Quantum speedups for lattice sieves are tenuous at best

Martin R. Albrecht[1], Vlad Gheorghiu[2], Eamonn W. Postlethwaite[1], and John M. Schanck[2⋆]

[1] Information Security Group, Royal Holloway, University of London
[2] Institute for Quantum Computing, University of Waterloo, Canada

Abstract. Quantum variants of lattice sieve algorithms are often used to assess the security of lattice based cryptographic constructions. In this work we provide a heuristic, non-asymptotic, analysis of the cost of several algorithms for near neighbour search on high dimensional spheres. These algorithms are used in lattice sieves. We design quantum circuits for near neighbour algorithms and provide software that numerically optimises algorithm parameters according to various cost metrics. Using this software we estimate the cost of classical and quantum near neighbour search on spheres. We find that quantum search may provide a small speedup in dimensions of cryptanalytic interest, but only under exceedingly optimistic physical and algorithmic assumptions.

1 Introduction

In recent years lattice sieve algorithms have received a great deal of attention [1, 43, 8, 18, 2]. These algorithms take as input the basis of a lattice and solve the shortest vector problem (SVP) by combining exponentially many lattice vectors. The attention mostly stems from lattice based cryptography, as many attacks on lattice based cryptographic constructions involve finding short lattice vectors [42, 39, 3].

One reason lattice based cryptography appears well placed is the lack of known quantum algorithms which are able to solve SVP (to small approximation factors) in subexponential time. This is not to say that there is no gain to be made given a large scale quantum computer. In particular Grover's algorithm [29] is often applied to the relevant parts of lattice sieves when considering the quantum security of lattice based schemes. This algorithm square roots the asymptotic complexity of searching for an element in an unstructured list; a search that is

⋆ The research of MA was supported by EPSRC grants EP/S020330/1, EP/S02087X/1, by the European Union Horizon 2020 Research and Innovation Program Grant 780701 and Innovate UK grant AQuaSec; the research of EP was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1). VG and JS were supported by NSERC and CIFAR. IQC is supported in part by the Government of Canada and the Province of Ontario.

expected to take $\Theta(N)$ queries on classical hardware will take $\Theta(\sqrt{N})$ queries on quantum hardware using Grover's algorithm.

All lattice sieves, at some point in their execution, check a lattice vector against a database of other lattice vectors to determine whether any combinations produce shorter vectors. By defining pairs of vectors that produce shorter vectors as "close" this search can be formulated as a nearest neighbour search (NNS). It is here where Grover's algorithm is applied to give an asymptotic speed up.

However, a query in the context of Grover's algorithm, while formed of some fixed quantum circuit, will almost certainly have a much larger cost than a classical query. Therefore solely comparing the number of classical and quantum queries does not capture the full subtlety of the situation. In particular such a comparison may overestimate the cryptanalytic advantage an adversary with a large scale quantum computer is able to achieve.

Added to this is the requirement, based on the current understanding of how quantum computers will operate, of embedding any sufficiently long quantum computation into a quantum error correcting code. This process adds an overhead, a great deal of which is classical, to any quantum computation.

Finally, quantum lattice sieves, similar to their classical counterparts, require exponential memory. In the quantum case this takes the form of an exponentially large quantum random access memory (qRAM) which supports superposition queries to classical data. While proposals exist for designing circuits that realise this functionality, the performance characteristics of such proposals are radically different from classical RAM, rendering the proposition of abstracting away the cost of memory even more questionable than in the classical world. We note, however, that the literature routinely considers quantum lattice sieves in the qRAM model. For example, several lattice schemes submitted to the NIST PQC process use quantum lattice sieving estimates from [36, 35] to conservatively quantify their security.

Contributions. We start with some preliminaries in Section 2, where we discuss our models of computation, black box search, geometric quantities of relevance to sieving algorithms, sieving algorithms themselves, and the relationship between sieving and NNS. Finally we discuss the key gadget that we consider in this work, the "XOR and Population Count" operation (henceforth `popcount`), which was introduced to lattice sieves in [19] and used in [18, 2]. The `popcount` operation is used as a cheap filter to identify close pairs of vectors.

In Section 3 we present a design for a performant—under the assumption of unit cost qRAM—quantum circuit for `popcount`. Our design is carefully optimised for size. For comparison we also give classical costs for `popcount`. In Section 4 we provide a heuristic analysis of the probability of success and failure when using `popcount` to identify vectors that are close. This analysis may be of independent interest; previous work [18, 2] has relied largely on experimental data for choosing `popcount` parameters.

In Section 5, we derive the overall cost, classically and quantumly, of NNS as it appears as a subroutine in [43], the `bgj1` specialisation [2] of [9], and in [8]. These costs are a function of the `popcount` parameters and sieve specific parameters,

e.g. the angles that define bucketing strategies. Among heuristic lattice sieves, the three we have chosen are, the earliest and most conceptually simple [43], the most performant yet implemented [2], and the fastest known asymptotically [8].

Our analysis allows us to optimise the cost of classical and quantum search under various cost metrics to produce Figure 2 of Section 6. In this work we focus on cost metrics on the logical layer of quantum computation rather than on physical qubits and error correction costs, although we do discuss the effects that recent estimates for error correction [22] would have on our conclusions. In Section 6 we also discuss barriers to obtaining the reported quantum advantages in NNS. We provide the source code used to compute all data in this work as well as the raw data produced in Appendix G (source code) and as attachments to the electronic version of this document (source code and raw data).

Interpretation. Quantum computation appears to be more difficult than classical computation. As such, there will almost certainly be some minimal dimension, a crossover point, below which classical sieves outperform quantum sieves. We establish non-trivial crossover points for all of the sieves that we consider. Yet, on a high level, our results do not rule out the relevance of quantum sieves to lattice cryptanalysis. The crossover points that we observe are well below the dimensions commonly thought to achieve 128 bit security against quantum adversaries. However, our analysis is optimistic. It ignores the cost of quantum random access memory and the cost of quantum error correction. To illustrate the potential impact of error correction, we consider the cost of reading (but not processing) error syndromes during surface code quantum computation. Using surface code resource estimates based on a recent analysis of Shor's algorithm [22], we find that just reading these syndromes pushes the crossover point for the NNS algorithm underlying [8] to dimension 272. Applying the same metrics in dimension 784 we estimate a classical cost of $2^{259.1}$ and a quantum cost that is only a factor of $2^{8.2}$ smaller. This, together with widely held concerns about the feasibility of quantum random access memory, suggests that a quantum advantage for sieving is, indeed, tenuous at best.

Related works. As mentioned above, in [36, 35] the asymptotic complexities of many lattice sieve algorithms are calculated by adapting their complexity proofs to use Grover's algorithm. The role of qRAM and its implementation challenges are discussed in [23, 6, 41]. A series of recent works have obtained quantum resource estimates for cryptanalytic problems including key recovery attacks on AES [26], preimage attacks on SHA-2 and SHA-3 [5], the elliptic curve discrete logarithm problem [46], integer factorisation [21, 22], and the computational supersingular isogeny problem [32].

2 Preliminaries

2.1 Models of computation

We describe quantum algorithms as circuits using the Clifford+T gate set, but we augment this gate set with a table lookup operation. We describe classical algorithms as programs for RAM machines (random access memory machines).

Clifford+T+qRAM quantum circuits. Quantum circuits can be described at the *logical layer*, wherein an array of n qubits encodes a unit vector in $(\mathbb{C}^2)^{\otimes n}$, or at the *physical layer*, wherein the state space may be much larger. Ignoring qubit initialisation and measurement, a circuit is a sequence of unitary operations, one per unit time. Each unitary in the sequence is constructed by parallel composition of gates. At most one gate can be applied to each qubit per time step. The Clifford+T gate set

$$\mathbf{H} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}, \quad \mathbf{CNOT} = \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}, \quad \mathbf{T} = \begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix},$$

is commonly used to describe circuits at the logical layer. This gate set is universal for quantum computation when combined with qubit initialisation (of $|0\rangle$ and $|1\rangle$ states) and measurement in the computational basis. The prevalance of, and our use of, the Clifford+T gate set deserves further explanation.

The n qubit Pauli group, $\mathcal{P}_n$, is generated under matrix multiplication by the n fold Kronecker products of

$$\mathbf{I} = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, \quad \mathbf{Z} = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}.$$

Each element of $\mathcal{P}_n$ is unitary, so $\mathcal{P}_n$ is a subgroup of the unitary group $U(2^n)$. A unitary is said to be *Clifford* if it is in the normaliser of $\mathcal{P}_n$ in $U(2^n)$, $N_{U(2^n)}(\mathcal{P}_n) = \{U \in U(2^n) \colon U\mathcal{P}_n U^{-1} = \mathcal{P}_n\}$, for some n, i.e. if it maps the Pauli operators to the Pauli operators under conjugation. For example, in the single qubit case, the **H** gate is Clifford as $\mathbf{HXH}^{-1} = \mathbf{Z}$, $\mathbf{HZH}^{-1} = \mathbf{X}$, $\mathbf{HYH}^{-1} = -\mathbf{Y}$. The **S** and **CNOT** gates are also Clifford. The **T** gate is not Clifford.

Stabiliser codes, a class of quantum error correcting codes that includes the surface code, store logical quantum states in the mutual +1-eigenspace of a set of Pauli operators. In the context of fault tolerant computation with stabiliser codes, Pauli gates and single qubit Clifford gates are thought to be inexpensive. Controlled-NOT gates (**CNOT**) have a cost that grows with the distance between the control and target qubits, but one can generally assume that most of the cost of a computation will come from applying non-Clifford gates.

Among non-Clifford gates, the **T** gate is particularly easy to apply as it maps one qubit Pauli gates to one qubit Clifford gates under conjugation and, as a result, can be applied using the *gate teleportation protocol* from [25]. This

protocol involves only Clifford gates and the preparation of resource states. The resource states are prepared using a process called *magic state distillation* [14], which accounts for most of the physical layer cost of applying **T** gates.

In addition to Clifford+T gates, we allow unit cost table lookups in the form of qRAM (quantum access to classical RAM). The difference between RAM and qRAM is that qRAM can construct arbitrary superpositions of table entries. Suppose that $(R_0, \ldots, R_{2^n-1})$ are registers of a classical RAM and that each register encodes an ℓ bit binary string. We allow our Clifford+T circuits access to these registers in the form of an $(n + \ell)$ qubit qRAM gate that enacts

$$\sum_{j=0}^{2^n-1} \alpha_j |j\rangle |x\rangle \stackrel{qRAM}{\longrightarrow} \sum_{j=0}^{2^n-1} \alpha_j |j\rangle |x \oplus R_j\rangle . \tag{1}$$

Here $\sum_j \alpha_j |j\rangle$ is a superposition of addresses and x is an arbitrary ℓ bit string.

Quantum access to classical RAM is a powerful resource, and the algorithms that we describe below fail to achieve an advantage over their classical counterparts when qRAM is not available. We discuss this at greater length in Section 6.

RAM machines. We describe classical algorithms in terms of random access memory machines. For comparability with the Clifford+T gate set, we will work with a limited instruction set, e.g. {NOT, AND, OR, XOR, LOAD, STORE}. For comparability with qRAM, LOAD and STORE act on n bit registers.

Cost. The cost of a RAM program is the number of operations that it performs. One can similarly define the *gate cost* of a quantum circuit to be the number of gates that it performs. Both metrics are reasonable in isolation, but it is not clear how one should compare the two.

Jaques and Schanck recommend that quantum circuits be assigned a cost in units of RAM operations to account for the role that classical computers play in dispatching instructions to quantum memories [32]. This cost does not capture the real cost of quantum hardware. Nevertheless, it can be used as a baseline for comparisons between classical and quantum programs. Jaques and Schanck also recommend assigning unit cost to the identity gate, to account for the fact that error correction is an active process. The *depth-width cost* of a quantum circuit is the total number of gate operations that it performs including identity gates.

One can argue that active error correction is not essential. However, such a position undermines the gate cost as well as the depth-width cost, as it undermines the relevance of Clifford+T circuits in general. In this work we have chosen to optimise for the depth-width cost.

2.2 Black box search

A predicate on $\{0, 1, \ldots, N-1\}$ is a function $f : \{0, 1, \ldots, N-1\} \to \{0, 1\}$. The kernel, or set of roots, of f is $\mathrm{Ker}(f) = \{x : f(x) = 0\}$. We write $|f|$ for $|\mathrm{Ker}(f)|$. A black box search algorithm finds a root of a predicate without exploiting any

structure present in the description of the predicate itself. Of course, black box search algorithms can be applied when structure is known, and we will often use structure such as "f has M roots" or "f is expected to have no more than M roots" in our analyses. We will also use the fact that the set of predicates on any given finite set can be viewed as a Boolean algebra. We write $f \cup g$ for the predicate with kernel $\mathrm{Ker}(f) \cup \mathrm{Ker}(g)$ and $f \cap g$ for the predicate with kernel $\mathrm{Ker}(f) \cap \mathrm{Ker}(g)$.

Exhaustive search. An exhaustive search evaluates $f(0)$, $f(1)$, $f(2)$, and so on until a root of f is found. The order does not matter so long as each element of the search space is queried at most once. If f is a uniformly random predicate with M roots, then this process has probability $1 - \binom{N-M}{j}/\binom{N}{j} \geq 1 - (1 - M/N)^j$ of finding a root after exactly j evaluations of f. This is true even if M is not known.

Filtered search. If f is expensive to evaluate, we may try to decrease the cost of exhaustive search by applying a search filter. We say that a predicate g is a filter for f if $f \neq g$ and $|f \cap g| \geq 1$. We say that g recognises f with a false positive rate of

$$\rho_f(g) = 1 - \frac{|f \cap g|}{|g|},$$

and a false negative rate of

$$\eta_f(g) = 1 - \frac{|f \cap g|}{|f|}.$$

A filtered search evaluates $g(0), f(0), g(1), f(1), g(2), f(2)$, and so on until a root of $f \cap g$ is found. The evaluation of $f(i)$ can be skipped when i is not a root of g, which may reduce the cost of filtered search below that of exhaustive search.

Quantum search. Grover's quantum search algorithm is a black box search algorithm that provides a quadratic advantage over exhaustive search in terms of query complexity. Suppose that f is a predicate with M roots. Let $\mathbf{D}$ be any unitary transformation that maps $|0\rangle$ to $\frac{1}{\sqrt{N}} \sum_i |i\rangle$, let $\mathbf{R}_0 = \mathbf{I}_N - 2|0\rangle\langle 0|$ and let $\mathbf{R}_f$ be the unitary $|x\rangle \mapsto (-1)^{f(x)}|x\rangle$. Measuring $\mathbf{D}|0\rangle$ yields a root of f with probability M/N. Grover's quantum search algorithm amplifies this to constant probability by repeatedly applying the unitary $\mathbf{G}(f) = \mathbf{D}\mathbf{R}_0\mathbf{D}^{-1}\mathbf{R}_f$ [29]. Suppose that j repetitions are applied. The analysis in [29, 11] shows that measuring the state $\mathbf{G}(f)^j\mathbf{D}|0\rangle$ yields a root of f with probability $\sin^2((2j+1)\theta)$ where $\sin^2(\theta) = M/N$. Assuming $M \ll N$, the probability of success is maximised at $j \approx \frac{\pi}{4}\sqrt{N/M}$ iterations. Boyer, Brassard, Høyer, and Tapp show that a constant success probability can be obtained after $O(\sqrt{N/M})$ iterations even when M is not known [11].

Amplitude amplification. Brassard, Høyer, Mosca, and Tapp observed that the $\mathbf{D}$ subroutine of Grover's algorithm can be replaced with any algorithm that finds a root of f with positive probability. This generalisation of Grover's algorithm is called amplitude amplification. Let $\mathbf{A}$ be a quantum algorithm that makes no measurements and let p be the probability that measuring $\mathbf{A}|0\rangle$ yields a root of f. Let $\mathbf{G}(\mathbf{A}, f) = \mathbf{A}\mathbf{R}_0\mathbf{A}^{-1}\mathbf{R}_f$, where $\mathbf{R}_0$ and $\mathbf{R}_f$ are as in Grover's algorithm. Let θ be such that $\sin^2(\theta) = p$. Suppose that j iterations of $\mathbf{G}(\mathbf{A}, f)$ are applied to $\mathbf{A}|0\rangle$. The analysis in [12] shows that measuring the state $\mathbf{G}(\mathbf{A}, f)^j\mathbf{A}|0\rangle$ yields a root of f with probability $\sin^2((2j+1)\theta)$.

Filtered quantum search. The benefit of classical filtered search comes from the fact that we can omit a call to $f(i)$ when $g(i) = 1$. This sort of conditional branching cannot be used in a quantum algorithm. Nevertheless, a filter can be used to reduce the cost of quantum search. The idea is to use Grover's algorithm to prepare a uniform superposition over roots of g and then amplify the probability that measuring this superposition yields a root of f. Algorithm 1 presents pseudocode for this strategy. The cost and success probability of Algorithm 1 are given by Proposition 1.

Algorithm 1 FilteredQuantumSearch

Input: A predicate f and a filter g defined on $\{0, \ldots, N-1\}$. Integer parameters m_1 and m_2.
Output: A root of f or $\perp$.

function FilteredQuantumSearch($f, g; m_1, m_2$)
  Sample uniformly random integers j and k with $0 \le j < m_1$ and $0 \le k < m_2$.
  Let $\mathbf{A}_j = \mathbf{G}(g)^j\mathbf{D}$.
  Let $\mathbf{B}_k = \mathbf{G}(\mathbf{A}_j, f \cap g)^k$.
  Prepare the state $|\psi\rangle = \mathbf{B}_k\mathbf{A}_j|0\rangle$.
  Let r be the result of measuring $|\psi\rangle$ in the computational basis.
  if $f(r) = 0$ **then**
    return r
  return $\perp$

Proposition 1. *Let P and Q be positive integers. Let $\gamma \ge 1$ be a real number. Suppose that f and g are predicates on a domain of size N and that g is a filter for f. If $|f \cap g| \ge Q$ and $P/\gamma \le |g| \le \gamma P$, then Algorithm 1 finds a root of f with probability at least $1/8$ and has a cost that is dominated by $\approx \frac{\gamma}{2}\sqrt{N/Q}$ times the cost of $\mathbf{G}(g)$ or by $\approx \frac{4}{3}\sqrt{\gamma P/Q}$ times the cost of $\mathbf{R}_{f\cap g}$.*

Proof. See Appendix A. □

The approximations in Proposition 1 refer to the constants $\frac{1}{2}$ and $\frac{4}{3}$. These are approximations to numerically optimised quantities ($0.52\ldots$ and $1.33\ldots$).

Remark 1. It is helpful to understand when we can ignore the cost of $\mathbf{R}_{f\cap g}$ in Proposition 1. Roughly speaking, if evaluating f is k times more expensive than evaluating g, then the cost of calls to $\mathbf{G}(g)$ will dominate when $N > k^2\,|g|$. Note that in a classical filtered search the cost of evaluating g dominates when $N > k\,|g|$.

2.3 Geometric figures on the sphere

Our analysis of the `popcount` filter requires some basic facts about the size of some geometric figures on the sphere. We measure the volume of subsets of $\mathcal{S}^{d-1} = \{v \in \mathbb{R}^d \colon \|v\| = 1\}$ using the $(d-1)$ dimensional spherical probability measure[1] μ^{d-1}. The angular distance of u, v is $\theta(u,v) = \arccos\left(\langle u, v\rangle / \left(\|u\|\|v\|\right)\right)$, $\arccos(x) \in [0, \pi]$. The spherical cap of angle θ about $u \in \mathcal{S}^{d-1}$ is $\mathcal{C}^{d-1}(u,\theta) = \{v \in \mathcal{S}^{d-1} : \theta(u,v) \leq \theta\}$. The measure of a spherical cap is

$$C_d(u,\theta) := \mu^{d-1}(\mathcal{C}^{d-1}(u,\theta)) = \frac{1}{\sqrt{\pi}} \frac{\Gamma(\frac{d}{2})}{\Gamma(\frac{d-1}{2})} \int_0^\theta \sin^{d-2}(t)\ \mathrm{d}t.$$

We will often interpret $C_d(u,\theta)$ as the probability that v drawn uniformly from $\mathcal{S}^{d-1}$ satisfies $\theta(u,v) \leq \theta$. We denote the density of the event $\theta(u,v) = \theta$ by

$$A_d(u,\theta) := \frac{\partial}{\partial\theta} C_d(u,\theta) = \frac{1}{\sqrt{\pi}} \frac{\Gamma(\frac{d}{2})}{\Gamma(\frac{d-1}{2})} \sin^{d-2}(\theta).$$

Note that $C_d(u,\theta)$ does not depend on u, so we may write $C_d(\theta)$ and $A_d(\theta)$ without ambiguity. The wedge formed by the intersection of two caps is $\mathcal{W}^{d-1}(u,\theta_u,v,\theta_v) = \mathcal{C}^{d-1}(u,\theta_u) \cap \mathcal{C}^{d-1}(v,\theta_v)$. The measure of a wedge only depends on $\theta = \theta(u,v)$, θ_u, and θ_v, so we denote it

$$W_d(\theta,\theta_u,\theta_v) = \mu^{d-1}(\mathcal{W}^{d-1}(u,\theta_u,v,\theta_v)).$$

We will often interpret $W_d(\theta,\theta_u,\theta_v)$ as is the probability that w drawn uniformly from $\mathcal{S}^{d-1}$ satisfies $\theta(u,w) \leq \theta_u$ and $\theta(v,w) \leq \theta_v$. Note that $\theta \geq \theta_u + \theta_v \Rightarrow W_d(\theta,\theta_u,\theta_v) = 0$. An integral representation of $W_d(\theta,\theta_u,\theta_v)$ is given in Appendix B.

2.4 Lattice sieving and near neighbour search on the sphere

Any set of $m \leq d$ linearly independent vectors $\{b_1,\ldots,b_m\} \in \mathbb{R}^d$ can be thought of as a basis of a rank m, dimension d Euclidean lattice. We consider full rank lattices, $m = d$. Letting these m vectors form the columns of a matrix B, the lattice they form is $\Lambda = B \cdot \mathbb{Z}^m$. The shortest vector problem in Λ asks for a non-zero vector $v \in \Lambda$ of minimal Euclidean norm.

A lattice sieve uses B to sample an exponentially sized list L of N long vectors of Λ, e.g. via [33], and iterates a procedure that searches for combinations

[1] By "probability measure" we mean that $\mu^{d-1}(\mathcal{S}^{d-1}) = 1$.

in L that give shorter lattice vectors. In particular for all $(u, v) \in L \times L$ (we always consider $u \neq v$) it checks whether $\|u - v\| < \|u\|$, or equivalently by inner product, $2\langle u, v\rangle - \|v\|^2 > 0$. If $\|u - v\| \geq \max\{\|u\|, \|v\|\}$ then we say (u, v) are reduced, else they are reducible.

A heuristic used to analyse lattice sieves states the lattice vectors in L are i.i.d. uniformly distributed in some thin annulus [43]. Once normalised we consider $L \subset \mathcal{S}^{d-1}$. This heuristic becomes invalid when the lattice vectors considered by our sieve are short, at which point we assume we have solved SVP. Under this heuristic, (u, v) are reducible if and only if $\theta(u, v) < \pi/3$.

It is this heuristic and the relation between reducibility and angular distance that allows us to phrase lattice sieves in the language of (angular) NNS on the sphere. Following [43] we may therefore define a naïve NNS procedure that checks all pairs in L as Algorithm 2. Such a procedure (wrapped in a simple algorithm that takes each $(u, v) \in L'$ and returns a new list formed of $u - v$) must be iterated $\mathsf{poly}(d)$ times with $\theta = \pi/3 - \varepsilon$ to find short vectors. The `popcount` filter, introduced in Section 2.5, acts as a first approximation to $\theta(\cdot, \cdot)$.

Algorithm 2 AllPairSearch

Input: A list $L = (v_1, v_2, \ldots v_N) \subset \mathcal{S}^{d-1}$ of N points. Parameter $\theta \in (0, \pi/2)$.
Output: A list of pairs $(u, v) \in L \times L$ with $\theta(u, v) \leq \theta$.

function AllPairSearch($L; \theta$)
  $L' \leftarrow \emptyset$
  for $1 \leq i < N$ **do**
    $L_i \leftarrow (v_{i+1}, \ldots, v_N)$
    Search L_i for any number of u that satisfy $\theta(u, v_i) \leq \theta$.
    For each such u found, add (u, v_i) to L'.
    If $|L'| \geq N$, **return** L'.
  return L'

Given our heuristic that elements of L are i.i.d. uniform on $\mathcal{S}^{d-1}$, we must determine which N guarantee we find near neighbours, and in particular, guarantee we *continue* to find near neighbours throughout the iterations. A heuristic used to analyse lattice sieves [43, 8] states that $N = (1/C_d(\pi/3))^{1+o(1)}$ is sufficient to find N reducible pairs, so that Algorithms 2, 3, 4 may be iterated.

In this work we set $N \in (2/C_d(\pi/3), 4/C_d(\pi/3))$, i.e. we ignore the $o(1)$ term. The multiplicative constant in $(2, 4)$ represents, in part, list growth due to `popcount` false negatives and is determined by the first term of Equation 7. For our experiments this constant is always less than 4.

2.5 The `popcount` filter

A locality sensitive hashing (LSH) scheme for the angular distance on $\mathcal{S}^{d-1}$ is a family of hash functions $\mathcal{H}$, defined on $\mathcal{S}^{d-1}$, for which $\Pr_{h \leftarrow \mathcal{H}}[h(u) = h(v)] =$

$1 - \theta(u, v)/\pi$. Charikar [15] constructed an LSH scheme for the angular distance using the fact that $\theta(u, v)/\pi$ is the probability that uniformly random u and v lie in opposite hemispheres. In Charikar's scheme,

$$\mathcal{H} = \left\{ u \mapsto \mathrm{sgn}(\langle r, u \rangle) : r \in \mathcal{S}^{d-1} \right\},$$

where $\mathrm{sgn}(x) = 1$ if $x \geq 0$ and $\mathrm{sgn}(x) = 0$ if $x < 0$.

Charikar also observed that one can estimate $\theta(u, v)/\pi$ by choosing a random hash function $h = (h_1, \ldots, h_n) \in \mathcal{H}^n$ and measuring the Hamming distance between $h(u) = (h_1(u), \ldots, h_n(u))$ and $h(v) = (h_1(v), \ldots, h_n(v))$. Each bit $h_i(u) \oplus h_i(v)$ is Bernoulli distributed with parameter $p = \theta(u, v)/\pi$. In the limit of large n, the normalised Hamming weight $wt(h(u) \oplus h(v))/n$ converges to a normal distribution with mean p and standard deviation $\sqrt{p(1-p)/n}$.

In the sieving literature, the process of filtering NNS using a threshold on the value of $wt(h(u) \oplus h(v))$ is referred to as the "XOR and population count trick" [19, 18, 2]. We write $\texttt{popcount}_{k,n}(u, v; h)$ for a search filter of this type

$$\texttt{popcount}_{k,n}(u, v; h) = \begin{cases} 0 & \text{if } \sum_{i=1}^{n} h_i(u) \oplus h_i(v) \leq k, \\ 1 & \text{otherwise.} \end{cases}$$

When the n hash functions are fixed we write $\texttt{popcount}_{k,n}(u, v)$. The threshold, k, is chosen based on the desired false positive and false negative rates. Heuristically, if one's goal is to detect points at angle at most θ, one should take $k/n \approx \theta/\pi$. If $k/n \ll \theta/\pi$ then the false negative rate will be large, and many neighbouring pairs will be missed. An important consequence of missing potential reductions is that the N required to iterate Algorithms 2, 3, 4 increases. In Section 5 this increase is captured in the quantity $\ell(k, n)$. If $k/n \gg \theta/\pi$ then the false positive rate will be large, and the full inner product test will be applied often. We calculate these false positive and negative rates in Section 4. These calculations and the fact that `popcount` is significantly cheaper than an inner product makes `popcount` a good candidate for use as a filter under the techniques of Section 2.2. Furthermore it is the filter used in the most performant sieves to date [18, 2].

3 Circuits for `popcount`

Consider a program for $\texttt{popcount}_{k,n}(u, v)$. This program loads u and v from specified memory addresses, computes $h(u)$ and $h(v)$, computes the Hamming weight of $h(u) \oplus h(v)$, and checks whether it is less than or equal to k. Recall $h(u)$ is defined by n inner products. If the `popcount` procedure is executed many times for each u, then it may be reasonable to compute $h(u)$ once and store it in memory. Moreover, if u is fixed for many sequential calls to the procedure, then it may be reasonable to cache $h(u)$ between calls. The algorithms that we consider in Section 5 use both such optimisations. In this section we describe RAM programs and quantum circuits that work for fixed $h(u)$. These routines load $h(v)$ from memory, compute the Hamming weight of $h(u) \oplus h(v)$, and check whether the Hamming weight is less than or equal to k. We ignore the cost of computing $h(u)$ here and in the remainder.

3.1 Quantum circuit for `popcount`

Loading $h(v)$ costs a single qRAM gate. Computing $h(u) \oplus h(v)$ can then be done in-place using a sequence of $\mathbf{X}$ gates that encode $h(u)$. The bulk of the effort is in computing the Hamming weight. For that we use a tree of in-place adders. The final comparison is also computed with an adder, although only one bit of the output is needed. See Figure 1 for a full description of the circuit.

We use the Cuccaro–Draper–Kutin–Petrie adder [16], with "incoming carry" inputs, to compute the Hamming weight. We argue in favour of this choice of adder in Appendix D. We use the Häner–Roetteler–Svore [30] carry bit circuit for implementing the comparison.

We will later use `popcount` within filtered quantum searches by defining predicates of the form $g(i) = \texttt{popcount}_{k,n}(u, v_i)$, $i \in \{0, \ldots, N-1\}$. To simplify that later discussion, we cost the entire Grover iteration $\mathbf{G}(g) = \mathbf{D}\mathbf{R}_0\mathbf{D}^{-1}\mathbf{R}_g$ here. In Appendix C we introduce the (possibly multiply controlled) Toffoli gate and discuss the Toffoli count for $\mathbf{G}(g)$, which in turn gives the $\mathbf{T}$ count for $\mathbf{G}(g)$.

The cost of $\mathbf{R}_g$. The $\mathbf{R}_g$ subroutine is computed by running the `popcount` circuit in Figure 1 and then uncomputing the carry bit, addition tree, and $\mathbf{X}$ gates. The circuit uses in-place i bit adders[2] for $i \in \{1, \ldots, \ell - 1\}$. The width of the circuit is given in Appendix C. The depth of the circuit is

$$\text{depth} = 2 + d(\text{CARRY}) + \sum_{i=1}^{\ell-1} 2d(\text{ADD}_i), \tag{2}$$

where $d(\cdot)$ denotes the depth of its argument. The factor of 2 accounts for uncomputation of the ADD_i circuits. The CARRY circuit is only cost once as the carry bit is computed directly into the $|-\rangle$ state during the CARRY circuit itself. The summand 2 accounts for the $\mathbf{X}$ gates used to compute, and later uncompute, $h(u) \oplus h(v)$.

The cost of $\mathbf{D}\mathbf{R}_0\mathbf{D}^{-1}$. Recall that $\mathbf{D}$ can be any circuit that maps $|0\rangle$ to the uniform distribution on the domain of the search predicate. While there is no serious difficulty in sampling from the uniform distribution on $\{0, \ldots, N-1\}$ for any integer N, when costing the circuit we assume that N is a power of two. In this case $\mathbf{D}$ is simply $\log_2 N$ parallel $\mathbf{H}$ gates. The reflection $\mathbf{R}_0$ is implemented as a multiply controlled Toffoli gate (Appendix C) that targets an ancilla initialised in the $|-\rangle$ state. We use Maslov's multiply controlled Toffoli from [40]. The depth and width of $\mathbf{D}\mathbf{R}_0\mathbf{D}^{-1}$ are both $O(\log N)$; our software calculates the exact value.

[2] An in-place i bit quantum adder takes two i bit inputs, initialises an ancilla qubit in the $|0\rangle$ state, and returns the addition result in an $i+1$ bit register that includes the new ancilla and overlaps with i bits of the input.

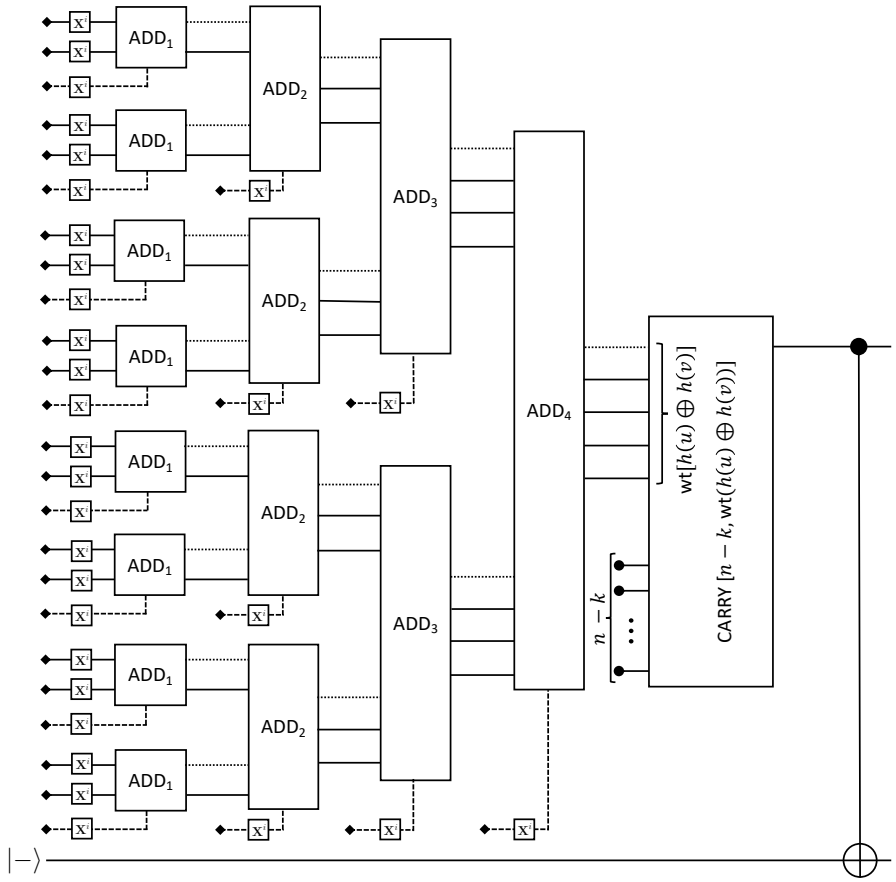


Fig. 1: A quantum circuit for popcount. This circuit computes $h(u) \oplus h(v)$ for a fixed n bit $h(u)$, computes the Hamming weight of $h(u) \oplus h(v)$, and checks whether the Hamming weight is less than or equal to k. Here $n = 2^\ell - 1 = 31$. The input qubits are represented as lines ending with a black diamond. The dashed lines represent incoming carry inputs, and the dotted lines represent carry outputs. Not all of the output wires are drawn. For space efficiency, some of the input qubits are fed into the incoming carry qubits of the adders (dashed lines). The $\mathbf{X}^i$ mean that gate $\mathbf{X}$ is applied to input qubit i if bit i of $h(u)$ is 1. The circuit uses a depth $\ell - 1$ binary tree of full bit adders from [16], where ADD_i denotes an i bit full adder. The output $wt(h(u) \oplus h(v))$ from the tree of adders together with the binary representation of the number $n - k$ are finally fed into the input of the CARRY circuit from [30], which computes the carry bit of $n - k + wt(h(u) \oplus h(v))$ (the carry bit will be 0 if $wt(h(u) \oplus h(v)) \leq k$, and 1 otherwise). The final **CNOT** is for illustration only. In actuality, the carry bit is computed directly into an ancilla that is initialised in the $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$ state, so we can obtain the needed phase kickback. The tree of adders and the initial $\mathbf{X}$ gates, but not the CARRY circuit, are run in reverse to clean up scratch space and return the inputs to their initial state. The uncomputation step is not depicted here.

3.2 RAM program for `popcount`

Recall that we use a RAM instruction set that consists of simple bit operations and table lookups. A Boolean circuit for `popcount` is schematically similar to Figure 1. Let $\ell = \lceil \log_2 n \rceil$ and $t = \lceil \log_2 k \rceil$. Loading $h(v)$ has cost 1. Computing $h(v) \oplus h(w)$ takes n XOR gates and has depth 1. Computing the Hamming weight and comparing it with a fixed k costs less than $(n - \ell - 1)c_{FA} + \ell$ gates, where $c_{FA} \approx 10$ is the number of gates in a full adder. The circuit depth is $(\ell - 1)(d_{\text{sum}} + d_{\text{carry}}) + 1$ [45]. We assume $d_{\text{sum}} = d_{\text{carry}} = 1$. Thus, the overall gate count is $11n - 9\ell - 10$ and the depth is 2ℓ.

3.3 Cost of inner products

We will also make use of rough estimates for the cost of computing inner products in dimension d. For this, we assume 32 bits of precision are sufficient. We then assume schoolbook multiplication is used for scalar products, costing approximately 32^2 gates. We then assume the cost of a full inner product is approximately $32^2\, d$, i.e. we ignore the cost of the final summation, assuming it is dwarfed by the multiplications.

4 The accuracy of `popcount`

Here we give an analysis of the `popcount` technique based on some standard simplifying assumptions. We are particularly interested in the probability that a `popcount` filter identifies a random pair of points as potential neighbours. We are also interested in the probability that a pair of potential neighbours are not actually neighbours, i.e. the false positive rate. Our software computes all of the quantities in this section to high precision.

Let $P_{k,n}(u, v)$ be the probability that $\texttt{popcount}_{k,n}(u, v; h)$ returns 0 for a uniformly random h (recall $\texttt{popcount}_{k,n}(u, v; h) = 0$ if u, v pass the filter). In other words, let $h = (h_1, \ldots, h_n)$ be a collection of independent random variables that are distributed uniformly on the sphere, and define

$$P_{k,n}(u, v) = 1 - \mathbb{E}\left[\texttt{popcount}_{k,n}(u, v; h)\right].$$

The hyperplane defined by h_i separates u and v with probability $\theta(u, v)/\pi$, and $\texttt{popcount}_{k,n}(u, v)$ is 0 if no more than k of the hyperplanes separate u and v. Hence,

$$P_{k,n}(u, v) = \sum_{i=0}^{k} \binom{n}{i} \left(\frac{\theta(u, v)}{\pi}\right)^i \left(1 - \frac{\theta(u, v)}{\pi}\right)^{n-i}.$$

Note that $P_{k,n}(u, v)$ depends only on the angle between u and v, so it makes sense to define $P_{k,n}(\theta)$ as well. The main heuristic in our analysis of `popcount` is that $P_{k,n}(u, v)$ is a good approximation to the probability that $\texttt{popcount}_{k,n}(u, v; h)$ returns 0 for *fixed* h and *varying* u and v. Under this assumption, all of the

quantities in question can be determined by integrating $P_{k,n}(u, v)$ over different regions of the sphere.

Let $\hat{P}_{k,n}$ denote the event that $\texttt{popcount}_{k,n}(u, v; h)$ returns 0 for uniformly random u, v, and h. Let $\hat{R}_\theta$ be the event that $\theta(u, v) \leq \theta$. Recall that $\Pr[\hat{R}_\theta] = C_d(\theta)$, and observe that $\Pr[\hat{R}_\theta]$ is a cumulative distribution with associated density $A_d(\theta) = \frac{\partial}{\partial\theta} C_d(\theta)$. We find

$$\begin{aligned}
\Pr[\hat{P}_{k,n}] &= \int_{\mathcal{S}} \int_{\mathcal{S}} P_{k,n}(u, v) \, \mathrm{d}\mu(v) \, \mathrm{d}\mu(u) \\
&= \int_{\mathcal{S}} \left(\int_0^\pi P_{k,n}(\theta) A_d(\theta) \, \mathrm{d}\theta \right) \mathrm{d}\mu(u) \\
&= \int_0^\pi P_{k,n}(\theta) A_d(\theta) \, \mathrm{d}\theta. \qquad (3)
\end{aligned}$$

Let u, v such that $\theta(u, v) \leq \varphi$ be neighbours. The false positive rate is $1 - \Pr[\hat{P}_{k,n} \mid \hat{R}_\varphi]$. The quantity $\Pr[\hat{P}_{k,n} \wedge \hat{R}_\varphi]$ can be calculated by changing the upper limit of integration in Equation 3. It follows that

$$1 - \Pr[\hat{P}_{k,n} \mid \hat{R}_\varphi] = 1 - \frac{1}{C_d(\varphi)} \int_0^\varphi P_{k,n}(\theta) A_d(\theta) \, \mathrm{d}\theta. \qquad (4)$$

In Section 5 we consider u and v that are uniformly distributed in a cap of angle $\beta < \pi/2$, rather than the uniformly distributed on the sphere. Let $\hat{B}_{w,\beta}$ be the event that u and v are uniformly distributed in a cap of angle β about w. We have

$$\begin{aligned}
\Pr[\hat{B}_{w,\beta}] &= \int_{\mathcal{S}} \int_{\mathcal{S}} \mathbb{1}\left\{ w \in \mathcal{W}^{d-1}(u, \beta, v, \beta) \right\} \, \mathrm{d}\mu(v) \, \mathrm{d}\mu(u) \\
&= \int_0^{2\beta} W_d(\theta, \beta, \beta) A_d(\theta) \, \mathrm{d}\theta. \qquad (5)
\end{aligned}$$

In the second line we have used the fact that $\beta < \pi/2$ and $W(\theta, \theta_1, \theta_2)$ is zero when $\theta \geq \theta_1 + \theta_2$. The quantity $\Pr[\hat{B}_{w,\beta} \wedge \hat{R}_\varphi]$ can be computed by changing the upper limit of integration in Equation 5 from 2β to $\min\{2\beta, \varphi\}$. We note that $\hat{B}_{w,\beta}$ has no dependence on w and therefore may also be written $\hat{B}_\beta$. The conditional probability that popcount returns 0 for u and v, given they are uniformly distributed in a cap, $\Pr[\hat{P}_{k,n} \mid \hat{B}_\beta]$, can be computed using Equation 5 and

$$\Pr[\hat{P}_{k,n} \wedge \hat{B}_\beta] = \int_0^{2\beta} P_{k,n}(\theta) W_d(\theta, \beta, \beta) A_d(\theta) \, \mathrm{d}\theta. \qquad (6)$$

The quantity $\Pr[\hat{P}_{k,n} \wedge \hat{B}_\beta \wedge \hat{R}_\varphi]$ can be computed by changing the upper limit of integration in Equation 6 from 2β to $\min\{2\beta, \varphi\}$. The false positive rate for popcount when restricted to a cap is $1 - \Pr[\hat{P}_{k,n} \mid \hat{B}_{w,\beta} \wedge \hat{R}_\varphi]$.

5 Tuning popcount for NNS

We now use the circuit sizes from Section 3 and the probabilities from Section 4 to optimise popcount for use in NNS algorithms. Our analysis is with respect to points sampled independently from the uniform distribution on the sphere. We further restrict our attention to *list-size preserving* parameterisations, which take an input list of size N and return an output list of (expected) size N.

We use the notation for events introduced in Section 4. In particular, we write $\hat{R}_\theta$ for the event that a uniformly random pair of vectors are neighbours, i.e. that they lie at angle less than or equal to θ of one another; $\hat{P}_{k,n}$ for the event that popcount identifies a uniformly random pair of vectors as potential neighbours; $\hat{B}_\beta$ for the event that a uniformly random pair of vectors lie in a uniformly random cap of angle β; and $\hat{B}_{w,\beta}$ for the same event except we highlight the cap is centred on w. Throughout this section we use $\texttt{popcount}_{k,n}(u,\cdot)$, for various fixed u, as a filter for the search predicate $\theta(u,\cdot) \leq \theta$. We write $\eta(k,n)$ for the false negative rate of popcount. We assume that $\theta(u,v) \leq \theta$ is computed using an inner product test.

Our goal in the following is to minimise the cost of list-size preserving NNS algorithms as a function of the input list size, the popcount parameters k and n, and sometimes angles that define caps. In a list of N points there are $\binom{N}{2}$ ordered pairs. We expect $\binom{N}{2}\Pr[\hat{R}_\theta] = \binom{N}{2}C_d(\theta)$ of these to be neighbours, and we expect a $1-\eta(k,n)$ fraction of neighbours to be detected by popcount. List-size preserving parmaterisations that use a popcount filter must therefore take an input list of size at least

$$\ell(k,n) = \frac{2}{1-\eta(k,n)}\frac{1}{C_d(\theta)}. \tag{7}$$

We assume that list-size preserving parameterisations take $N = \ell(k,n)$. Note that $\eta(k,n) = 1 - \Pr[\hat{P}_{k,n} \mid \hat{R}_\theta]$ when the search is over a set of points uniformly distributed on the sphere, and $\eta(k,n) = 1 - \Pr[\hat{P}_{k,n} \mid \hat{R}_\theta \wedge \hat{B}_\beta]$ when the search is over a set of points uniformly distributed in a cap of angle β (left implicit).

Throughout, c_1 represents the gate cost of the inner product test from Section 3.3, $c_2(k,n)$ the gate cost of popcount from Section 3.2, q_1 the quantum cost of the reflection $\mathbf{R}_{f\cap g}$ using the inner product test from Section 3.3 and $q_2(k,n)$ the quantum cost of $\mathbf{G}(g)$ from Section 3.1. We note that c_1, q_1 have a dependence on d which we suppress. We also write $q_0(m)$ for the number of $\mathbf{G}(g)$ iterations that are applied during a search on a set of size m. Proposition 1 can be used to estimate $q_0(m)$. To apply it, we need integer parameters $P \approx |g|$ (with relative error $\gamma \geq 1$) and $Q \leq |f \cap g|$. Proposition 1 also establishes that the search finds a neighbour of the input, when one exists, with probability at least $1/8$. We make two assumptions that may cause us to underestimate the cost of search in all three of the following analyses. First, we ignore the probability that the search fails to find a neighbour when one exists. Second, we assume that $\gamma = 1$, i.e. that $P = |g|$.

5.1 AllPairSearch

As a warmup, we optimise AllPairSearch. Asymptotically its complexity is $2^{(0.415\ldots+o(1))d}$ classically and $2^{(0.311\ldots+o(1))d}$ quantumly. We describe implementations of Line 5 of Algorithm 2 based on filtered search and filtered quantum search, and optimise `popcount` relative to these implementations.

Filtered search. Suppose that Line 5 applies $\texttt{popcount}_{k,n}(v_i, \cdot)$ to each of v_{i+1} through v_N and then applies an inner product test to each vector that passes. With an input list of size $N = \ell(k, n)$, we expect this implementation to test all $\binom{N}{2}$ pairs before finding N neighbouring pairs. Moreover, we expect the `popcount` filter to identify $\binom{N}{2} \Pr[\hat{P}_{k,n}]$ potential neighbours, and to perform an equal number of inner product tests. The optimal parameters are obtained by minimising

$$\left(c_1 \Pr[\hat{P}_{k,n}] + c_2(k, n)\right) \cdot \binom{\ell(k, n)}{2}. \tag{8}$$

Filtered quantum search. Suppose that Line 5 is implemented using Algorithm 1. Specifically, we take the predicate f to be $\theta(v_i, \cdot) \leq \theta$ with domain L_i. We take the filter g to be $\texttt{popcount}_{k,n}(v_i, \cdot)$. Each call to the search routine returns at most one neighbour of v_i. To find all detectable neighbours of v_i in L_i we must repeat the search $|L_i| \cdot |f \cap g|$ times. This is expected to be $|L_i| \cdot \Pr[\hat{P}_{k,n} \wedge \hat{R}_\theta]$. Known neighbours of v_i can be removed from L_i to avoid a coupon collector scenario. We consider an implementation in which searches are repeated until a search fails to find a neighbour of v_i.

We expect to call the search subroutine $|L_i| \cdot \Pr[\hat{P}_{k,n} \wedge \hat{R}_\theta] + 1$ times in iteration i. Proposition 1 with $P = |L_i| \cdot \Pr[\hat{P}_{k,n}]$, $Q = 1$, and $\gamma = 1$ gives $q_0(|L_i|) = \frac{1}{2}\sqrt{|L_i|}$ iterations of $\mathbf{G}(g)$. As i ranges from 1 to $N - 1$ the quantity $|L_i|$ takes each value in $\{1, \ldots, N - 1\}$. Our proposed implementation therefore performs an expected

$$\sum_{j=1}^{N-1} \frac{1}{2}\sqrt{j}\left(j \Pr[\hat{P}_{k,n} \wedge \hat{R}_\theta] + 1\right)$$
$$= \Pr[\hat{P}_{k,n} \wedge \hat{R}_\theta]\left(\frac{1}{5}N^{5/2} - \frac{1}{4}N^{3/2}\right) + \frac{1}{3}N^{3/2} + O(\sqrt{N}) \tag{9}$$

applications of $\mathbf{G}(g)$; the expansion is obtained by the Euler–Maclaurin formula. When $N = \ell(k, n)$ we expect $N \cdot \Pr[\hat{P}_{k,n} \wedge \hat{R}_\theta] = 2 + O(1/N)$. The right hand side of Equation 9 is then $\frac{11}{15}N^{3/2} + O(\sqrt{N})$.

Proposition 1 also provides an estimate for the rate at which reflections about the true positives, $\mathbf{R}_{f \cap g}$ are performed. With P and Q as above, we find that $\mathbf{R}_{f \cap g}$ is performed at roughly $p(k, n) = \sqrt{\Pr[\hat{P}_{k,n}]}$ the rate of calls to $\mathbf{G}(g)$. The

optimal `popcount` parameters (up to some small error due to the $O(\sqrt{N})$ term in Equation 9) are obtained by minimising the total cost

$$\frac{11}{15}\left(q_1 p(k,n) + q_2(k,n)\right) \cdot \ell(k,n)^{3/2}. \tag{10}$$

5.2 RandomBucketSearch

One can improve AllPairSearch by *bucketing* the search space such that vectors in the same bucket are more likely to be neighbours. For example, one could pick a hemisphere H and divide the list into $L_1 = L \cap H$ and $L_2 = L \setminus L_1$. These lists would be approximately half the size of the original and the combined cost of AllPairSearch within L_1 and then within L_2 would be half the cost of an AllPairSearch within L. However, this strategy would fail to detect the expected θ/π fraction of neighbours that lie in opposite hemispheres.

Becker, Gama, and Joux [9] present a very efficient generalisation of this strategy. They propose bucketing the input list into subsets of the form $\{v \in L : \texttt{popcount}_{k,n}(0, v; p) = 0\}$ with varying choices of p. This bucketing strategy is applied recursively until the buckets are of a minimum size. Neighbouring pairs are then found by an AllPairSearch.

A variant of the Becker–Gama–Joux algorithm that uses buckets of the form $L \cap C_d(f, \theta_1)$, with randomly chosen f and fixed θ_1, was proposed and implemented in [2]. This variant sometimes called `bgj1`. Here we call it RandomBucketSearch. This algorithm has asymptotic complexity $2^{(0.349\ldots+o(1))d}$ classically [2] and $2^{(0.301\ldots+o(1))d}$ quantumly. This is worse than the Becker–Gama–Joux algorithm, but RandomBucketSearch is conceptually simple and still provides an enormous improvement over AllPairSearch. Pseudocode is presented in Algorithm 3.

Algorithm 3 RandomBucketSearch

Input: A list $L = (v_1, v_2, \ldots v_N) \subset \mathcal{S}^{d-1}$ of N points. Parameters $\theta, \theta_1 \in (0, \pi/2)$ and $t \in \mathbb{Z}_+$.

Output: A list of pairs $(u, v) \in L \times L$ with $\theta(u, v) \leq \theta$.

function RandomBucketSearch($L; \theta, \theta_1, t$)
  $L' \leftarrow \emptyset$
  for $1 \leq i \leq t$ **do**
    Sample f uniformly on $\mathcal{S}^{d-1}$
    $L_f \leftarrow L \cap \mathcal{C}^{d-1}(f, \theta_1)$
    for j such that $v_j \in L_f$ **do**
      $L_{f,j} \leftarrow \{v_k \in L_f : j < k \leq N\}$
      Search $L_{f,j}$ for any number of u that satisfy $\theta(v_j, u) \leq \theta$
      For each such u found, add (v_j, u) to L'.
      If $|L'| \geq N$, **return** L'.
  return L'

Description of Algorithm 3. The algorithm takes as input a list of N points uniformly distributed on the sphere. A random bucket centre f is drawn uniformly from $\mathcal{S}^{d-1}$ in each of the t iterations of the outer loop. The choice of f defines a bucket in Line 5, $L_f = L \cap \mathcal{C}^{d-1}(f, \theta_1)$, which is of expected size $N \cdot C_d(\theta_1)$. For each $v_j \in L_f$, the inner loop searches a set $L_{f,j} \subset L_f$ for neighbours of v_j. The quantity $|L_{f,j}|$ takes each value in $\{1, \ldots, |L_f| - 1\}$ as v_j ranges over L_f. The inner loop is identical to the loop in AllPairSearch apart from indexing and the fact that elements of L_f are known to be in the cap $\mathcal{C}^{d-1}(f, \theta_1)$.

A bucket L_f is expected to contain $\binom{N}{2} \Pr[\hat{R}_\theta \wedge \hat{B}_{f,\theta_1}]$ neighbouring pairs. Only a $1 - \eta(k, n)$ fraction of these are expected to be identified by the `popcount` filter. Our software calculates $\Pr[\hat{R}_\theta \wedge \hat{B}_{f,\theta_1}]$ numerically, but when $\theta_1 > \theta$ it is reasonable to assume that $\Pr[\hat{R}_\theta \wedge \hat{B}_{f,\theta_1}] \approx C_d(\theta) \cdot W_d(\theta, \theta_1, \theta_1)$. We use this approximation for the remainder of this section. The expected number of neighbouring pairs in L_f that are detected by the `popcount` filter is approximately $\binom{N}{2} ((1 - \eta(k, n)) \cdot C_d(\theta) \cdot W_d(\theta, \theta_1, \theta_1))$. When $N = \ell(k, n)$ this is approximately $N \cdot W_d(\theta, \theta_1, \theta_1)$. If all detectable neighbours are found by the search routine then the algorithm is list-size preserving when $N = \ell(k, n)$ and $t = 1/W_d(\theta, \theta_1, \theta_1)$.

We can now derive optimal `popcount` parameters for various implementations of Line 8.

Filtered search. Suppose that Line 8 of Algorithm 3 applies $\texttt{popcount}_{k,n}(v_j, \cdot)$ to each element of $L_{f,j}$ and then applies an inner product test to each vector that passes. This implementation applies `popcount` tests to all $\binom{|L_f|}{2} \approx \binom{N \cdot C_d(\theta_1)}{2}$ pairs of elements in L_f and finds all of the neighbouring pairs that pass. In the process it applies inner product tests to a $p(\theta_1, k, n) = \Pr[\hat{P}_{k,n} \mid \hat{B}_{f,\theta_1}]$ fraction of pairs. The cost of populating buckets in one iteration of Line 5 is $c_1 \cdot \ell(k, n)$. The cost of one search in Line 8 is $(c_1 \cdot p(\theta_1, k, n) + c_2(k, n)) \binom{NC_d(\theta_1)}{2}$. With the list-size preserving parameters N and t given above, the optimal θ_1, k, and n can be obtained by minimising the total cost

$$\frac{1}{W_d(\theta, \theta_1, \theta_1)} \left(c_1 \cdot \ell(k, n) + (c_1 \cdot p(\theta_1, k, n) + c_2(k, n)) \binom{\ell(k, n) C_d(\theta_1)}{2} \right). \quad (11)$$

Filtered quantum search. Suppose that Line 8 is implemented using Algorithm 1. We take the predicate f to be $\theta(v_j, \cdot) \leq \theta$ with domain $L_{f,j}$. We take the filter g to be $\texttt{popcount}_{k,n}(v_j, \cdot)$. Each call to the search routine returns at most one neighbour of v_j. To find all detectable neighbours of v_j in $L_{f,j}$ we must repeat the search several times. Known neighbours of v_j can be removed from $L_{f,j}$ to avoid a coupon collector scenario. Proposition 1 with $P = |L_{f,j}| \cdot \Pr[\hat{P}_{k,n} \mid \hat{B}_{f,\theta_1}]$, $Q = 1$, and $\gamma = 1$ gives us that the number of $\mathbf{G}(g)$ iterations in a search on a set of size $|L_{f,j}|$ is $q_0(|L_{f,j}|) = \frac{1}{2}\sqrt{|L_{f,j}|}$.

We consider an implementation of Line 8 in which searches are repeated until a search fails to find a neighbour of v_j. With $N = \ell(k, n)$, the set L_f is of expected size $\ell(k, n) \cdot C_d(\theta_1)$ and contains an expected $\ell(k, n) \cdot W_d(\theta, \theta_1, \theta_1)$

neighbouring pairs detectable by `popcount`. The set $L_{f,j}$ is expected to contain a proportional fraction of these pairs. As such, we expect to call the search subroutine $|L_{f,j}| \cdot r(\theta_1, k, n) + 1$ times in iteration j where

$$r(\theta_1, k, n) = \frac{N \cdot W_d(\theta, \theta_1, \theta_1)}{\binom{|L_f|}{2}} \approx \frac{2\, W_d(\theta, \theta_1, \theta_1)}{\ell(k, n) \cdot C_d(\theta_1)^2}.$$

The inner loop makes an expected

$$\sum_{j=1}^{|L_f|-1} \frac{1}{2}\sqrt{j}\,(jr(\theta_1, k, n) + 1)$$

applications of $\mathbf{G}(g)$. This admits an asymptotic expansion similar to that of Equation 9. If we assume that $|L_f|$ takes its expected value of $\ell(k, n)C_d(\theta_1)$, then the inner loop makes

$$q_3(\theta_1, k, n) \cdot (\ell(k, n) \cdot C_d(\theta_1))^{3/2}$$

applications of $\mathbf{G}(g)$, where

$$q_3(\theta_1, k, n) = \frac{2\, W_d(\theta, \theta_1, \theta_1)}{5\, C_d(\theta_1)} + \frac{1}{3}.$$

Proposition 1 also provides an estimate for the rate at which reflections about the true positives, $\mathbf{R}_{f \cap g}$ are performed. With P and Q as above, we find that $\mathbf{R}_{f \cap g}$ is applied at roughly $p(\theta_1, k, n) = \sqrt{\Pr[\hat{P}_{k,n} \mid \hat{B}_{f,\theta_1}]}$ the rate of $\mathbf{G}(g)$ iterations. The total cost of searching for neighbouring pairs in L_f is therefore

$$s(\theta_1, k, n) = (q_1 \cdot p(\theta_1, k, n) + q_2(k, n)) \cdot q_3(\theta_1, k, n) \cdot \left(\ell(k, n) \cdot C_d(\theta_1)\right)^{3/2}. \quad (12)$$

Populating L_f has a cost of $c_1 \cdot \ell(k, n)$. With the list-size preserving t given above, the optimal parameters θ_1, k, and n can be obtained by minimising the total cost

$$\frac{c_1 \cdot \ell(k, n) + s(\theta_1, k, n)}{W_d(\theta, \theta_1, \theta_1)}. \quad (13)$$

5.3 ListDecodingSearch

The optimal choice of θ_1 in RandomBucketSearch balances the cost of $N \cdot t$ cap membership tests against the cost of all calls to the search subroutine. It can be seen that reducing the cost of populating the buckets would allow us to choose a smaller θ_1, which would reduce the cost of searching within each bucket.

Algorithm 4, ListDecodingSearch, is due to Becker, Ducas, Gama, and Laarhoven [8]. Its complexity is $2^{(0.292\ldots+o(1))d}$ classically and $2^{(0.265\ldots+o(1))d}$ quantumly [36, 35]. Like RandomBucketSearch, it computes a large number of list-cap intersections. However, these list-cap intersections involve a structured list—the list-cap intersections in RandomBucketSearch involve the inherently unstructured input list.

Algorithm 4 ListDecodingSearch

Input: A list $L = (v_1, v_2, \ldots v_N) \subset \mathcal{S}^{d-1}$ of N. Parameters $\theta, \theta_1, \theta_2 \in (0, \pi/2)$ and $t \in \mathbb{Z}_+$.

Output: A list of pairs $(u, v) \in L \times L$ with $\theta(u, v) \leq \theta$.

1: **function** ListDecodingSearch($L; \theta, \theta_1, \theta_2, t$)
2: Sample a random product code F of size t
3: Initialise an empty list L_f for each $f \in F$
4: **for** $1 \leq i < N$ **do**
5: $F_i \leftarrow F \cap \mathcal{C}^{d-1}(v_i, \theta_2)$
6: Add v_i to L_f for each f in F_i
7: **for** $1 \leq j < N$ **do**
8: $F_j \leftarrow F \cap \mathcal{C}^{d-1}(v_j, \theta_1)$
9: **for** $f \in F_j$ **do**
10: $L_{f,j} \leftarrow \{v_k \in L_f : j < k \leq N\}$
11: $L_{F,j} \leftarrow \coprod_{f \in F_j} L_{f,j}$ (disjoint union)
12: Search $L_{F,j}$ for any number of u that satisfy $\theta(v_j, u) \leq \theta$
13: For each such u found, add (v_j, u) to L'.
14: If $|L'| \geq N$, **return** L'.
15: **return** L'

Description of Algorithm 4. The algorithm first samples a t point *random product code* F. See [8] for background on random product codes. In our analysis, we treat F as a list of uniformly random points on $\mathcal{S}^{d-1}$. Some justification for this heuristic is given in [8, Appendix B].

The first loop populates t buckets that are defined in terms of the points of F. Bucket L_f stores elements of L that lie in the cap of angle θ_2 about f. Each bucket is of expected size $N \cdot C_d(\theta_2)$.

The second loop iterates over $v_j \in L$ and searches for neighbours of v_j in the disjoint union of the buckets that are defined by centres that are within angle θ_1 of v_j. The set F_j constructed on Line 8 contains an expected $t \cdot C_d(\theta_1)$ bucket centres. The disjoint union of elements from the corresponding buckets, denoted $L_{F,j}$, is of expected size $N \cdot C_d(\theta_2) \cdot t \cdot C_d(\theta_1)$.

Suppose that w is a neighbour of v_j, so $\theta(v_j, w) \leq \theta$. The measure of the wedge formed by a cap of angle θ_1 about v_j and a cap of angle θ_2 about w is at least $W_d(\theta, \theta_1, \theta_2)$. Assuming that the points of a random product code are indistinguishable from points sampled uniformly on the sphere, the probability that some $f \in F_j$ contains w is at least $t \cdot W_d(\theta, \theta_1, \theta_2)$.

The second loop is executed N times. Iteration j searches $L_{F,j}$ for neighbours of v_j. With $N = \ell(k, n)$ there are expected to be N detectable neighbouring pairs in L. With $t = 1/W_d(\theta, \theta_1, \theta_2)$ we expect that each neighbouring pair is of the form (v_j, w) with $w \in L_{F,j}$.

We have omitted the list decoding mechanism by which list-cap intersections are computed. In our analysis we assume that the cost of a list-cap intersection such as $F_i = F \cap \mathcal{C}^{d-1}(v_i, \theta_1)$ is proportional to $|F_i|$, but independent of $|F|$.

Specifically, we assume that the cost is equal to $|F_i|$ inner product tests. A cost of $|F_i|^{1+o(1)}$ is justified by [8, Section 5]. A cursory examination of that method shows the $o(1)$ term masks a cost much larger than one inner product test, but there may yet be improvements in this area.

Filtered search. Suppose that the implementation of Line 12 of Algorithm 4 applies $\texttt{popcount}_{k,n}(v_j, \cdot)$ to each element of $L_{F,j}$ and then applies an inner product test to each vector that passes. This implementation applies popcount tests to all $N \cdot C_d(\theta_2) \cdot t \cdot C_d(\theta_1)$ elements $L_{F,j}$ and finds all of the neighbours of v_j that pass. Note that $w \in L_{F,j}$ implies that there exists some $f \in F$ such that both v_j and w lie in a cap of angle θ_1 around f. Inner product tests are applied to a $p(\theta_1, k, n) \geq \Pr[\hat{P}_{k,n} \mid \hat{B}_{f,\theta_1}]$ fraction of all pairs.[3]

The cost of preparing all t buckets in the first loop is $c_1 \cdot N \cdot t \cdot C_d(\theta_2)$. The cost of constructing the search spaces in the second loop is $c_1 \cdot N \cdot t \cdot C_d(\theta_1)$. Each search has a cost of $|L_{F,j}|$ popcount tests and $|L_{F,j}| \cdot p(\theta_1, k, n)$ inner product tests. With the list-size preserving parameterisation given above, the optimal θ_1, θ_2, k, and n can be obtained by minimising the total cost

$$\frac{\ell(k,n)}{W_d(\theta,\theta_1,\theta_2)}\Big(c_1 \cdot C_d(\theta_1) + c_1 \cdot C_d(\theta_2) + \big(c_1 \cdot p(\theta_1,k,n) + c_2(k,n)\big) \cdot \ell(k,n) \cdot C_d(\theta_1) \cdot C_d(\theta_2)\Big). \quad (14)$$

Filtered quantum search. Suppose that Line 12 is implemented using Algorithm 1. We take the predicate f to be $\theta(v_j, \cdot) \leq \theta$ with domain $L_{F,j}$. We take the filter g to be $\texttt{popcount}_{k,n}(v_j, \cdot)$. Each call to the search routine returns at most one neighbour of v_j. Known neighbours of v_j can be removed from $L_{F,j}$ to avoid a coupon collector scenario. Proposition 1 with $P = |L_{F,j}| \cdot \Pr[\hat{P}_{k,n} \mid \hat{B}_{f,\theta_2}]$, $Q = 1$, and $\gamma = 1$ gives us that the number of $\mathbf{G}(g)$ iterations in a search on a set of size $|L_{f,j}|$ is $q_0(|L_{F,j}|) \approx \frac{1}{2}\sqrt{|L_{F,j}|}$.

Assuming that computing $F_j = F \cap C(v_j, \theta_1)$ has a cost of $c_1 |F_j|$, the N iterations of Lines 5 and 8 have a total cost of

$$c_1 \cdot N \cdot t \cdot (C_d(\theta_1) + C_d(\theta_2)) \quad (15)$$

Each search applies an expected

$$q_0(|L_{F,j}|) \approx \frac{1}{2}\sqrt{N \cdot C_d(\theta_1) \cdot t \cdot C_d(\theta_2)}$$

applications of $\mathbf{G}(g)$. Reflections about the true positives, $\mathbf{R}_{f \cap g}$, are performed at roughly $p(\theta_1, k, n) = \sqrt{\Pr[\hat{P}_{k,n} \mid B_{f,\theta_1}]}$ the rate of $\mathbf{G}(g)$ iterations. We consider an implementation of Line 8 in which searches are repeated until a search fails to find a neighbour of v_j. With the list-size preserving parameters given above,

[3] The inequality is because v_j and w may be contained in multiple buckets, $L_{f,j}$.

we expect to perform two filtered quantum searches per iteration of the second loop. The optimal parameters can be obtained by minimising the total cost

$$\ell(k,n)\left(c_1\frac{C_d(\theta_1)+C_d(\theta_2)}{W_d(\theta,\theta_1,\theta_2)}+(q_1p(\theta_1,k,n)+q_2(k,n))\sqrt{\frac{\ell(k,n)C_d(\theta_1)C_d(\theta_2)}{W_d(\theta,\theta_1,\theta_2)}}\right).$$

6 Cost estimates

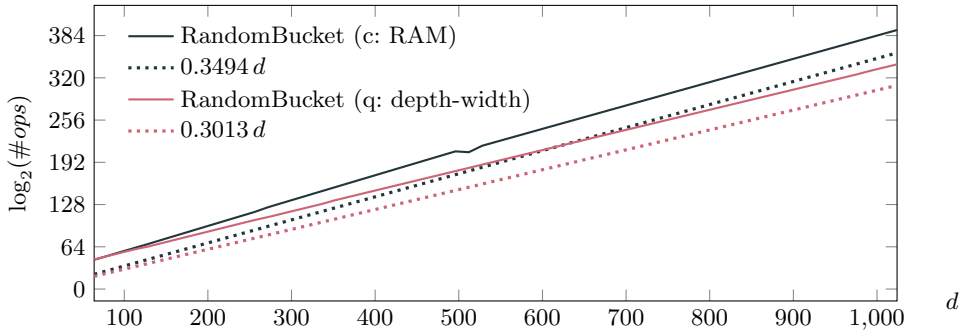
Optimising the cost functions in Sections 5.1, 5.2 and 5.3 we can now give estimates for the cost of running the three nearest neighbour search algorithms.

We consider several classical and quantum cost metrics to concretise the subroutine costs from Section 5, and we provide raw data for each metric in Figure 2 and Appendix E. The classical cost metrics that we consider are: *c (unit cost)*, which assigns unit cost to `popcount`; *c (RAM)*, which uses the classical circuits of Section 3. The quantum cost metrics that we consider are: *q (unit cost)*, which assigns unit cost to $\mathbf{G}(g)$ and ignores $\mathbf{R}_{f\cap g}$; *q (depth-width)*, which assigns unit cost to every gate (including the identity) in the quantum circuits of Section 3; *q (gates)*, which assigns unit cost only to the non-identity gates; *q (T count)*, which assigns unit cost only to T gates; and *q (GE19)*, which is described in Section 6.1.

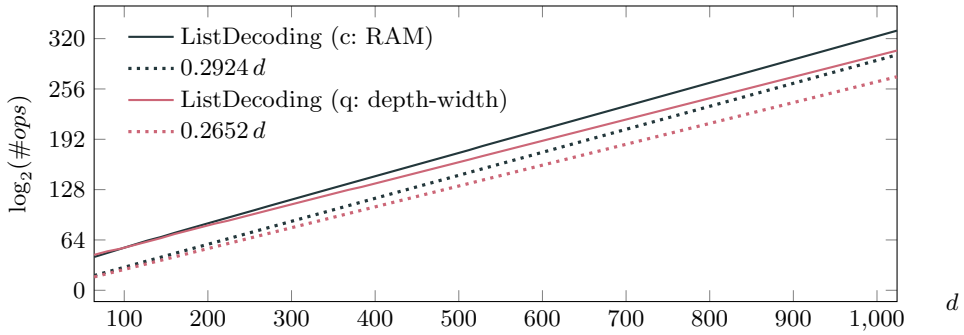
We stress that our figures do not give full costs of running the respective algorithms but are underestimates (in both the classical and the quantum case). In particular, we do not cost the resources needed to realise a full inner product *inside* the search, i.e. effectively set $c_1 = q_1 = 0$. Instead, we *assume* that a full inner product needs 32 bits of precision and costs approximately $(32^2 \cdot d)/n$ times as much as a `popcount` regardless of metric (cf. Section 3.2, Section 3.3, and Appendix C). We then insist that `popcount` dominates the cost of our search by enforcing that we are performing at least $(32^2 \cdot d)/n$ calls to `popcount` per inner product call classically and $\left((32^2 \cdot d)/n\right)^2$ quantumly. Therefore in, e.g. Section 5.3, $c_1p(\theta_1,k,n) < c_2(k,n)$ and $q_1p(\theta_1,k,n) < q_2(k,n)$. In other words, given our assumption, we satisfy Remark 1 and ignore costs relating to $\mathbf{R}_{f\cap g}$ in Section 5. When this assumption is correct, foregoing costing the full inner products does not alter the relative performance of classical and quantum cases. We recall that we also do not cost the setup of computing $u \mapsto h(u)$, which is always formed of n classical inner products. However, we do assign a classical cost to filling buckets, i.e. $c_1 \neq 0$. Following the discussion in Section 2.5 we set $k = \lfloor n/3 \rfloor$.

Our results are presented in Figure 2. We also plot the leading term of the asymptotic complexity of the respective algorithms as these are routinely referred to in the literature as lower bound estimates for the expected cost. We give the source code to produce our figures in Appendix G. The raw data used to produce our figures as well as raw data for all considered cost metrics is available as an attachment to the electronic version of this document.

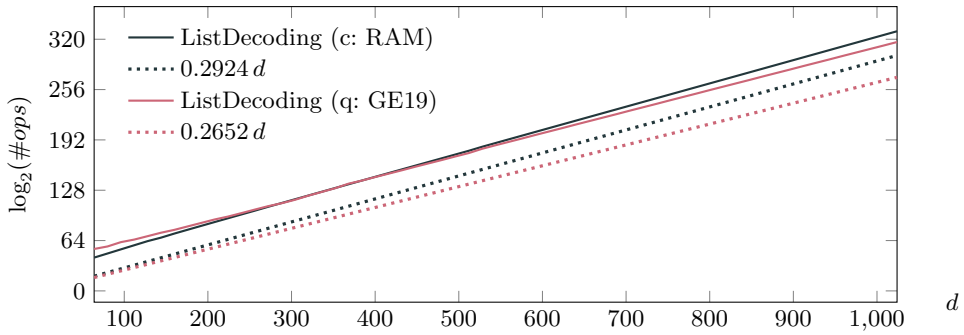
Fig. 2: Quantum ("q") and classical ("c") resource estimates for NNS search.



RandomBucketSearch. Raw data: c (unit cost), c (RAM), q (unit cost), q (depth-width), q (gates), q (T count). q (GE19).



ListDecodingSearch. Raw data: c (unit cost), c (RAM), q (unit cost), q (depth-width), q (gates), q (T count). q (GE19).



6.1 Barriers to a quantum advantage

Our results in Figure 2 indicate the quantum variant of each algorithm exhibits a (typically) small advantage compared to its classical counterpart. In this section, we list various obstacles to be overcome to realise any such advantage. We have highlighted the most important barriers here. We discuss some other barriers—the poor parallelisability of quantum search and the real cost of quantum hardware—in Appendix F.

Dependence on qRAM. Quantum accessible classical memories are used in many quantum algorithms. For example, they are used when Grover's algorithm is applied to a database search [29], in collision finding [13], and in some algorithms for the the dihedral hidden subgroup problem [34]. The use of qRAM is not without some controversy [28, 10], and previous work on quantum lattice sieve algorithms [36, 35] have noted that constructing practical qRAM seems challenging. Morally, looking up an ℓ bit value in a table with 2^n entries should have a cost that grows at least with $n + \ell$. Recent results [6, 7, 41] indicate that realistic implementations of qRAM have costs that grow much more quickly than this. When ancillary qubits are kept to a minimum, the best known Clifford+T implementation of a qRAM has a $\mathbf{T}$ count of $4 \cdot (2^n - 1)$ [7]. While it is conceivable that a qRAM could be constructed at lower cost on a different architecture, as has been suggested in [23], our unit cost qRAM gates should be seen as a powerful, and potentially unrealistic, resource.

One can argue that classical RAMs also have a large cost. Indeed, the real cost of classical sieving algorithms may be much larger than their RAM model cost. This is not to say that classical and quantum RAMs have the same cost. In particular, some savings can be had in the classical case by careful use of localised access to non-volatile memory. A classical program may be able to keep large amounts of its memory "off-line" at any given time, but a quantum program must have "on-line" access to every memory cell that may lie in the support of a superposition query. In the case of the uniform superposition, which we have used above, this is the entire memory.

Cost underestimates. We have ignored the possibility that quantum search fails to find a marked element when one exists. Proposition 1 only tells us that search succeeds with probability not less than $1/8$. While it may be possible to improve the probability of success without increasing the cost of search, it is likely that some cost increase is necessary for list-size preserving parameterisations. We have also set $\gamma = 1$ in each of our applications of Proposition 1.

Finally, we have assumed that the $\mathbf{D}$ subroutine in Section 3.1 is simply parallel $\mathbf{H}$ gates. We suspect that this is close to the cost of constructing the uniform superposition over $\{k : v_k \in L_i\}$ in Line 5 of Algorithm 2 and over $\{k : v_k \in L_{f,j}\}$ in Line 8 of Algorithm 3. A precise cost estimate could be derived using standard techniques, e.g. [27]. Constructing the uniform superposition over $\{k : v_k \in L_{F,j}\}$ in Line 12 of Algorithm 4 appears to be more complicated. While

we do not expect the cost to be large, it could easily exceed the cost of `popcount`. We leave this to future work.

Error correction overhead. The depth-width metric for quantum circuits assumes that dispatching a logical gate to a logical qubit costs one RAM operation. In practice, however, the cost is likely to scale with the number of physical qubits that are used to encode each logical qubit. This may be significant. In a circuit of depth D and width W, the surface code maps each logical qubit into a two dimensional array of physical qubits with edge length, or *distance*, $\Theta(\log(DW))$. To perform a single logical operation, classical control hardware will dispatch several instructions to each of the $\Theta(\log^2(DW))$ physical qubits in this array. The classical control hardware also performs a non-trivial error tracking routine between logical operations, which takes measurement results from half of the physical qubits as input.[4]

Gidney and Ekerå have estimated the resources required to factor a 2048 bit RSA modulus using Shor's algorithm [22]. Under a plausible assumption on the physical qubit error rate, they calculate that a factoring circuit with $2^{12.6}$ logical qubits and depth 2^{31} requires a distance $\delta = 27$ surface code. Each logical qubit is encoded in $2\delta^2 = 1458$ physical qubits, and the error tracking routine applies at least $\delta^2 = 729$ bit operations, per logical qubit per layer of logical circuit depth, to read its input.

We have adapted scripts provided by Gidney and Ekerå to estimate the surface code distance needed by our quantum search circuits. The last plot of Figure 2 shows the cost of ListDecodingSearch when every logical gate (including the identity) is assigned a cost of δ^2. The advantage over classical search in dimension 512 is only $2^{2.9}$ operations. It is $2^{7.8}$ in dimension 768 and $2^{13.7}$ in dimension 1024.

One should note that error correction for the surface code sets a natural clock speed, which Gidney and Ekerå estimate at one cycle per microsecond. The cost metric that we have used in Figure 2 ignores the cost of processing the measurement results. Hence, it assigns unit cost to what may be a full microsecond of classical computation on hardware that is dedicated to the control of just a few logical qubits. Even the $2^{13.7}$ factor improvement in dimension 1024 could evaporate when all of the classical control is accounted for. Moreover, quantum computations operating at a megahertz clock rate are thousands of times slower than classical computations with similar logical operation counts. Gidney and Ekerå estimate that their factoring circuit, the cost of which is dominated by a single modular exponentiation, would take 7.44 hours to run.

On the positive side, the cost estimate used in Figure 2 is specific to the surface code architecture. Significant improvements may be possible. Gottesman has shown that an overhead of $\Theta(1)$ physical qubits per logical qubit is theoretically possible [24]. Whether this technique offers lower overhead than the surface code in practice is yet to be seen.

[4] For a thorough introduction to how logical operations are performed on the surface code see [20], and for more advanced techniques see e.g. [31].

6.2 Relevance to SVP

The NNS algorithms that we have analysed are closely related to lattice sieves for SVP. While the asymptotic cost of the best known lattice sieve is often used as a proxy for the asymptotic cost of solving SVP, we caution the reader against making this comparison in a non-asymptotic setting. On the one hand, our estimates might lead one to underestimate the cost of solving SVP:

- the costs given in Figure 2 represent one iteration of NNS within a sieve, while sieve algorithms make $\mathsf{poly}(d)$ iterations;
- the costs given in Figure 2 do not account for all of the subroutines within each NNS algorithm.

On the other hand, our estimates might lead one to overestimate the cost of solving SVP:

- the "dimensions for free" technique of [18] can be used to solve SVP in dimension d by calling a sieving routine in dimension $d' < d$;
- there are heuristics, that exploit the vector space structure present in sieving, not captured in the general NNS setting. For example, reductions can be made with either $+u$ or $-u$, and the database of vectors can be kept sorted by length (as short vectors are more likely to be useful than long vectors).

Regarding this final point, we compare the database size recommended by our methodology to the database size used in the largest sieving experiments to date [2]. Under the *c (RAM)* metric and with `popcount` parameters matching [2], our analysis of Algorithm 3 recommends a database of size $N(d) \approx 2/C_d(\pi/3)$. The experiments reported in [2] run Algorithm 3 with a database of size $N'(d) = 3.2 \cdot 2^{0.2075d}$ up to dimension 127. One should note that $N'(127)$ is smaller than $N(127)$ by a factor of approximately eight. A factor of two can be explained by the fact that [2] treats each database entry v as $\pm v$. It is possible that the remaining factor of four can be explained by the other heuristics that are used in [2] and that $N(d)/8$ is a sufficiently large database in higher dimensions. On the other hand, it could be that a database of size $N'(d)$ continues to be sufficient. It seems unlikely that implementation tricks can explain the factor of 22 gap between $N'(1024)$ and $N(1024)$. We leave as future work the task of more accurately determining the database size that is sufficient for solving SVP.

References

1. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: 33rd ACM STOC. pp. 601–610. ACM Press (Jul 2001)
2. Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E.W., Stevens, M.: The general sieve kernel and new records in lattice reduction. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 717–746. Springer, Heidelberg (May 2019)
3. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 327–343. USENIX Association (Aug 2016)

4. Amy, M., Maslov, D., Mosca, M., Roetteler, M.: A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32(6), 818–830 (June 2013)
5. Amy, M., Matteo, O.D., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J.M.: Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. In: Avanzi, R., Heys, H.M. (eds.) SAC 2016. LNCS, vol. 10532, pp. 317–337. Springer, Heidelberg (Aug 2016)
6. Arunachalam, S., Gheorghiu, V., Jochym-O'Connor, T., Mosca, M., Srinivasan, P.V.: On the robustness of bucket brigade quantum ram. New Journal of Physics 17(12), 123010 (2015), http://stacks.iop.org/1367-2630/17/i=12/a=123010
7. Babbush, R., Gidney, C., Berry, D.W., Wiebe, N., McClean, J., Paler, A., Fowler, A., Neven, H.: Encoding electronic spectra in quantum circuits with linear T complexity. Phys. Rev. X 8, 041015 (Oct 2018), https://link.aps.org/doi/10.1103/PhysRevX.8.041015
8. Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: Krauthgamer, R. (ed.) 27th SODA. pp. 10–24. ACM-SIAM (Jan 2016)
9. Becker, A., Gama, N., Joux, A.: Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522 (2015), http://eprint.iacr.org/2015/522
10. Bernstein, D.J.: Cost analysis of hash collisions: Will quantum computers make sharcs obsolete? Workshop Record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems (2009), http://cr.yp.to/papers.html#collisioncost
11. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. Fortschritte der Physik 46(4-5), 493–505 (1998), https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291521-3978%28199806%2946%3A4/5%3C493%3A%3AAID-PROP493%3E3.0.CO%3B2-P
12. Brassard, G., Hoyer, P., Mosca, M., Tapp, A.: Quantum amplitude amplification and estimation. Contemporary Mathematics 305, 53–74 (2002), https://arxiv.org/abs/quant-ph/0005055
13. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. SIGACT News 28(2), 14–19 (Jun 1997), http://doi.acm.org/10.1145/261342.261346
14. Bravyi, S., Kitaev, A.: Universal quantum computation with ideal Clifford gates and noisy ancillas. Phys. Rev. A 71, 022316 (Feb 2005), https://link.aps.org/doi/10.1103/PhysRevA.71.022316
15. Charikar, M.: Similarity estimation techniques from rounding algorithms. In: 34th ACM STOC. pp. 380–388. ACM Press (May 2002)
16. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit (2004), arXiv:quant-ph/0410184
17. Draper, T., Kutin, S., Rains, E., Svore, K.M., Svore, K.M.: A logarithmic-depth quantum carry-lookahead adder. Quantum Info. Comput. pp. 351–369 (January 2006), https://www.microsoft.com/en-us/research/publication/a-logarithmic-depth-quantum-carry-lookahead-adder/, lANL ArXiv: quant-ph/0406142.
18. Ducas, L.: Shortest vector from lattice sieving: A few dimensions for free. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 125–145. Springer, Heidelberg (Apr / May 2018)

19. Fitzpatrick, R., Bischof, C.H., Buchmann, J., Dagdelen, Ö., Göpfert, F., Mariano, A., Yang, B.Y.: Tuning GaussSieve for speed. In: Aranha, D.F., Menezes, A. (eds.) LATINCRYPT 2014. LNCS, vol. 8895, pp. 288–305. Springer, Heidelberg (Sep 2015)
20. Fowler, A.G., Mariantoni, M., Martinis, J.M., Cleland, A.N.: Surface codes: Towards practical large-scale quantum computation. Phys. Rev. A 86, 032324 (Sep 2012), https://link.aps.org/doi/10.1103/PhysRevA.86.032324
21. Gheorghiu, V., Mosca, M.: Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes (2019), arXiv:1902.02332v2
22. Gidney, C., Ekerå, M.: How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits (2019), https://arxiv.org/abs/1905.09749, arXiv:1905.09749
23. Giovannetti, V., Lloyd, S., Maccone, L.: Quantum random access memory. Phys. Rev. Lett. 100, 160501 (Apr 2008), http://link.aps.org/doi/10.1103/PhysRevLett.100.160501
24. Gottesman, D.: Fault-tolerant quantum computation with constant overhead (2013), https://arxiv.org/abs/1310.2984, arXiv:1310.2984
25. Gottesman, D., Chuang, I.L.: Quantum teleportation is a universal computational primitive (1999), arXiv:quant-ph/9908010
26. Grassl, M., Langenberg, B., Roetteler, M., Steinwandt, R.: Applying grover's algorithm to AES: Quantum resource estimates. In: Takagi, T. (ed.) Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016. pp. 29–43. Springer, Heidelberg (2016)
27. Grover, L., Rudolph, T.: Creating superpositions that correspond to efficiently integrable probability distributions (Aug 2002), arXiv:quant-ph/0208112
28. Grover, L., Rudolph, T.: How significant are the known collision and element distinctness quantum algorithms. Quantum Info. Comput. 4, 201–206 (May 2004)
29. Grover, L.K.: Quantum mechanics helps in searching for a needle in a haystack. Phys. Rev. Lett. 79, 325–328 (Jul 1997), http://link.aps.org/doi/10.1103/PhysRevLett.79.325
30. Häner, T., Roetteler, M., Svore, K.M.: Factoring using 2n + 2 qubits with toffoli based modular multiplication. Quantum Info. Comput. 17(7-8), 673–684 (Jun 2017), http://dl.acm.org/citation.cfm?id=3179553.3179560
31. Horsman, C., Fowler, A.G., Devitt, S., Meter, R.V.: Surface code quantum computing by lattice surgery. New Journal of Physics 14(12), 123011 (2012), http://stacks.iop.org/1367-2630/14/i=12/a=123011
32. Jaques, S., Schanck, J.M.: Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 32–61. Springer, Heidelberg (Aug 2019)
33. Klein, P.N.: Finding the closest lattice vector when it's unusually close. In: Shmoys, D.B. (ed.) 11th SODA. pp. 937–941. ACM-SIAM (Jan 2000)
34. Kuperberg, G.: Another subexponential-time quantum algorithm for the Dihedral Hidden Subgroup Problem. In: Theory of Quantum Computation, Communication and Cryptography – TQC 2013. pp. 20–34. LIPIcs 22 (2013), http://drops.dagstuhl.de/opus/volltexte/2013/4321
35. Laarhoven, T.: Search problems in cryptography: from fingerprinting to lattice sieving. Ph.D. thesis, Department of Mathematics and Computer Science (2 2016), proefschrift
36. Laarhoven, T., Mosca, M., van de Pol, J.: Solving the shortest vector problem in lattices faster using quantum search. In: Gaborit, P. (ed.) Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013. pp. 83–101. Springer, Heidelberg (Jun 2013)

37. Lee, Y., Kim, W.C.: Concise formulas for the surface area of the intersection of two hyperspherical caps. Tech. rep., KAIST (February 2014), https://ie.kaist.ac.kr/uploads/professor/tech_file/Concise+Formulas+for+the+Surface+Area+of+the+Intersection+of+Two+Hyperspherical+Caps.pdf
38. Li, S.: Concise formulas for the area and volume of a hyperspherical cap. Asian Journal of Mathematics and Statistics 4(1), 66–70 (2011)
39. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 319–339. Springer, Heidelberg (Feb 2011)
40. Maslov, D.: Advantages of using relative-phase toffoli gates with an application to multiple control toffoli optimization. Physical Review A 93(2), 022311 (2016)
41. Matteo, O.D., Gheorghiu, V., Mosca, M.: Fault tolerant resource estimation of quantum random-access memories (2019), arXiv:1902.01329v1
42. Micciancio, D., Regev, O.: Lattice-based cryptography. In: Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.) Post-Quantum Cryptography, pp. 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-540-88702-7_5
43. Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. of Mathematical Cryptology 2(2) (2008)
44. NIST: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Tech. rep., NIST (January 2017), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf
45. Parhami, B.: Efficient hamming weight comparators for binary vectors based on accumulative and up/down parallel counters. IEEE Trans. on Circuits and Systems 56-II(2), 167–171 (2009), https://doi.org/10.1109/TCSII.2008.2010176
46. Roetteler, M., Naehrig, M., Svore, K.M., Lauter, K.E.: Quantum resource estimates for computing elliptic curve discrete logarithms. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part II. LNCS, vol. 10625, pp. 241–270. Springer, Heidelberg (Dec 2017)
47. Takahashi, Y., Kunihiro, N.: A fast quantum circuit for addition with few qubits. Quantum Info. Comput. 8(6), 636–649 (Jul 2008), http://dl.acm.org/citation.cfm?id=2016976.2016981

A Filtered quantum search

Let f be a predicate on a domain of size N, and suppose that f has M roots. A theorem of Boyer, Brassard, Høyer, and Tapp establishes that one can find a root of f in $O(\sqrt{N/M})$ Grover iterations even when $p = M/N$ is not known. Lemma 1 expresses their core observation.

Lemma 1. *Suppose that measuring* $\mathbf{A}|0\rangle$ *would yield a root of* f *with probability* $\sin^2(\theta)$. *Fix a positive integer* m *and let* j *be chosen uniformly from* $\{0, \ldots, m-1\}$. *Then measuring* $\mathbf{G}(\mathbf{A}, f)^j\mathbf{D}|0\rangle$ *will yield a root of* f *with probability* $\frac{1}{m}\sum_{j=0}^{m-1}\sin^2((2j+1)\theta) = \frac{1}{2} - \frac{\sin(4m\theta)}{4m\sin(2\theta)}$.

The complete strategy is made precise by [11, Theorem 3]. One simply runs the algorithm repeatedly with j chosen uniformly from successively larger intervals.

Once $m > 1/\sin(2\theta) \approx \sqrt{1/p}$, each attempt succeeds with probability at least $1/4$. This generalises to amplitude amplification directly [12].

We cannot apply apply the BBHT strategy directly in Algorithm 1. The number of iterations of $\mathbf{G}(g)$ in the inner Grover search (j in Line 3) is fixed throughout the outer amplitude amplification. However, if the inner Grover search only succeeds with probability x, then we apply a factor $\sqrt{1/x}$ more iterations of amplitude amplification to compensate. Lemma 2 establishes the probability that the inner Grover search succeeds with probability at least x when j is chosen uniformly from $\{0, \ldots, m-1\}$ for any m and any choice of $x \in [0, 1)$.

Lemma 2. *Fix $\theta \in [0, \pi/2]$ and $x \in [0, 1)$. Let $p(j) = \sin^2((2j+1)\theta)$ and let $q_x(m) = \frac{1}{m}\left|\{j \in \mathbb{Z} : 0 \leq j < m, p(j) < x\}\right|$. If $m > \frac{\pi}{4\theta}$, then*

$$\frac{\arcsin(\sqrt{x})}{\pi - \arcsin(\sqrt{x})} - \delta\theta < q_x(m) < \frac{3\arcsin(\sqrt{x})}{\pi + \arcsin(\sqrt{x})} + \delta\theta$$

for some explicit constant δ.

Proof. Let I_0 be the interval $[0, \arcsin(\sqrt{x}))$, and for integers $t \geq 1$ let $I_t = (t\pi - \arcsin(\sqrt{x}), t\pi + \arcsin(\sqrt{x}))$. Let c be the largest integer for which the interval $[0, (2m-1)\theta)$ intersects I_c. Note that $mq_x(m)$ counts the number of non-negative integers $i < m$ for which $(2i+1)\theta$ lies in $I_0 \cup I_1 \cup \cdots \cup I_c$. Thus $mq_x(m) < \left|I_0 \cup \cdots \cup I_c\right|/2\theta + O(c)$ and $mq_x(m) > \left|I_0 \cup \ldots I_{c-1}\right|/2\theta + O(c)$. Moreover, $q_x(m) > q_x(m-1)$ when $(2m-1)\theta$ lies in I_c, and $q_x(m) < q_x(m-1)$ otherwise. It follows that $q_x(m) < (2c+1)\arcsin(\sqrt{x})/(c\pi + \arcsin(\sqrt{x})) + O(\theta)$ and $q_x(m) > (2c-1)\arcsin(\sqrt{x})/(c\pi - \arcsin(\sqrt{x})) + O(\theta)$. The upper bound is decreasing as a function of c and the lower bound is increasing as a function of c. Hence the claim holds when $c \geq 1$. It remains to show that it holds when $m > \frac{\pi}{4\theta}$ and $c = 0$. For this, observe that $q_x(\lfloor \frac{\pi}{4\theta} \rfloor) = 2\arcsin(\sqrt{x})/\pi + O(\theta)$ and $q_x(m)$ is decreasing until $c = 1$. □

Proof (of Proposition 1). Fix $x \in (0, 1)$. We will analyse Algorithm 1 with respect to the parameters $m_1 = \left\lceil \frac{\pi}{4}\sqrt{\gamma N/P} \right\rceil$ and $m_2 = \left\lceil \frac{\pi}{4}\sqrt{\gamma P/xQ} \right\rceil$, so the values j and k chosen in Line 2 satisfy $0 \leq j < m_1$ and $0 \leq k < m_2$. Let θ_g be such that $\sin^2(\theta_g) = |g|/N$. Let $p(j) = \sin^2((2j+1)\theta_g)$. Let $\theta_h(j)$ be such that $\sin^2(\theta_h(j)) = |g|/(p(j)\,|f \cap g|)$. Let $q_x(m)$ be defined as in Lemma 2. With probability at least $1 - q_x(m_1)$ we have $p(j) \geq x$ and, consequently, $1/\sin(2\theta_h(j)) < m_2$. In this case we can apply Lemma 1 to obtain the probability that measuring measuring $\mathbf{G}(\mathbf{A}_j, f \cap g)^k \mathbf{A}_j|0\rangle$ yields a root of $f \cap g$. It follows that Algorithm 1 succeeds with probability at least $(1 - q_x(m_1))/4$.

The algorithm evaluates $\mathbf{G}(g)$ exactly $kj + 1$ times and evaluates $\mathbf{G}(g)^{-1}$ exactly kj times. The expected total number of evaluations of $\mathbf{G}(g)$ and $\mathbf{G}(g)^{-1}$ is approximately $c_1(x)\gamma\sqrt{N/Q}$ where $c_1(x) = (\pi/4)^2/(2\sqrt{x})$. Likewise the algorithm evaluates $\mathbf{R}_h$ exactly k times. This is approximately $c_2(x)\gamma\sqrt{P/Q}$ in expectation where $c_2(x) = \pi/(4\sqrt{x})$. Taking $x = 0.345$, and applying the upper bound on $q_x(m_1)$ from Lemma 2, we have $(1 - q_x(m_1))/4 \geq 1/8$ and $c_1(x) \approx 1/2$ and $c_2(x) \approx 4/3$. □

B Caps and wedges

The expression of $C_d(\theta)$ in terms of the regularised incomplete beta function, $C_d(\theta) = \frac{1}{2} I_{\sin^2(\theta)}\left(\frac{d-1}{2}, \frac{1}{2}\right)$, is perhaps not as well known as it should be.[5] Apart from being concise, this representation makes it clear that $C_d(\theta)$ has a hypergeometric representation (see `https://dlmf.nist.gov/8.17#E7`). Computer algebra systems often provide routines for computing hypergeometric functions that are more robust than generic numerical integration routines.

An exact expression for $W_d(\theta, \theta_u, \theta_v)$ involves a subtle case analysis [37]. Fortunately, we only need the following case.

Fact 1 ([37, Case 8]) *Let $\theta, \theta_u, \theta_v$ be real numbers is in $(0, \pi/2)$ with $\theta < \theta_u + \theta_v$ and $(\cos(\theta_v) - \cos(\theta_u)\cos(\theta))(\cos(\theta_v)\cos(\theta) - \cos(\theta_u)) < 0$. Define $\theta^* \in (0, \pi/2)$ by $\cos(\theta_v)\sec(\theta^*) = \cos(\theta_u)\sec(\theta - \theta^*)$. For any $u, v \in \mathcal{S}^{d-1}$ with $\langle u, v\rangle = \cos(\theta)$, the wedge $\mathcal{W}^{d-1}(u, \theta_u, v, \theta_v)$ has μ^{d-1} measure*

$$W_d(\theta, \theta_u, \theta_v) = J_d\left(\theta^*, \theta_v\right) + J_d\left(\theta - \theta^*, \theta_u\right)$$

where

$$J_d(t_1, t_2) = \frac{1}{\sqrt{\pi}} \frac{\Gamma\left(\frac{d}{2}\right)}{\Gamma\left(\frac{d-1}{2}\right)} \int_{t_1}^{t_2} \sin^{d-2}(\varphi) C_{d-1}\left(\arccos\left(\frac{\tan(t_1)}{\tan(\varphi)}\right)\right) \mathrm{d}\varphi.$$

While we use fixed dimension in this work, asymptotics for $C_d(\theta)$ and $W_d(\theta, \theta_u, \theta_v)$, as a function of d, are occasionally useful. For fixed θ it follows immediately from the integral representation of $C_d(\theta)$ above that $C_d(\theta) = (\sin\theta)^{(1+o(1))d}$. The expression for W_d is slightly more complicated. For fixed θ, θ_u, and θ_v let θ^* be such that $\cos\theta_u \sec\theta^* = \cos\theta_v \sec(\theta - \theta^*)$, and let $\varphi = \arccos(\cos\theta_u \sec\theta^*)$. Note that $\theta^* = \theta/2$ when $\theta_u = \theta_v$. The argument in [8, Appendix A] establishes that $W_d(\theta, \theta_u, \theta_v) = (\sin\varphi)^{(1+o(1))d}$.

Remark 2. The constraint $(\cos(\theta_v) - \cos(\theta_u)\cos(\theta))(\cos(\theta_v)\cos(\theta) - \cos(\theta_u)) < 0$ in the statement of Fact 1 ensures that u and v lie on opposite sides of the hyperplane orthogonal to $w = u/\cos(\theta_u) - v/\cos(\theta_v)$. The quantity $\cos(\theta_v)\sec(\theta^*)$ that appears in the statement of Fact 1 is equal to the quantity γ that appears in the statement of [8, Lemma 2.2]. It is the norm of the shortest $x \in \mathbb{R}^n, \|x\| \leq 1$ with $\langle x, u\rangle = \cos(\theta_u)$, $\langle x, v\rangle = \cos(\theta_v)$, and $\langle x, w\rangle = 0$. The entire wedge is contained in a cap of angle $\arccos(\|x\|)(= \arccos(\gamma))$ about this shortest x. The proof of [8, Lemma 2.2] establishes that the wedge is essentially a factor of $\sqrt{d}$ smaller than this cap asymptotically, i.e. $W_d(\theta, \theta_u, \theta_v) = \Theta\left(\frac{1}{d^2}(1-\gamma^2)^{d/2}\right)$.

[5] We learnt this from [38] via [37]. The identity is easy to check

$$\begin{aligned}\frac{1}{2} I_{\sin^2(\theta)}\left(\tfrac{d-1}{2}, \tfrac{1}{2}\right) &= \frac{1}{2\sqrt{\pi}} \frac{\Gamma(\frac{d}{2})}{\Gamma(\frac{d-1}{2})} \int_0^{\sin^2(\theta)} t^{\frac{d-1}{2}-1}(1-t)^{-\frac{1}{2}}\, \mathrm{d}t \\ &= \frac{1}{\sqrt{\pi}} \frac{\Gamma(\frac{d}{2})}{\Gamma(\frac{d-1}{2})} \int_0^{\theta} \sin^{d-2}(t)\, \mathrm{d}t.\end{aligned}$$

C Toffoli counts and circuit width

Let the notation be as in Section 3.1. The Toffoli gate is often used in circuit design. Here we count the Toffoli (or $\mathbf{T}$) gates required to implement $\mathbf{G}(g)$ when $g(i) = \texttt{popcount}_{k,n}(u, v_i)$. Recall $\mathbf{G}(g) = \mathbf{D}\mathbf{R}_0\mathbf{D}^{-1}\mathbf{R}_g$ and that $\mathbf{R}_0$ is implemented with a "multiply controlled" Toffoli. For some $m \geq 3$ the m bit Toffoli gate is $TOF^m \colon \{0,1\}^m \to \{0,1\}^m, (x_1, \ldots, x_{m-1}, x_m) \mapsto (x_1, \ldots, x_{m-1}, (x_1 \wedge \cdots \wedge x_{m-1}) \oplus x_m)$. The $m = 3$ case is referred to as a Toffoli gate, rather than, as for $m \geq 4$, a "multiply controlled" Toffoli gate. The Toffoli count is a pertinent quantity because it determines the $\mathbf{T}$ count. The $\mathbf{T}$ count plays a large part in determining the distance required for error correction, and hence the cost of error correction, see Figure 2. In the $m \geq 4$ case, i.e. for $\mathbf{R}_0$, more efficient decompositions into $\mathbf{T}$ gates are possible [40] and therefore our software does not count Toffoli gates, but rather directly counts $\mathbf{T}$ gates.

The Toffoli Count of $\mathbf{R}_g$. The Toffoli count of the full circuit described in Fig 1, i.e. including the uncomputation, is

$$c(\text{CARRY}) + 2\sum_{i=0}^{\ell-2} 2^i c(\text{ADD}_{\ell-i-1}), \tag{16}$$

where $c(\cdot)$ denotes the Toffoli count of its argument. The factor of 2 accounts for uncomputation and, as explained in Section 3.1, the CARRY circuit is only cost once. The i bit full adder ADD_i of [16] has a Toffoli count of $2i - 1$. The CARRY circuit of [30] has a Toffoli count of $4(\ell - 2) + 2$. A routine calculation gives the overall Toffoli count as $3n - 5$. When decomposing Toffoli gates ($m = 3$) into $\mathbf{T}$ gates we make use of [4]. Roughly, a Toffoli gate costs 7 $\mathbf{T}$ gates.

The Toffoli Count of $\mathbf{D}\mathbf{R}_0\mathbf{D}^{-1}$. As explained in Section 3.1, $\mathbf{D}$ and $\mathbf{D}^{-1}$ comprise solely of $\mathbf{H}$ gates. We decompose the multiply controlled Toffoli into $\mathbf{T}$ gates via [40, Table I]. In particular we use the bottommost "Ours" row for TOF^4 and $TOF^m, m \geq 5$.

The circuit width of $\mathbf{R}_g$. The first layer of adders in Figure. 1 requires $3 \times 2^{\ell-2}$ qubits. The subsequent layers of adders reuse the qubits from their "parent" layers, e.g. the adders ADD_2 in the second layer reuse the qubits from the layer consisting of ADD_1 adders, with the exception of the input carry for each adder. Hence the total number of qubits required by the adders is simply equal to the number of input lines (that end with a black diamond), i.e. $2^\ell - 1$.

The CARRY in Fig. 1 requires 2ℓ input qubits and one dirty ancilla, ℓ of which are reused from the last adder.

Therefore, the total number of qubits required by $\mathbf{R}_g$ is

$$\text{width} = (2^\ell - 1) + \ell + 1 = 2^\ell + \ell. \tag{17}$$

D The choice of adder

Let the notation be as in Section 3.1. The i bit adders of [17, 47] function similarly to [16] but instead require $O(i)$ and $O(i/\log i)$ ancillae, respectively. While these adders require more ancillae, they have an optimal depth of $O(\log i)$ compared to the depth $O(i)$ of the [16] adder. We argue below to not consider [17, 47] when costing our circuit.

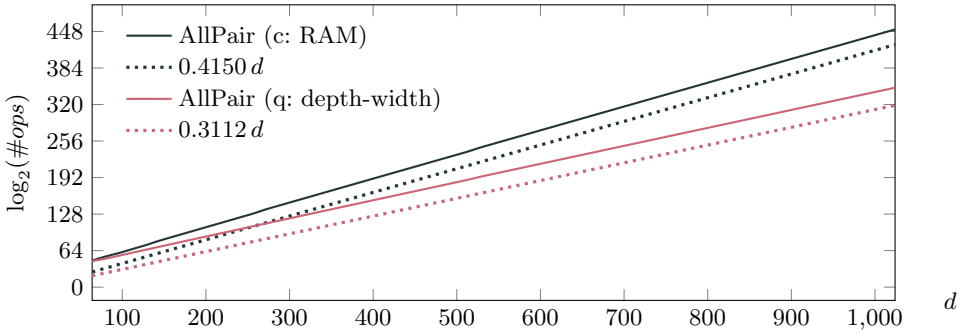
The largest n we consider is 32767 and therefore $\ell = 15$ and the largest adder we require has $i = 14$ bits. Such an in-place adder, with incoming carry bit, of [16, Table. 1] has depth 34, width 29 and requires 27 Toffoli gates. Such an in-place adder, with incoming carry bit, of [17, Table. 2] has depth 18, width 50 and requires 102 Toffoli gates. The adder of [47] does not allow an incoming carry, and hence Figure 1 would require $2^{\ell-1} = n + 1$, i.e. $\ell = 16$ and the largest adder to have $i = 15$ bits. Following the discussion at the end of [47, Section. 3.4], while ultimately the depth-width is $O(i)$ compared to $O(i \log i)$ of [17], for our small i these circuits are larger—ignoring the asymptotic subtracted terms and requiring $i = 15$, we get depth approximately 90, width approximately 45 and require approximately 435 Toffoli gates.

Therefore, while we have not explicitly constructed the relevant circuits, we expect any gain due to the smaller depths of [17, 47] would be minimal at best for our parameters, given the extra ancillae. Certainly, if one were to consider error correction, such a gain would be outweighed entirely by their higher Toffoli count.

E Additional figures

Figure 3 provides estimates for AllPairSearch from Section 5.1.

Fig. 3: Quantum ("q") and classical ("c") resource estimates for NNS search.



AllPairSearch. Raw data: c (unit cost), c (RAM), q (unit cost), q (depth-width), q (gates), q (T count). q (GE19).

F Further barriers

Poor parallelisation. Depth constraints have been suggested by NIST for evaluating the efficacy of quantum attacks [44]. Separate searches can be parallelised, but individual quantum searches cannot be parallelised without increasing their cost. We estimate that each quantum search in ListDecodingSearch in dimension $d = 432$ has depth $2^{39.1}$. Moreover, the advantage of quantum ListDecodingSearch over classical—the ratio of the depth-width cost to the RAM cost—is $2^{10.1}$. If quantum computations are constrained to depth 2^{40}, then this would be the maximum attainable advantage. Likewise, with a depth constraint of 2^{64}, the quantum advantage would attain a maximum of $2^{19.6}$ in dimension 800.

The real cost of quantum hardware. Even if all of the above barriers are avoided, there may still be a large disparity between the real cost (in, say, dollars) of classical and quantum attacks. Consider Figure 2, ListDecodingSearch (depth-width metric), in dimension $d = 352$. We estimate a cost of $2^{130.8}$ RAM operations for our classical instantiation and $2^{123.4}$ RAM operations for our quantum instantiation. The quantum variant is, nominally, $2^{7.4}$ times less expensive. But this comparison ignores the cost of hardware.

In this example, we estimate that each quantum search has depth-width cost approximately $2^{43.5}$ RAM operations from depth approximately $2^{34.0}$ and maximum width approximately $2^{11.6}$. In contrast, we estimate that each classical search costs approximately $2^{49.2}$ RAM operations and has depth approximately $2^{41.0}$. Suppose that we are prepared to wait for 2^{96} steps on however many parallel computers are required. The loop starting on Line 7 of Algorithm 4 is easily parallelised. The minimal amount of quantum hardware necessary to run the algorithm in less than 2^{96} sequential steps is $N \cdot 2^{34.0}/2^{96} = 2^{17.3}$ processors with a total footprint of $2^{28.9}$ qubits. The minimal classical hardware is $N \cdot 2^{41.0}/2^{96} = 2^{24.3}$ processors. From this perspective it is less obvious that the quantum search is better. The machines are used for similar amounts of time. Are $2^{28.9}$ qubits less expensive than $2^{24.3}$ small classical processors by a factor of $2^{7.4}$?

G Source code

G.1 config.py

config.py collects constants that may reflect a choice or a potentially moving state of the art.

```
# -*- coding: utf-8 -*-


class MagicConstants:

    """
    k/n ≈ 1/3.
    """

    k_div_n = 1/3.
```

```
    """
    We assume one Toffoli gate takes 7 T-gates.
    """

    t_div_toffoli = 7
    t_depth_div_toffoli = 3
    gates_div_toffoli = 17  # 7 T, 7 CNOT, 2 H, 1 S

    AMMR12_tof_t_count = 7
    AMMR12_tof_t_depth = 3
    AMMR12_tof_gates   = 17
    AMMR12_tof_depth   = 10

    """
    Assuming 32 bits are used to represent full vectors, we expect the ratio
    between a full inner product and a popcount call to be 32^2 * d / n, where
    32^2 is the cost of a naive multiplier and n approximates the cost of a
    hamming weight call.
    """

    word_size = 32
```

G.2 probabilities.py

probabilities.py provides utility functions for computing the probabilities from Sections 2.3 and 4.

```
# -*- coding: utf-8 -*-
"""
Estimating relevant probabilities on the sphere and for popcount.

To run doctests, run: ``PYTHONPATH=`pwd` sage -t probabilities.py``

"""
from mpmath import mp

from collections import namedtuple
from functools import partial
from memoize import memoize

Probabilities = namedtuple(
    "Probabilities", ("d", "n", "k", "gr", "ngr", "pf", "ngr_pf", "gr_pf", "rho", "eta", "beta", "prec")
)


def C(d, theta, integrate=False, prec=None):
    """
    The probability that some v from the sphere has angle at most θ with some fixed u.

    :param d: We consider spheres of dimension `d-1`
    :param theta: angle in radians
    :param: compute via explicit integration
    :param: precision to use

    EXAMPLE::

        sage: C(80, pi/3)
        mpf('1.0042233739846629e-6')

    """
    prec = prec if prec else mp.prec
    with mp.workprec(prec):
        theta = mp.mpf(theta)
        d = mp.mpf(d)
        if integrate:
            r = (
                1
                / mp.sqrt(mp.pi)
                * mp.gamma(d / 2)
                / mp.gamma((d - 1) / 2)
                * mp.quad(lambda x: mp.sin(x) ** (d - 2), (0, theta), error=True)[0]
            )
        else:
            r = mp.betainc((d - 1) / 2, 1 / 2.0, x2=mp.sin(theta) ** 2, regularized=True) / 2
        return r


def A(d, theta, prec=53):
    """
    The density of the event that some v from the sphere has angle θ with some fixed u.

    :param d: We consider spheres of dimension `d-1`
    :param theta: angle in radians

    :param: compute via explicit integration
```

```
    :param: precision to use

    EXAMPLES::

        sage: A(80, pi/3)
        mpf('4.7395659506025816e-5')

        sage: A(80, pi/3) * 2*pi/100000
        mpf('2.9779571143234787e-9')

        sage: C(80, pi/3+pi/100000) - C(80, pi/3-pi/100000)
        mpf('2.9779580567976835e-9')

    """
    prec = prec if prec else mp.prec
    with mp.workprec(prec):
        theta = mp.mpf(theta)
        d = mp.mpf(d)
        r = 1 / mp.sqrt(mp.pi) * mp.gamma(d / 2) / mp.gamma((d - 1) / 2) * mp.sin(theta) ** (d - 2)
        return r


@memoize
def log2_sphere(d):
    # NOTE: hardcoding 53 here
    with mp.workprec(53):
        return (d / 2 * mp.log(mp.pi, 2) + 1) / mp.gamma(d / 2)


@memoize
def sphere(d):
    # NOTE: hardcoding 53 here
    with mp.workprec(53):
        return 2 ** (d / 2 * mp.log(mp.pi, 2) + 1) / mp.gamma(d / 2)


@memoize
def W(d, alpha, beta, theta, integrate=True, prec=None):
    assert alpha <= mp.pi / 2
    assert beta <= mp.pi / 2
    assert 0 >= (mp.cos(beta) - mp.cos(alpha) * mp.cos(theta)) * (mp.cos(beta) * mp.cos(theta) - mp.cos(alpha))

    if theta >= alpha + beta:
        return mp.mpf(0.0)

    prec = prec if prec else mp.prec
    with mp.workprec(prec):
        alpha = mp.mpf(alpha)
        beta = mp.mpf(beta)
        theta = mp.mpf(theta)
        d = mp.mpf(d)
        if integrate:
            c = mp.atan(mp.cos(alpha) / (mp.cos(beta) * mp.sin(theta)) - 1 / mp.tan(theta))

            def f_alpha(x):
                return mp.sin(x) ** (d - 2) * mp.betainc(
                    (d - 2) / 2,
                    1 / 2.0,
                    x2=mp.sin(mp.re(mp.acos(mp.tan(theta - c) / mp.tan(x)))) ** 2,
                    regularized=True,
                )

            def f_beta(x):
                return mp.sin(x) ** (d - 2) * mp.betainc(
                    (d - 2) / 2, 1 / 2.0, x2=mp.sin(mp.re(mp.acos(mp.tan(c) / mp.tan(x)))) ** 2, regularized=True
                )

            S_alpha = mp.quad(f_alpha, (theta - c, alpha), error=True)[0] / 2
            S_beta = mp.quad(f_beta, (c, beta), error=True)[0] / 2

            return (S_alpha + S_beta) * sphere(d - 1) / sphere(d)
        else:
            # Wedge volume formula from Lemma 2.2 of [BDGL16] Anja Becker, Léo Ducas, Nicolas Gama,
            # Thijs Laarhoven. "New directions in nearest neighbor searching with applications to
            # lattice sieving." SODA 2016. https://eprint.iacr.org/2015/1128
            # g_sq = (mp.cos(alpha)**2 + mp.cos(beta)**2 -
            # 2*mp.cos(alpha)*mp.cos(beta)*mp.cos(theta))/mp.sin(theta)**2
            # log2_A = mp.log(g_sq, 2) - 2*mp.log(1-g_sq, 2)
            # r = (d-4) * mp.log(mp.sqrt(1-g_sq), 2) + log2_A - 2*mp.log(d-4, 2) + log2_sphere(d-2) - log2_sphere(d)
            # return 2**r
            raise NotImplementedError("Results don't match.")


@memoize
def binomial(n, i):
    # NOTE: hardcoding 53 here
    with mp.workprec(53):
        return mp.binomial(n, i)
```

```
@memoize
def P(n, k, theta, prec=None):
    """
    Probability that two vectors with angle θ pass a popcount filter

    :param n: number of popcount vectors
    :param k: number of popcount tests required to pass
    :param theta: angle in radians

    """
    prec = prec if prec else mp.prec
    with mp.workprec(prec):
        theta = mp.mpf(theta)
        # binomial cdf for 0 <= successes <= k
        # r = 0
        # for i in range(k):
        #     r += binomial(n, i) * (theta/mp.pi)**i * (1-theta/mp.pi)**(n-i)
        # r = mp.betainc(n-k, k+1, x2=1-(theta/mp.pi), regularized=True)
        # NOTE: This routine uses obscene precision

        def _betainc(a, b, x2):
            return (
                x2 ** a
                * mp.hyp2f1(
                    a, 1 - b, a + 1, x2, maxprec=2 ** mp.ceil(2 * mp.log(n, 2)), maxterms=2 ** mp.ceil(mp.log(n, 2))
                )
                / a
                / mp.beta(a, b)
            )

        r = _betainc(n - k, k + 1, x2=1 - (theta / mp.pi))
        return r


def pf(d, n, k, beta=None, lb=None, ub=None, beta_and=False, prec=None):
    """
    Let `Pr[P_{k,n}]` be the probability that a popcount filter passes.  We assume the probability
    is over the vectors `u,v`. Let `¬G` be the event that two random vectors are not Gauss reduced.

    We start with Pr[P_{k,n}]::

        sage: pf(80, 128, 40)
        mpf('0.00031063713572376122')

        sage: pf(80, 128, 128)
        mpf('1.0000000000000002')

    Pr[P_{k,n} ∧ ¬G]::

        sage: pf(80, 128, 40, ub=mp.pi/3)
        mpf('3.359809258955273 2e-7')

    Pr[¬G]::

        sage: pf(80, 128, 128, ub=mp.pi/3)
        mpf('1.0042233739846644e-6')

        sage: ngr_pf(80, 128, 128)
        mpf('1.0042233739846644e-6')

        sage: ngr(80)
        mpf('1.0042233739846629e-6')

    Pr[Pr_{k,n} ∧ G]::

        sage: pf(80, 128, 40, lb=mp.pi/3)
        mpf('0.00031030115479786595')

    Pr[G]::

        sage: pf(80, 128, 128, lb=mp.pi/3)
        mpf('0.99999899577662632')

        sage: gr_pf(80, 128, 128)
        mpf('0.99999899577662632')

        sage: gr(80)
        mpf('0.99999899577662599')

    Pr[P_{k,n} | C(w,β)]::

        sage: pf(80, 128, 40, beta=mp.pi/3)
        mpf('0.019786655048072234')

    Pr[P_{k,n}  ∧ ¬G | C(w,β)]::

        sage: pf(80, 128, 40, beta=mp.pi/3, ub=mp.pi/3)
        mpf('0.00077177364924089652')

    Pr[¬G | C(w,β)]::
```

```
        sage: pf(80, 128, 128, beta=mp.pi/3, ub=mp.pi/3)
        mpf('0.0021964683579090904')
        sage: ngr_pf(80, 128, 128, beta=mp.pi/3)
        mpf('0.0021964683579090904')
        sage: ngr(80, beta=mp.pi/3)
        mpf('0.0021964683579090904')

    Pr[Pr_{k,n} ∧ G | C(w,β)]::

        sage: pf(80, 128, 40, beta=mp.pi/3, lb=mp.pi/3)
        mpf('0.019014953591444488')

        sage: gr_pf(80, 128, 40, beta=mp.pi/3)
        mpf('0.019014953591444488')

    Pr[G | C(w,β)]::

        sage: pf(80, 128, 128, beta=mp.pi/3, lb=mp.pi/3)
        mpf('0.99780353164285229')
        sage: gr_pf(80, 128, 128, beta=mp.pi/3)
        mpf('0.99780353164285229')
        sage: gr(80, beta=mp.pi/3)
        mpf('0.9978035316420909')

    :param d: We consider the sphere `S^{d-1}`
    :param n: Number of popcount vectors
    :param k: popcount threshold
    :param beta: If not ``None`` vectors are considered in a bucket around some `w` with angle β.
    :param lb: lower bound of integration (see above)
    :param ub: upper bound of integration (see above)
    :param beta_and: return Pr[P_{k,n} ∧ C(w,β)] instead of Pr[P_{k,n} | C(w,β)]
    :param prec: compute with this precision

    """
    prec = prec if prec else mp.prec
    with mp.workprec(prec):
        if lb is None:
            lb = 0
        if ub is None:
            ub = mp.pi
        if beta is None:
            return mp.quad(lambda x: P(n, k, x) * A(d, x), (lb, ub), error=True)[0]
        else:
            num = mp.quad(lambda x: P(n, k, x) * W(d, beta, beta, x) * A(d, x), (lb, min(ub, 2 * beta)), error=True)[0]
            if not beta_and:
                den = mp.quad(lambda x: W(d, beta, beta, x) * A(d, x), (0, 2 * beta), error=True)[0]
            else:
                den = 1
            return num / den


ngr_pf = partial(pf, lb=0, ub=mp.pi / 3)
gr_pf = partial(pf, lb=mp.pi / 3)


def ngr(d, beta=None, prec=None):
    """
    Probability that two random vectors (in a cap parameterised by β) are not Gauss reduced.

    :param d: We consider the sphere `S^{d-1}`
    :param beta: If not ``None`` vectors are considered in a bucket around some `w` with angle β.
    :param prec: compute with this precision

    """
    prec = prec if prec else mp.prec
    with mp.workprec(prec):
        if beta is None:
            return C(d, mp.pi / 3)
        elif beta < mp.pi / 6:
            return mp.mpf(1.0)
        else:
            # Pr[¬G ∧ E]
            num = mp.quad(lambda x: W(d, beta, beta, x) * A(d, x), (0, mp.pi / 3), error=True)[0]
            # Pr[E]
            den = mp.quad(lambda x: W(d, beta, beta, x) * A(d, x), (0, 2 * beta), error=True)[0]
            # Pr[¬G | E] = Pr[¬G ∧ E]/Pr[E]
            return num / den


def gr(d, beta=None, prec=None):
    """
    Probability that two random vectors (in a cap parameterised by β) are Gauss reduced.

    :param d: We consider the sphere `S^{d-1}`
    :param beta: If not ``None`` vectors are considered in a bucket around some `w` with angle β.
    :param prec: compute with this precision

    """
    prec = prec if prec else mp.prec
```

```
    with mp.workprec(prec):
        return 1 - ngr(d, beta, prec)


def probabilities(d, n, k, beta=None, prec=None):
    """
    Useful probabilities.

    :param d: We consider the sphere `S^{d-1}`
    :param n: Number of popcount vectors
    :param k: popcount threshold
    :param beta: If not ``None`` vectors are considered in a bucket around some `w` with angle β.
    :param prec: compute with this precision

    """
    prec = prec if prec else mp.prec

    with mp.workprec(prec):
        pf_ = pf(d, n, k, beta=beta, prec=prec)
        ngr_ = ngr(d, beta=beta, prec=prec)
        ngr_pf_ = ngr_pf(d, n, k, beta=beta, prec=prec)
        gr_pf_ = gr_pf(d, n, k, beta=beta, prec=prec)
        rho = 1 - ngr_pf_ / pf_
        eta = 1 - ngr_pf_ / ngr_

        probs = Probabilities(
            d=d,
            n=n,
            k=k,
            ngr=ngr_,
            gr=1 - ngr_,
            pf=pf_,
            gr_pf=gr_pf_,
            ngr_pf=ngr_pf_,
            rho=rho,
            eta=eta,
            beta=beta,
            prec=prec,
        )
        return probs
```

G.3 cost.py

cost.py provides functions to cost quantum and classical circuits. The main functions are `all_pairs`, `random_buckets` and `list_decoding` which correspond to the algorithms with the same name. Each algorithm returns a specific Python namedtuple.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Quantum and Classical Nearest Neighbor Cost.
"""

from mpmath import mp
from collections import namedtuple
from utils import load_probabilities, PrecomputationRequired
from config import MagicConstants
from probabilities import W, C, pf, ngr_pf, ngr
from ge19 import estimate_abstract_to_physical

"""
COSTS
"""

"""
Logical Quantum Costs

:param label: arbitrary label
:param qubits_in: number of input qubits
:param qubits_out: number of output qubits
:param qubits_max:
:param depth: longest path from input to output (including identity gates)
:param gates: gates except identity gates
:param dw: not necessarily depth*qubits
:param toffoli_count: number of Toffoli gates
:param t_count: number of T gates
:param t_depth: T gate depth

"""

LogicalCosts = namedtuple(
```

```
    "LogicalCosts",
    (
        "label",
        "qubits_in",
        "qubits_out",
        "qubits_max",  # NOTE : not sure if this is useful
        "depth",
        "gates",
        "dw",
        "toffoli_count",  # NOTE: not sure if this is useful
        "t_count",
        "t_depth",
    ),
)

"""
Classic Costs

:param label: arbitrary label
:param gates: number of gates
:param depth: longest path from input to output

"""

ClassicalCosts = namedtuple("ClassicalCosts", ("label", "gates", "depth"))

"""
METRICS
"""

ClassicalMetrics = {"classical", "naive_classical"}

QuantumMetrics = {
    "g",  # gate count
    "dw",  # depth x width
    "ge19",  # depth x width x physical qubit measurements Gidney Ekera
    "t_count",  # number of T-gates
    "naive_quantum",  # query cost
}

Metrics = ClassicalMetrics | QuantumMetrics


def log2(x):
    return mp.log(x) / mp.log(2)


def local_min(f, x, d1=1, d2=5, low=None, high=None):
    """
    Search the neighborhood around ``f(x)`` for a local minimum between ``low`` and ``high``.

    ..  note :: We could replace this function with a call to ``scipy.optimize.fminbound``.
    However, this doesn't seem to be faster.  Also, ``f(x)`` is not necessarily defined on all of
    `[low...high]`, raising an ``AssertionError`` which would need to be caught by a wrapper around
    ``f``.

    :param f: function to call
    :param x: initial guess for ``x`` minimizing ``f(x)``
    :param d1: We move in steps of size `0.1^{d1}`...0.1^{d2}`, starting with ``d1``
    :param d2: We move in steps of size `0.1^{d1}`...0.1^{d2}`, finishing at ``d2``
    :param low: lower bound on input space
    :param high: upper bound on input space

    """
    y = f(x)
    for k in range(d1, d2 + 1):
        d = 0.1 ** k
        y2 = f(x + d) if x + d < high else f(high)
        if y2 > y:
            d = -d
            y2 = f(x + d) if x + d > low else f(low)
        while (y2 < y) and (low < x + d) and (x + d < high):
            y = y2
            x = x + d
            y2 = f(x)
    return x


def null_costf(qubits_in=0, qubits_out=0):
    """
    Cost of initialization/measurement.
    """

    return LogicalCosts(
        label="null",
        qubits_in=qubits_in,
        qubits_out=qubits_out,
        qubits_max=max(qubits_in, qubits_out),
        gates=0,
        depth=0,
```

```
        dw=0,
        toffoli_count=0,
        t_count=0,
        t_depth=0,
    )


def delay(cost, depth, label="_"):
    # delay only affects the dw cost
    dw = cost.dw + cost.qubits_out * depth
    return LogicalCosts(
        label=label,
        qubits_in=cost.qubits_in,
        qubits_out=cost.qubits_out,
        qubits_max=cost.qubits_max,
        gates=cost.gates,
        depth=cost.depth + depth,
        dw=dw,
        toffoli_count=cost.toffoli_count,
        t_count=cost.t_count,
        t_depth=cost.t_depth,
    )


def reverse(cost):
    return LogicalCosts(
        label=cost.label,
        qubits_in=cost.qubits_out,
        qubits_out=cost.qubits_in,
        qubits_max=cost.qubits_max,
        gates=cost.gates,
        depth=cost.depth,
        dw=cost.dw,
        toffoli_count=cost.toffoli_count,
        t_count=cost.t_count,
        t_depth=cost.t_depth,
    )


def compose_k_sequential(cost, times, label="_"):
    # Ensure that sequential composition makes sense
    assert cost.qubits_in == cost.qubits_out

    return LogicalCosts(
        label=label,
        qubits_in=cost.qubits_in,
        qubits_out=cost.qubits_out,
        qubits_max=cost.qubits_max,
        gates=cost.gates * times,
        depth=cost.depth * times,
        dw=cost.dw * times,
        toffoli_count=cost.toffoli_count * times,
        t_count=cost.t_count * times,
        t_depth=cost.t_depth * times,
    )


def compose_k_parallel(cost, times, label="_"):
    return LogicalCosts(
        label=label,
        qubits_in=times * cost.qubits_in,
        qubits_out=times * cost.qubits_out,
        qubits_max=times * cost.qubits_max,
        gates=times * cost.gates,
        depth=cost.depth,
        dw=times * cost.dw,
        toffoli_count=times * cost.toffoli_count,
        t_count=times * cost.t_count,
        t_depth=cost.t_depth,
    )


def compose_sequential(cost1, cost2, label="_"):
    # Ensure that sequential composition makes sense
    assert cost1.qubits_out >= cost2.qubits_in

    # Pad unused wires with identity gates
    dw = cost1.dw + cost2.dw
    if cost1.qubits_out > cost2.qubits_in:
        dw += (cost1.qubits_out - cost2.qubits_in) * cost2.depth
    qubits_out = cost1.qubits_out - cost2.qubits_in + cost2.qubits_out
    qubits_max = max(cost1.qubits_max, cost1.qubits_out - cost2.qubits_in + cost2.qubits_max)

    return LogicalCosts(
        label=label,
        qubits_in=cost1.qubits_in,
        qubits_out=qubits_out,
        qubits_max=qubits_max,
        gates=cost1.gates + cost2.gates,
        depth=cost1.depth + cost2.depth,
```

```
        dw=dw,
        toffoli_count=cost1.toffoli_count + cost2.toffoli_count,
        t_count=cost1.t_count + cost2.t_count,
        t_depth=cost1.t_depth + cost2.t_depth,
    )


def compose_parallel(cost1, cost2, label="_"):
    # Pad wires from shallower circuit with identity gates
    dw = cost1.dw + cost2.dw
    if cost1.depth >= cost2.depth:
        dw += (cost1.depth - cost2.depth) * cost2.qubits_out
    else:
        dw += (cost2.depth - cost1.depth) * cost1.qubits_out

    return LogicalCosts(
        label=label,
        qubits_in=cost1.qubits_in + cost2.qubits_in,
        qubits_out=cost1.qubits_out + cost2.qubits_out,
        qubits_max=cost1.qubits_max + cost2.qubits_max,
        gates=cost1.gates + cost2.gates,
        depth=max(cost1.depth, cost2.depth),
        dw=dw,
        toffoli_count=cost1.toffoli_count + cost2.toffoli_count,
        t_count=cost1.t_count + cost2.t_count,
        t_depth=max(cost1.t_depth, cost2.t_depth),
    )


def classical_popcount_costf(n, k):
    """
    Classical gate count for popcount.

    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on ≤ k

    """
    ell = mp.ceil(mp.log(n, 2))
    t = mp.ceil(mp.log(k, 2))
    gates = 11 * n - 9 * ell - 10
    depth = 2 * ell

    cc = ClassicalCosts(label="popcount", gates=gates, depth=depth)

    return cc


def adder_costf(i, ci=False):
    """
    Logical cost of i bit adder (Cuccaro et al). With Carry Input if ci=True

    """
    adder_cnots = 6 if i == 1 else (5 * i + 1 if ci else 5 * i - 3)
    adder_depth = 7 if i == 1 else (2 * i + 6 if ci else 2 * i + 4)
    adder_nots = 0 if i == 1 else (2 * i - 2 if ci else 2 * i - 4)
    adder_tofs = 2 * i - 1
    adder_qubits_in = 2 * i + 1
    adder_qubits_out = 2 * i + 2
    adder_qubits_max = 2 * i + 2
    adder_t_depth = adder_tofs * MagicConstants.t_depth_div_toffoli
    adder_t_count = adder_tofs * MagicConstants.t_div_toffoli
    adder_gates = adder_cnots + adder_nots + adder_tofs * MagicConstants.gates_div_toffoli

    return LogicalCosts(
        label=str(i) + "-bit adder",
        qubits_in=adder_qubits_in,
        qubits_out=adder_qubits_out,
        qubits_max=adder_qubits_max,
        gates=adder_gates,
        depth=adder_depth,
        dw=adder_qubits_in * adder_depth,
        toffoli_count=adder_tofs,
        t_count=adder_t_count,
        t_depth=adder_t_depth,
    )


def hamming_wt_costf(n):
    """
    Logical cost of mapping |v>|0> to |v>|H(v)>.

    ..  note :: The adder tree uses in-place addition, so some of the bits of |v> overlap |H(v)> and
    there are ancilla as well.

    :param n: number of bits in v

    """
    b = int(mp.floor(log2(n)))
    qc = null_costf(qubits_in=n, qubits_out=n)
    if bin(n + 1).count("1") == 1:
```

```
        # When n = 2**(b+1) - 1 the adder tree is "packed". We can use every input bit including
        # carry inputs.
        for i in range(1, b + 1):
            L = compose_k_parallel(adder_costf(i, ci=True), 2 ** (b - i))
            qc = compose_sequential(qc, L)
    else:
        # Decompose into packed adder trees joined by adders.
        # Use one adder tree on (2**b - 1) bits and one on max(1, n - 2**b) bits.
        # Reserve one bit for carry input of adder (unless n = 2**b).
        carry_in = n != 2 ** b
        qc = compose_sequential(
            qc, compose_parallel(hamming_wt_costf(2 ** b - 1), hamming_wt_costf(max(1, n - 2 ** b)))
        )
        qc = compose_sequential(qc, adder_costf(b, ci=carry_in))

    qc = compose_parallel(qc, null_costf(), label=str(n) + "-bit hamming weight")
    return qc


def carry_costf(m):
    """
    Logical cost of mapping |x> to (-1)^{(x+c)_m}|x> where (x+c)_m is the m-th bit (zero indexed) of
    x+c for an arbitrary m bit constant c.

    ..  note :: numbers here are adapted from Fig 3 of https://arxiv.org/pdf/1611.07995.pdf
                m is equivalent to \ell in the LaTeX

    """
    if m < 2:
        raise NotImplementedError("Case m==1 not implemented.")

    carry_cnots = 2 * m
    carry_depth = 8 * m - 8
    carry_nots = 2 * (m - 1)
    carry_tofs = 4 * (m - 2) + 2
    carry_qubits_in = 2 * m
    carry_qubits_out = 2 * m
    carry_qubits_max = 2 * m
    carry_dw = carry_qubits_max * carry_depth
    carry_t_depth = carry_tofs * MagicConstants.t_depth_div_toffoli
    carry_t_count = carry_tofs * MagicConstants.t_div_toffoli
    carry_gates = carry_cnots + carry_nots + carry_tofs * MagicConstants.gates_div_toffoli

    return LogicalCosts(
        label="carry",
        qubits_in=carry_qubits_in,
        qubits_out=carry_qubits_out,
        qubits_max=carry_qubits_max,
        gates=carry_gates,
        depth=carry_depth,
        dw=carry_dw,
        toffoli_count=carry_tofs,
        t_count=carry_t_count,
        t_depth=carry_t_depth,
    )


def popcount_costf(L, n, k):
    """
    Logical cost of mapping |i> to (-1)^{popcount(u,v_i)}|i> for fixed u.

    :param L: length of the list, i.e. |L|
    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on ≤ k

    """
    assert 0 <= k and k <= n

    index_wires = int(mp.ceil(log2(L)))

    # Initialize space for |v_i>
    qc = null_costf(qubits_in=index_wires, qubits_out=n + index_wires)

    # Query table index i
    # NOTE: We're skipping a qRAM call here.
    qc = delay(qc, 1)

    # XOR in the fixed sketch "u"
    # NOTE: We're skipping ~ n NOT gates for mapping |v> to |u^v>
    qc = delay(qc, 1)

    # Use tree of adders compute hamming weight
    #      |i>|u^v_i>|0>     ->    |i>|u^v_i>|wt(u^v_i)>
    hamming_wt = hamming_wt_costf(n)
    qc = compose_sequential(qc, null_costf(qubits_in=qc.qubits_out, qubits_out=index_wires + hamming_wt.qubits_in))
    qc = compose_sequential(qc, hamming_wt)

    # Compute the high bit of (2^ceil(log(n)) - k) + hamming_wt
    #      |i>|v_i>|wt(u^v_i)>   ->     (-1)^popcnt(u,v_i) |i>|u^v_i>|wt(u^v_i)>
    qc = compose_sequential(qc, carry_costf(int(mp.ceil(log2(n)))))
```

```
    # Uncompute hamming weight.
    qc = compose_sequential(qc, reverse(hamming_wt))

    # Uncompute XOR
    # NOTE: We're skipping ~ n NOT gates for mapping |u^v> to |v>
    qc = delay(qc, 1)

    # Uncompute table entry
    # NOTE: We're skipping a qRAM call here.
    qc = delay(qc, 1)

    # Discard ancilla
    # (-1)^popcnt(u,v_i) |i>|0>|0>   ->    (-1)^popcnt(u,v_i) |i>

    qc = compose_sequential(qc, null_costf(qubits_in=qc.qubits_out, qubits_out=index_wires))

    qc = compose_parallel(qc, null_costf(), label="popcount" + str((n, k)))

    return qc


def n_toffoli_costf(n, have_ancilla=False):
    """
    Logical cost of toffoli with n-1 controls.

    ..  note :: Table I of Maslov arXiv:1508.03273v2 (Source = "Ours", Optimization goal = "T/CNOT")

    """

    assert n >= 3

    if n >= 5 and not have_ancilla:
        # Use Barenco et al (1995) Lemma 7.3 split into two smaller Toffoli gates.
        n1 = int(mp.ceil((n - 1) / 2.0)) + 1
        n2 = n - n1 + 1
        return compose_sequential(
            compose_parallel(null_costf(qubits_in=n - n1, qubits_out=n - n1), n_toffoli_costf(n1, True)),
            compose_parallel(null_costf(qubits_in=n - n2, qubits_out=n - n2), n_toffoli_costf(n2, True)),
        )

    if n == 3:  # Normal toffoli gate
        n_tof_t_count = MagicConstants.AMMR12_tof_t_count
        n_tof_t_depth = MagicConstants.AMMR12_tof_t_depth
        n_tof_gates = MagicConstants.AMMR12_tof_gates
        n_tof_depth = MagicConstants.AMMR12_tof_depth
        n_tof_dw = n_tof_depth * (n + 1)
    elif n == 4: # Note: the cost can be smaller if using "clean" ancillas
                 # (see first "Ours" in Table 1 of Maslov's paper)
        n_tof_t_count = 16
        n_tof_t_depth = 16
        n_tof_gates = 36
        n_tof_depth = 36  # Maslov Eq. (5), Figure 3 (dashed), Eq. (3) (dashed).
        n_tof_dw = n_tof_depth * (n + 1)
    elif n >= 5:
        n_tof_t_count = 8 * n - 16
        n_tof_t_depth = 8 * n - 16
        n_tof_gates = (8 * n - 16) + (8 * n - 20) + (4 * n - 10)
        n_tof_depth = (8 * n - 16) + (8 * n - 20) + (4 * n - 10)
        n_tof_dw = n_tof_depth * (n + 1)

    n_tof_qubits_max = n if have_ancilla else n + 1

    return LogicalCosts(
        label=str(n) + "-toffoli",
        qubits_in=n,
        qubits_out=n,
        qubits_max=n_tof_qubits_max,
        gates=n_tof_gates,
        depth=n_tof_depth,
        dw=n_tof_dw,
        toffoli_count=0,
        t_count=n_tof_t_count,
        t_depth=n_tof_t_depth,
    )


def diffusion_costf(L):
    """
    Logical cost of the diffusion operator D R_0 D^-1

    where D samples the uniform distribution on {1,...,L} R_0 is the unitary I - 2|0><0|

    :param L: length of the list, i.e. |L|
    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on ≤ k

    """
    index_wires = int(mp.ceil(log2(L)))
```

```
    H = LogicalCosts(
        label="H",
        qubits_in=1,
        qubits_out=1,
        qubits_max=1,
        gates=1,
        depth=1,
        dw=1,
        toffoli_count=0,
        t_count=0,
        t_depth=0,
    )
    Hn = compose_k_parallel(H, index_wires)

    anc = null_costf(qubits_in=index_wires, qubits_out=index_wires + 1)

    qc = compose_sequential(Hn, anc)
    qc = compose_sequential(qc, n_toffoli_costf(index_wires + 1))
    qc = compose_sequential(qc, reverse(anc))
    qc = compose_sequential(qc, Hn)

    qc = compose_parallel(qc, null_costf(), label="diffusion")
    return qc


def popcount_grover_iteration_costf(L, n, k):
    """
    Logical cost of G(popcount) = (D R_0 D^-1) R_popcount.

    where D samples the uniform distribution on {1,...,L} (D R_0 D^-1) is the diffusion operator.
    R_popcount maps |i> to (-1)^{popcount(u,v_i)}|i> for some fixed u

    :param L: length of the list, i.e. |L|
    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on <= k

    """

    popcount_cost = popcount_costf(L, n, k)
    diffusion_cost = diffusion_costf(L)

    return compose_sequential(diffusion_cost, popcount_cost, label="oracle")


def popcounts_dominate_cost(positive_rate, d, n, metric):
    ip_div_pc = (MagicConstants.word_size ** 2) * d / float(n)
    if metric in ClassicalMetrics:
        return 1.0 / positive_rate > ip_div_pc
    else:
        return 1.0 / positive_rate > ip_div_pc ** 2


def raw_cost(cost, metric):
    if metric == "g":
        result = cost.gates
    elif metric == "dw":
        result = cost.dw
    elif metric == "ge19":
        phys = estimate_abstract_to_physical(
            cost.toffoli_count, cost.qubits_max, cost.depth, prefers_parallel=False, prefers_serial=True
        )
        result = cost.dw * phys[0] ** 2
    elif metric == "t_count":
        result = cost.t_count
    elif metric == "classical":
        result = cost.gates
    elif metric == "naive_quantum":
        return 1
    elif metric == "naive_classical":
        return 1
    else:
        raise ValueError("Unknown metric '%s'" % metric)
    return result


AllPairsResult = namedtuple("AllPairsResult", ("d", "n", "k", "log_cost", "pf_inv", "metric", "detailed_costs"))


def all_pairs(d, n=None, k=None, optimize=True, metric="dw", allow_suboptimal=False):
    """
    Nearest Neighbor Search via a quadratic search over all pairs.

    :param d: search in \(S^{d-1}\)
    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on ≤ k
    :param optimize: optimize `n`
    :param metric: target metric
    :param allow_suboptimal: when ``optimize=True``, return the best possible set of parameters given what is precomputed

    """
```

```
    if n is None:
        n = 1
        while n < d:
            n = 2 * n

    k = k if k else int(MagicConstants.k_div_n * (n-1))

    pr = load_probabilities(d, n-1, k)

    def cost(pr):
        N = 2 / ((1 - pr.eta) * C(pr.d, mp.pi / 3))

        if metric in ClassicalMetrics:
            look_cost = classical_popcount_costf(pr.n, pr.k)
            looks = (N ** 2 - N) / 2.0
        else:
            look_cost = popcount_grover_iteration_costf(N, pr.n, pr.k)
            looks_factor = 11.0/15
            looks = int(mp.ceil(looks_factor * N ** (3 / 2.0)))

        full_cost = looks * raw_cost(look_cost, metric)
        return full_cost, look_cost

    positive_rate = pf(pr.d, pr.n, pr.k)
    while optimize and not popcounts_dominate_cost(positive_rate, pr.d, pr.n, metric):
        try:
            pr = load_probabilities(pr.d, 2 * (pr.n+1) - 1, int(MagicConstants.k_div_n * (2 * (pr.n+1) - 1)))
        except PrecomputationRequired as e:
            if allow_suboptimal:
                break
            else:
                raise e
        positive_rate = pf(pr.d, pr.n, pr.k)

    fc, dc = cost(pr)

    return AllPairsResult(
        d=pr.d,
        n=pr.n,
        k=pr.k,
        log_cost=float(log2(fc)),
        pf_inv=int(round(1 / positive_rate)),
        metric=metric,
        detailed_costs=dc,
    )


RandomBucketsResult = namedtuple(
    "RandomBucketsResult", ("d", "n", "k", "theta", "log_cost", "pf_inv", "eta", "metric", "detailed_costs")
)


def random_buckets(d, n=None, k=None, theta1=None, optimize=True, metric="dw", allow_suboptimal=False):
    """
    Nearest Neighbor Search using random buckets as in BGJ1.

    :param d: search in \(S^{d-1}\)
    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on ≤ k
    :param theta1: bucket angle
    :param optimize: optimize `n`
    :param metric: target metric
    :param allow_suboptimal: when ``optimize=True``, return the best possible set of parameters
        given what is precomputed

    """
    if n is None:
        n = 1
        while n < d:
            n = 2 * n

    k = k if k else int(MagicConstants.k_div_n * (n-1))
    theta = theta1 if theta1 else 1.2860
    pr = load_probabilities(d, n-1, k)
    ip_cost = MagicConstants.word_size ** 2 * d

    def cost(pr, T1):
        eta = 1 - ngr_pf(pr.d, pr.n, pr.k, beta=T1)/ngr(pr.d, beta=T1)
        N = 2 / ((1 - eta) * C(pr.d, mp.pi / 3))
        W0 = W(pr.d, T1, T1, mp.pi / 3)
        buckets = 1.0 / W0
        bucket_size = N * C(pr.d, T1)

        if metric in ClassicalMetrics:
            look_cost = classical_popcount_costf(pr.n, pr.k)
            looks_per_bucket = (bucket_size ** 2 - bucket_size) / 2.0
        else:
            look_cost = popcount_grover_iteration_costf(N, pr.n, pr.k)
            looks_factor = (2 * W0) / (5 * C(pr.d, T1)) + 1.0 / 3
            looks_per_bucket = looks_factor * bucket_size ** (3 / 2.0)
```

```
        fill_bucket_cost = N * ip_cost
        search_bucket_cost = looks_per_bucket * raw_cost(look_cost, metric)
        full_cost = buckets * (fill_bucket_cost + search_bucket_cost)

        return full_cost, look_cost, eta

    if optimize:
        theta = local_min(lambda T: cost(pr, T)[0], theta, low=mp.pi / 6, high=mp.pi / 2)
        positive_rate = pf(pr.d, pr.n, pr.k, beta=theta)
        while not popcounts_dominate_cost(positive_rate, pr.d, pr.n, metric):
            try:
                pr = load_probabilities(pr.d, 2 * (pr.n + 1) - 1, int(MagicConstants.k_div_n * (2 * (pr.n + 1) - 1)))
            except PrecomputationRequired as e:
                if allow_suboptimal:
                    break
                else:
                    raise e
            theta = local_min(lambda T: cost(pr, T)[0], theta, low=mp.pi / 6, high=mp.pi / 2)
            positive_rate = pf(pr.d, pr.n, pr.k, beta=theta)
    else:
        positive_rate = pf(pr.d, pr.n, pr.k, beta=theta)

    fc, dc, eta = cost(pr, theta)

    return RandomBucketsResult(
        d=pr.d,
        n=pr.n,
        k=pr.k,
        theta=float(theta),
        log_cost=float(log2(fc)),
        pf_inv=int(round(1 / positive_rate)),
        eta=eta,
        metric=metric,
        detailed_costs=dc,
    )


ListDecodingResult = namedtuple(
    "ListDecodingResult", ("d", "n", "k", "theta1", "theta2", "log_cost", "pf_inv", "eta", "metric", "detailed_costs")
)


def list_decoding(d, n=None, k=None, theta1=None, theta2=None, optimize=True, metric="dw", allow_suboptimal=False):
    """
    Nearest Neighbor Search via a decodable buckets as in BDGL16.

    :param d: search in \(S^{d-1}\)
    :param n: number of entries in popcount filter
    :param k: we accept if two vectors agree on ≤ k
    :param theta1: filter creation angle
    :param theta2: filter query angle
    :param optimize: optimize `n`
    :param metric: target metric
    :param allow_suboptimal: when ``optimize=True``, return the best possible set of parameters
        given what is precomputed
    """

    if n is None:
        n = 1
        while n < d:
            n = 2 * n

    k = k if k else int(MagicConstants.k_div_n * (n-1))
    theta = theta1 if theta1 else mp.pi / 3
    pr = load_probabilities(d, n-1, k)
    ip_cost = MagicConstants.word_size ** 2 * d

    def cost(pr, T1):
        eta = 1 - ngr_pf(pr.d, pr.n, pr.k, beta=T1)/ngr(pr.d, beta=T1)
        T2 = T1
        N = 2 / ((1 - eta) * C(d, mp.pi / 3))
        W0 = W(d, T1, T2, mp.pi / 3)
        filters = 1.0 / W0
        insert_cost = filters * C(d, T2) * ip_cost
        query_cost = filters * C(d, T1) * ip_cost
        bucket_size = (filters * C(d, T1)) * (N * C(d, T2))

        if metric in ClassicalMetrics:
            look_cost = classical_popcount_costf(pr.n, pr.k)
            looks_per_bucket = bucket_size
            search_one_cost = ClassicalCosts(
                label="search", gates=look_cost.gates * looks_per_bucket, depth=look_cost.depth * looks_per_bucket
            )
        else:
            look_cost = popcount_grover_iteration_costf(N, pr.n, pr.k)
            looks_per_bucket = bucket_size ** (1 / 2.0)
            search_one_cost = compose_k_sequential(look_cost, looks_per_bucket)

        search_cost = raw_cost(search_one_cost, metric)
```

```
        return N * insert_cost + N * query_cost + N * search_cost, search_one_cost, eta

    if optimize:
        theta = local_min(lambda T: cost(pr, T)[0], theta, low=mp.pi / 6, high=mp.pi / 2)
        positive_rate = pf(pr.d, pr.n, pr.k, beta=theta)
        while not popcounts_dominate_cost(positive_rate, pr.d, pr.n, metric):
            try:
                pr = load_probabilities(pr.d, 2 * (pr.n + 1) - 1, int(MagicConstants.k_div_n * (2 * (pr.n + 1) - 1)))
            except PrecomputationRequired as e:
                if allow_suboptimal:
                    break
                else:
                    raise e
            theta = local_min(lambda T: cost(pr, T)[0], theta, low=mp.pi / 6, high=mp.pi / 2)
            positive_rate = pf(pr.d, pr.n, pr.k, beta=theta)
    else:
        positive_rate = pf(pr.d, pr.n, pr.k, beta=theta)

    fc, dc, eta = cost(pr, theta)

    return ListDecodingResult(
        d=pr.d,
        n=pr.n,
        k=pr.k,
        theta1=float(theta),
        theta2=float(theta),
        log_cost=float(log2(fc)),
        pf_inv=int(round(1 / positive_rate)),
        eta = eta,
        metric=metric,
        detailed_costs=dc,
    )
```

G.4 utils.py

utils.py contains the functions producing our tables and graphs. First, the relevant probabilities need to be preprcomputed and stored to disk by calling `bulk_create_and_store_bundles`. Then, to estimate costs we call `bulk_cost_estimate`.

```
# -*- coding: utf-8 -*-
"""
Highlevel functions for bulk producing estimates.
"""
from mpmath import mp
from collections import OrderedDict
from probabilities import probabilities, Probabilities
from config import MagicConstants

import os
import csv
try:
    import cPickle as pickle
except ImportError:
    import pickle


class PrecomputationRequired(Exception):
    pass


def pretty_probs(probs, dps=10):
    """
    Take a ``Probabilities`` object and pretty print the estimated probabilities.

    :param probs: a ``Probabilitiess`` object.

    """
    fmt = "{0:7s}: {1:%ds}" % dps
    with mp.workdps(dps):
        print(fmt.format("gr", probs.gr))
        print(fmt.format("ngr", 1 - probs.gr))
        print(fmt.format("pf", probs.pf))
        print(fmt.format("npf", 1 - probs.pf))
        print(fmt.format("gr^pf", probs.gr_pf))
        print(fmt.format("ngr^pf", probs.ngr_pf))
        print(fmt.format("gr|pf", probs.gr_pf / probs.pf))
        print(fmt.format("pf|gr", probs.gr_pf / probs.gr))
        print(fmt.format("ngr|pf", probs.ngr_pf / probs.pf))


def create_bundle(d, n, K=None, BETA=None, prec=None):
    """
    Create a bundle of probabilities.
```

```
    :param d: We consider the sphere `S^{d-1}`.
    :param n: Number of popcount vectors.
    :param K: We consider all `k ∈ K` as popcount thresholds (default `k = 5/16·n`).
    :param BETA: We consider all caps parameterized by `β in BETA` (default: No cap).
    :param prec: We compute with this precision (default: 53).

    """
    bundle = OrderedDict()

    prec = prec if prec else mp.prec
    BETA = BETA if BETA else (None,)
    K = K if K else (int(MagicConstants.k_div_n * n),)

    # if 2 ** mp.floor(mp.log(n, 2)) != n:
    #     raise ValueError("n must be a power of two but got %d" % n)

    for k in K:
        if not 0 <= k <= n:
            raise ValueError("k not in [0, %d]" % (0, n))

    for beta in BETA:
        beta_mpf = mp.mpf(beta) if beta else None
        beta_flt = float(beta) if beta else None
        for k in K:
            bundle[(d, n, k, beta_flt)] = probabilities(d, n, k, beta=beta_mpf, prec=prec)

    return bundle


def bundle_fn(d, n=None):
    if n is None:
        d, n = [keys[:2] for keys in d.keys()][0]
    return os.path.join("probabilities", "%03d_%04d" % (d, n))


def store_bundle(bundle):
    """
    Store a bundle in a flat format for compatibility reasons.

    In particular, mpf values are converted to strings.

    """
    bundle_ = OrderedDict()

    for (d, n, k, beta) in bundle:
        with mp.workprec(bundle[(d, n, k, beta)].prec):
            vals = OrderedDict([(k_, str(v_)) for k_, v_ in bundle[(d, n, k, beta)]._asdict().items()])
        bundle_[(d, n, k, beta)] = vals

    with open(bundle_fn(bundle), "wb") as fh:
        pickle.dump(bundle_, fh)


def load_bundle(d, n, compute=False):
    """
    Load bundle from the flat format and convert into something we can use.

    """
    bundle = OrderedDict()
    try:
        with open(bundle_fn(d, n), "rb") as fh:
            bundle_ = pickle.load(fh)
            for (d, n, k, beta) in bundle_:
                with mp.workprec(int(bundle_[(d, n, k, beta)]["prec"])):
                    d_ = dict()
                    for k_, v_ in bundle_[(d, n, k, beta)].items():
                        if "." in v_:
                            v_ = mp.mpf(v_)
                        elif v_ == "None":
                            v_ = None
                        else:
                            v_ = int(v_)
                        d_[k_] = v_
                    bundle[(d, n, k, beta)] = Probabilities(**d_)
            return bundle
    except IOError:
        if compute:
            return create_bundle(d, n, prec=int(compute))
        else:
            raise PrecomputationRequired("d: {d}, n: {n}".format(d=d, n=n))


def __bulk_create_and_store_bundles(args):
    d, n, BETA, prec = args
    bundle = create_bundle(d, n, BETA=BETA, prec=prec)
    store_bundle(bundle)


def bulk_create_and_store_bundles(
```

```
    D,
    N=(128, 256, 512, 1024, 2048, 4096, 8192),
    BETA=(None, mp.pi / 3 - mp.pi / 10, mp.pi / 3, mp.pi / 3 + mp.pi / 10),
    prec=2 * 53,
    ncores=1,
):
    """
    Precompute a bunch of probabilities.
    """
    from multiprocessing import Pool

    jobs = []
    for d in D:
        for n in N:
            jobs.append((d, n, BETA, prec))

    if ncores > 1:
        return list(Pool(ncores).imap_unordered(__bulk_create_and_store_bundles, jobs))
    else:
        return map(__bulk_create_and_store_bundles, jobs)


def load_probabilities(d, n, k, beta=None, compute=False):
    probs = load_bundle(d, n, compute=compute)[(d, n, k, beta)]
    return probs


def __bulk_cost_estimate(args):
    try:
        f, d, metric, kwds = args
        return f(d, metric=metric, **kwds)
    except Exception as e:
        print("Exception in f: {f}, d: {d}, metric: {metric}".format(f=f, d=d, metric=metric))
        raise e


def bulk_cost_estimate(f, D, metric, filename=None, ncores=1, **kwds):
    """
    Run cost estimates and write to csv file.

    :param f: one of ``all_pairs``, ``random_buckets`` or ``list_decoding`` or an iterable of those
    :param D: an iterable of dimensions to run ``f`` on
    :param metric: a metric from ``Metrics`` or an iterable of such metrics
    :param filename: csv filename to write to (may accept "{metric}" and "{f}" placeholders)
    :param ncores: number of CPU cores to use
    :returns: ``None``, but files are written to disk.

    """
    from cost import LogicalCosts, ClassicalCosts, QuantumMetrics, ClassicalMetrics

    try:
        for f_ in f:
            bulk_cost_estimate(f_, D, metric, ncores=ncores, **kwds)
        return
    except TypeError:
        pass

    if not isinstance(metric, str):
        for metric_ in metric:
            bulk_cost_estimate(f, D, metric_, ncores=ncores, **kwds)
        return

    from multiprocessing import Pool

    jobs = []
    for d in D[::-1]:
        jobs.append((f, d, metric, kwds))

    if ncores > 1:
        r = list(Pool(ncores).imap_unordered(__bulk_cost_estimate, jobs))
    else:
        r = list(map(__bulk_cost_estimate, jobs))

    r = sorted(r)  # relying on "d" being the first entry here

    if filename is None:
        filename = os.path.join("..", "data", "cost-estimate-{f}-{metric}.csv")

    filename = filename.format(f=f.__name__, metric=metric)

    with open(filename, "w") as csvfile:
        csvwriter = csv.writer(csvfile, delimiter=",", quotechar='"', quoting=csv.QUOTE_MINIMAL)

        fields = r[0]._fields[:-1]
        if r[0].metric in QuantumMetrics:
            fields += LogicalCosts._fields[1:]
        elif r[0].metric in ClassicalMetrics:
            fields += ClassicalCosts._fields[1:]
        else:
            raise ValueError("Unknown metric {metric}".format(metric=r[0].metric))
```

```
        csvwriter.writerow(fields)

        for r_ in r:
            csvwriter.writerow(r_[:-1] + r_.detailed_costs[1:])
```

G.5 runall.py

runall.py is the one-stop script to produce all data.

```
# -*- coding: utf-8 -*-
import sys
from utils import bulk_create_and_store_bundles, bulk_cost_estimate
from cost import all_pairs, random_buckets, list_decoding, Metrics
from texconstsf import main

NCORES = int(sys.argv[1])

D = range(64, 256+1, 16)
N = [2**i-1 for i in range(5, 16)]
_ = bulk_create_and_store_bundles(D, N, BETA=[], ncores=NCORES)

D = range(272, 1024+1, 16)
N = [2**i-1 for i in range(5, 14)]
_ = bulk_create_and_store_bundles(D, N, BETA=[], ncores=NCORES)

D = range(64, 1024+1, 16)
SIEVES = [all_pairs, random_buckets, list_decoding]
bulk_cost_estimate(SIEVES, D, metric=Metrics, ncores=NCORES)

main()
```