3}$ and $c_{2,3}$ may be different. However, by having picked a good configuration and the right initial list size $|L|$ we still have $n = \Theta(|L|)$ triangles with one edge of colour $c_{1,2}$, one edge of colour $c_{1,3}$ and one edge of colour $c_{2,3}$. The total number of edges of colour $c_{1,2}$ is $m = \mathcal{O}(|L||L_2(\mathbf{x}_1)|)$. The probability that Steps 1–2 succeed is equal to the probability that Step 1 picks an edge of colour $c_{1,2}$, belonging to a coloured triangle. This probability is equal to $\Theta(n/m) = \Theta(1/|L_2(\mathbf{x}_1)|)$.

In the $k = 4$ case, by having picked a good configuration and the right initial list size $|L|$, even though the number of edges of different colours varies, the expected number of coloured 4-cliques is still $n = \Theta(|L|)$. The probability in Step 1 of picking a coloured edge belonging to a 4-clique is $\Theta(n/m) = \Theta(1/|L_2(\mathbf{x}_1)|)$. Given such an edge $(\mathbf{x}_1, \mathbf{x}_2)$, comparing with Figure 2b, we see that the number of coloured triangles $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$, such that $(\mathbf{x}_1, \mathbf{x}_2)$ is an edge of colour $c_{1,2}$, is equal to $\mathcal{O}(|L_3(\mathbf{x}_1, \mathbf{x}_2)|)$. Given that we have picked an edge belonging to a 4-clique, the number of these triangles belonging to a 4-clique is $\Theta(1)$. Thus, the total success probability for the $k = 4$ case is equal to $\Theta(1/(|L_2(\mathbf{x}_1)||L_3(\mathbf{x}_1, \mathbf{x}_2)|))$.

In the general case, in the first step the probability of finding an edge of colour $c_{1,2}$ belonging to a coloured k-clique is $\Theta(1/|L_2(\mathbf{x}_1)|)$. Next, assume that step $i-1$, for $2 \leq i \leq k-2$, has found a coloured i-clique $(\mathbf{x}_1, \ldots, \mathbf{x}_i)$ belonging to a coloured k-clique. The number of coloured $(i+1)$-cliques on the form $(\mathbf{x}_1, \ldots, \mathbf{x}_i, \mathbf{x}_{i+1})$ is $\Theta(\max(|L_{i+1}(\mathbf{x}_1, \ldots, \mathbf{x}_i)|, 1))$. This corresponds to the diagonal elements of Figure 2b. Of these $(i+1)$-cliques, $\Theta(1)$ belong to a coloured k-clique. Thus the probability of picking a coloured $(i+1)$-clique belonging to a coloured k-clique is $\Theta(1/\max(|L_{i+1}(\mathbf{x}_1,\dots,\mathbf{x}_i)|,1))$. The total success probability is thus

$$\Theta\left(\left(\prod_{i=1}^{k-2}\max\left(|L_{i+1}(\mathbf{x}_1,\dots,\mathbf{x}_i)|,1\right)\right)^{-1}\right).$$

By applying the amplitude amplification on Step k, we get the cost

$$\mathcal{O}\left(\sqrt{|L|}\sqrt{\prod_{i=1}^{k-2}\max\left(|L_{i+1}(\mathbf{x}_1,\dots,\mathbf{x}_i)|,1\right)}\right),$$

for finding one k-clique. The total complexity of this algorithm is thus equal to

$$\widetilde{\mathcal{O}}\left(|L|^{3/2}\sqrt{\prod_{i=1}^{k-2}\max\left(|L_{i+1}(\mathbf{x}_1,\dots,\mathbf{x}_i)|,1\right)}\right).$$

This matches the complexity of Algorithm 4.1 in the unbalanced setting, i.e. Eq. (9).

D Proofs of Lemmas from Section 7

Here we present the proofs of Lemmas 1, 2, 3 essentially giving a blueprint of the construction of the circuit implementing Algorithm 7.1. We will often describe unitaries (and reduced transformations) in detail, but equally often simply describe the operation and just claim there is some unitary that implements it. Particularly, when it is clear or can be derived by observation, such as when it can be shown there is an efficient classical reversible circuit implementing a function, using only Toffoli and NOT gates, one can as well construct an efficient quantum circuit to implement that. This allows us to focus on conveying the non-trivial and interesting aspects of the construction, rather than tiring an interested reader with trivialities such as presenting a unitary comprised of elementary gate sets that carries out matrix multiplication efficiently in the size of the input bitstrings. Lastly, we assume the setup as described earlier and crucially the fact that the circuit is prepared and initialised before the input, which consists of basis vectors, is received.

Proof of Lemma 1. We denote the input as a state $|B\rangle$. We begin by copying $|B\rangle$ into N processors. To do this, first define unitary $RC_{A,B}$ that implements a CNOT to registers A and B, with A being the control and B the target. Assume both registers are of same size.

$$RC_{A,B} := CNOT_{A1,B1} \otimes CNOT_{A2,B2} \otimes CNOT_{A\bar{d},B\bar{d}}, \tag{14}$$

where Ai and Bi are the $i-$th qubit of register A and B respectively. Notice thst all control and target qubits are different and can be composed in parallel. Thus the operator $RC_{A,B}$ can be applied applied with a depth-1 circuit. Operationally, a register of arbitrary size can be correlated with another register of (at least) the same size in 1 time step.

To copy $|B\rangle$ to all N processors a cascade of RC operations suffice. To be more precise, we first copy $|B\rangle$ to register $\mathtt{R}_1$ in processor $\mathtt{P}_1$. In the next time step copy $|B\rangle$ and $\mathtt{R}_1$ to registers $\mathtt{R}_2$ and $\mathtt{R}_3$ in processors $\mathtt{P}_2$, $\mathtt{P}_3$respectively. Continue the process for $\log(N)$ steps, where $|B\rangle, \mathtt{R}_1 \ldots \mathtt{R}_{N/2-1}$ are copied to $\mathtt{R}_{N/2}, \mathtt{R}_{N/2+1} \ldots \mathtt{R}_{N-1}$ registers of the last $N/2$ processors respectively (wlog, we assume N is even). Effectively, a $\log(N)$ depth circuit copies $|B\rangle$ to all N processors. The width of the circuit is $\widetilde{\mathcal{O}}(N)$, since each processor only needs a $\mathrm{poly}(d) = \mathrm{poly}\log N$ to store all the basis vectors.

Each processor $\mathtt{P}_i$ samples a vector $\mathbf{x}_i$ from the lattice $\mathcal{L}(B)$ using an efficient randomised sampling procedure. There are various ways to do it, we employ the standard technique due to Klein [Kle00]. It consists of sampling a vector $\mathbf{t}_i$ from $\mathbb{R}^d$ and running Babai's decoding algorithm to find a lattice point close to $\mathbf{t}_i$. Babai's algorithm can be implemented using a reversible circuit of depth $\mathrm{poly}(d)$. To sample $\mathbf{t}_i \in \mathbb{R}^d$ we first put some (hard wired) large enough bound C and produce a vector $\mathbf{t}_i \in [-C, C]^d$ coordinate-wise using the following process. We construct a $C+1$ dimensional maximally entangled state, $\frac{1}{\sqrt{C+1}} \sum_{j=0}^{C+1} |jj\rangle$ and discard the second of the two subsystems. Thus $\mathbf{t}_i = \mathrm{Tr}_2(\frac{1}{C+1} \sum_{i,j} |ii\rangle \langle jj|_{12})$.

Operationally this would be equivalent to picking a random value less than $C + 1$. For purposes of the circuit the first value picks the sign, and rest of it a number between 0 and C. This corresponds to picking one element of the vector $\mathbf{t}_i$. Repeating the same construction d times in parallel gives the complete vector $\mathbf{t}_i$. It is efficient to construct a C-dimensional maximally entangled state as $\mathcal{O}(C)$ depth circuit suffices. Note that the requirement on the maximally entangled state and C are independent of the input vectors themselves, hence this can be considered as a resource from the initialisation phase. Thus a circuit that samples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ needs $\mathcal{O}(\text{poly}\log(N))$ depth and $\widetilde{\mathcal{O}}(N)$ width.

Finally to write each of these $\mathbf{x}_i$ into a distinguished memory cell $\texttt{L1}$, the unitary

$$RC_{\texttt{R}_1,\texttt{L1}[1]} \otimes RC_{\texttt{R}_2,\texttt{L1}[2]} \ldots \otimes RC_{\texttt{R}_N,\texttt{L1}[N]}$$

is applied and this is again of depth 1. Note that $\texttt{L1}[0]$ is the zero vector.

Thus the list L_1 can be sampled efficiently, given an input $|B\rangle$, of $\text{poly}(d)$ qubits, and the circuit that implements it, is of depth $\text{poly}\log(N)$ and width $\widetilde{\mathcal{O}}(N)$. This concludes proof of Lemma 1.

$\square$

Proof sketch of Lemma 2. With a new list L_1 that stores N vectors, we want to construct another list L_2, such that for any $i \in [N]$ it holds that $\mathbf{x}'_i \in L_2$ is such that $|\langle \mathbf{x}'_i \, , \mathbf{x}_i \rangle| \geq 1/2$ and $\mathbf{x}_i \in L_1$ (If no such $\mathbf{x}_i \in L_1$ exist for a given $\mathbf{x}_i$, then set $\mathbf{x}_i = 0$).

We begin by defining a checking function $f(\mathbf{x}, \mathbf{y}) = 1$ if $|\langle \mathbf{x} \, , \mathbf{y} \rangle| \geq 1/2$ and 0 otherwise. The corresponding unitary is of the form

$$U_f : |\mathbf{x}\rangle \, |\mathbf{y}\rangle \, |b\rangle \rightarrow |\mathbf{x}\rangle \, |\mathbf{y}\rangle \, |b \oplus f(\mathbf{x}, \mathbf{y})\rangle \, .$$

First notice this unitary can be implemented efficiently in the size of the inputs $\mathbf{x}, \mathbf{y}$ since it only computes the inner product and already has an classically efficient algorithm.

Now for each processor i, define $f_i(\mathbf{y}) = 1$ if $|\langle \mathbf{x}_i \, , \mathbf{y} \rangle| \geq 1/2$ and 0 otherwise with the corresponding unitary

$$U_{f_i} : |\mathbf{y}\rangle \, |b\rangle \rightarrow |\mathbf{y}\rangle \, |b \oplus f(\mathbf{x}_i, \mathbf{y})\rangle \, .$$

It is important to stress that $\mathbf{x}_i$ is fixed for the f_i, thus the domain of f_i is only the set of d-dimensional vectors, whereas for f it is the set of d^2 dimensional vectors and the unitaries U_{f_i} are really reduced unitaries.

Next, in order to load the data on which parallel Grover's search is performed, we define the unitary

$$V : |j_1 \ldots j_N\rangle \, |\mathbf{y}_1 \ldots \mathbf{y}_N\rangle \, |\mathbf{x}_1 \ldots \mathbf{x}_N\rangle \rightarrow |j_1 \ldots j_N\rangle \, |\mathbf{y}_1 \oplus \mathbf{x}_1 \ldots \mathbf{y}_N \oplus x_N\rangle \, |\mathbf{x}_1 \ldots \mathbf{x}_N\rangle \, .$$

One of the results of [BBG+13] states that there is a uniform family of circuits of width $\mathcal{O}(N(d \cdot \log N))$ and depth $\mathcal{O}(\log N \cdot \log(d \cdot \log N))$ that implements V. We use this unitary together with U_{f_i} to construct

$$\mathcal{W} = V \circ U_{f_1} \otimes \ldots \otimes U_{f_N} \circ V.$$

The purpose of $\mathcal{W}$ is to load the database in the memory using V, compute f_i on each of the processor in parallel using U_{f_i}, and then unload the database using $V^\dagger = V$. The action of $\mathcal{W}$ on the initial state is

$$\begin{aligned}
&\left|j_1 \ldots j_N\right\rangle \left|0 \ldots 0\right\rangle \left|\mathbf{y}_1 \ldots \mathbf{y}_N\right\rangle \left|\mathbf{x}_1 \ldots \mathbf{x}_N\right\rangle \\
\xrightarrow{V}\, &\left|j_1 \ldots j_N\right\rangle \left|\mathbf{x}_{j_1} \ldots \mathbf{x}_{j_N}\right\rangle \left|\mathbf{y}_1 \ldots \mathbf{y}_N\right\rangle \left|\mathbf{x}_1 \ldots \mathbf{x}_N\right\rangle \\
\xrightarrow{\otimes U_{f_i}}\, &\left|j_1 \ldots j_N\right\rangle \left|\mathbf{x}_{j_1} \ldots \mathbf{x}_{j_N}\right\rangle \left|\mathbf{y}_1 \oplus f_1(\mathbf{x}_{j_1}) \ldots \mathbf{y}_N \oplus f_N(\mathbf{x}_{j_N})\right\rangle \left|\mathbf{x}_1 \ldots \mathbf{x}_N\right\rangle \\
\xrightarrow{V}\, &\left|j_1 \ldots j_N\right\rangle \left|0 \ldots 0\right\rangle \left|\mathbf{y}_1 \oplus f_1(\mathbf{x}_{j_1}) \ldots \mathbf{y}_N \oplus f_N(\mathbf{x}_{j_N})\right\rangle \left|\mathbf{x}_1 \ldots \mathbf{x}_N\right\rangle
\end{aligned}$$

Setting $\left|\mathbf{y}_i\right\rangle = \left|-\right\rangle$, for all i, the reduced unitary becomes

$$\mathcal{W} : \left|j_1 \ldots j_N\right\rangle \to (-1)^{f_1(\mathbf{x}_{j_1})} \left|j_1\right\rangle \ldots (-1)^{f_N(\mathbf{x}_{j_N})} \left|j_N\right\rangle .$$

The unitary $\mathcal{W}$ can be implemented using a uniform family of circuits of width $N \cdot (\log N + d) = N \cdot \operatorname{poly}\log N$ and depth $\mathcal{O}(\log N \log(d \log N) + \operatorname{poly}(d)) = \operatorname{poly}\log(N)$.

Define $\left|\Psi\right\rangle = \frac{1}{\sqrt{N}} \sum_i \left|i\right\rangle$, and let $R = (2\left|\Psi\right\rangle\left\langle\Psi\right| - I)^{\otimes N}$. Notice that R can be implemented using a $\mathcal{O}(\log N)$ depth circuit. Executing $R \circ \mathcal{W}$ (Grover's iteration) $\Theta(\sqrt{N})$ times on the initial state

$$\sum_{j_1,\ldots j_N} \left|j_1 \ldots j_N\right\rangle \left|0 \ldots 0\right\rangle \left|- \ldots -\right\rangle \left|\mathbf{x}_1 \ldots \mathbf{x}_N\right\rangle$$

leads to the sate (close to)

$$\left|J\right\rangle_4 \left|\mathbf{0}\right\rangle_3 \left|-\right\rangle_2 \left|X\right\rangle_1 ,$$

where $J = (j'_1 \ldots j'_N)$, $X = \{\mathbf{x}_1 \ldots \mathbf{x}_N\}$, with the property that if the first block $\left|J\right\rangle$ measured, it would give a sample of indices, $J = (j'_1 \ldots j'_N)$ such that $\Theta(N)$ of them would be 'good indices'. Namely, for (almost) all j'_i, we have that $|\langle \mathbf{x}_{j'_i}, \mathbf{x}_i\rangle| \geq 1/2$ for the $\mathbf{x}_i$ from the first block $\left|X\right\rangle$. This is an instantiation of parallel quantum search over a single database and is the key ingredient of the algorithm. The circuit implementing this procedure is width and depth $\widetilde{\mathcal{O}}(N)$ and $\widetilde{\mathcal{O}}(\sqrt{N})$ respectively [BBG+13].

The success probability of each processor $\mathtt{P}_i$ to find j'_i is at least $1 - \Theta(1/N)$ (see Theorem 1 and the fact that we expect to have $\Theta(1)$ "good" j_i's for $\mathbf{x}_i$). We might as well repeat this procedure a constant number of times, and collect the set of indices $J^0, \ldots J^q$ and check if there exits a $j^l_k \in \{j^0_k, \ldots j^q_k\}$ such that $|\langle \mathbf{x}_{j^l_k}, \mathbf{x}_k\rangle| \geq 1/2$ (where j^l_k denotes the k-th register of J^l). Each processor can do this check in parallel, i.e. the k^{th} processor checks among the collected q-elements to find if there is a correct solution. If there is one, it copies, using the RC operator defined in Eq. (14), this index to a distinguished register, called $\mathtt{L2}[k]$ corresponding to the k^{th} processor $\mathtt{P}_k$. If no solution is found, $\mathtt{L2}[k]$ is set to the zero vector. This repetition contributes with a small poly(d) overhead in depth and width. Now we have virtual list L_3 stored at the memory block $\mathtt{L2}$, whose k^{th} register holds the index of a "good" vector.

To create the list L_3, the crucial idea is to imagine a classical reversible subroutine, for each processor k, that first computes two vectors $\mathbf{y}_k^+ = \mathbf{x}_k + \mathbf{x}_{\texttt{L2}[k]}$ and $\mathbf{y}_k^- = \mathbf{x}_k - \mathbf{x}_{\texttt{L2}[k]}$. Then if $||\mathbf{y}_k^+|| < ||\mathbf{y}_k^-||$, copy $\mathbf{y}_k^+$ to a designated register $\texttt{L3}[k]$ otherwise copy $\mathbf{y}_k^-$ to $\texttt{L3}[k]$. It can be verified that a circuit implementing this subroutine using only Toffoli and NOT gates has poly(d) depth and width.

Finally to swap the labels of `L1` and `L3` register blocks, apply a single swap gate $CNOT_{i,j} \circ CNOT_{j,i} \circ CNOT_{i.j}$ pairwise for each qubit pair, i.e., swap the first two qubits of the register block `L1` and `L3`, then in parallel the second two qubits of `L1`, `L3` and so on. The swap procedure is a depth 3 circuit.

Finally we would want to free the memory from `L2` and `L3` for later computations. The easiest way is to simply transfer the content (by using a SWAP gate like before) to auxiliary memory cells of the same size that were already initialised to state $|\mathbf{0}\rangle$.

Thus we have shown Steps (3)-(5) of Algorithm 7.1 can be implemented with a circuit of depth $\widetilde{\mathcal{O}}(\sqrt{N})$ and width $\widetilde{\mathcal{O}}(N)$. This concludes the proof of Lemma 2.

$\square$

Proof sketch of Lemma 3. We now want to find an element in the list L_1 with Euclidean norm less than a given threshold λ. We set $\lambda = \sqrt{d} \cdot \det(B)^{1/d}$ to satisfy Minowski's bound for $\mathcal{L}(B)$. This bound is tight up to a constant, which we do not know a priori. In order to avoid too many vectors from L_1 to satisfy the bound, we run several trials of Grover's algorithm starting with a small value for λ and re-iterating Grover's search with this value increased. We expect to terminate after $\mathcal{O}(1)$ iterations. The checking function for each Grover's search is $g(\mathbf{x}) = 1$ if $||\mathbf{x}|| < \lambda$, and 0 otherwise, and the corresponding unitary implementing the function $T : |\mathbf{x}\rangle |\mathbf{y}\rangle \rightarrow |\mathbf{x}\rangle |\mathbf{y} \oplus f(\mathbf{x})\rangle$. This admits efficient classical and quantum circuits of depth and width poly(d). As each Grover's iteration requires $\mathcal{O}(\sqrt{N})$ depth circuit, we show that Step 7 of Algorithm 7.1 needs $\widetilde{\mathcal{O}}(N)$ width (input) and $\widetilde{\mathcal{O}}(T\sqrt{N})$. It concludes the proof of Lemma 3.

$\square$