Finding many Collisions via Reusable Quantum Walks*

Application to Lattice Sieving

Xavier Bonnetain[1], André Chailloux[2], André Schrottenloher[3]**[0000-0002-1329-8630], and Yixin Shen[4][0000-0002-8657-9337]

[1] Université de Lorraine, CNRS, Inria, Nancy, France
[2] Inria, Paris, France
[3] Inria, Univ. Rennes, IRISA, Rennes, France
[4] Royal Holloway, University of London, Egham, UK

Abstract. Given a random function f with domain $[2^n]$ and codomain $[2^m]$, with $m \geq n$, a collision of f is a pair of distinct inputs with the same image. Collision finding is an ubiquitous problem in cryptanalysis, and it has been well studied using both classical and quantum algorithms. Indeed, the quantum query complexity of the problem is well known to be $\Theta(2^{m/3})$, and matching algorithms are known for any value of m.
The situation becomes different when one is looking for *multiple* collision pairs. Here, for 2^k collisions, a query lower bound of $\Theta(2^{(2k+m)/3})$ was shown by Liu and Zhandry (EUROCRYPT 2019). A matching algorithm is known, but only for relatively small values of m, when many collisions exist. In this paper, we improve the algorithms for this problem and, in particular, extend the range of admissible parameters where the lower bound is met.
Our new method relies on a *chained quantum walk* algorithm, which might be of independent interest. It allows to extract multiple solutions of an MNRS-style quantum walk, without having to recompute it entirely: after finding and outputting a solution, the current state is reused as the initial state of another walk.
As an application, we improve the quantum sieving algorithms for the shortest vector problem (SVP), with a complexity of $2^{0.2563d+o(d)}$ instead of the previous $2^{0.2570d+o(d)}$.

Keywords: Quantum algorithms, quantum walks, collision search, lattice sieving

1 Introduction

Quantum walks are a powerful algorithmic tool which has been used to provide state-of-the-art algorithms for various important problems in post-quantum

** Part of this work was done while the author was at CWI, Amsterdam, The Netherlands.

cryptography, such as the shortest vector problem (SVP) via lattice sieving [9], the subset sum problem [5], information set decoding [22], etc.

These applications are all established under a particular quantum walk framework called the MNRS framework [27], and the quantum walks look for marked nodes in a so-called Johnson graph [22] (or a product of Johnson graphs). When walking on this particular graph, the MNRS framework is somewhat rigid. First, it requires to setup the uniform superposition of all nodes along with their attached data structure, then it applies multiple times reflection operators which move this quantum state close to the uniform superposition of all marked nodes.

Due to this rigidity, previously, the best way to find k different marked nodes was to run the whole quantum walk (including the setup) k times. In [9] the authors noticed that a way to output *multiple* solutions instead of a single one with quantum walks would improve the quantum time complexity of their algorithm for solving the SVP.

A natural observation which guides us throughout this paper is that in certain cases, after obtaining the uniform superposition of all marked nodes via the MNRS quantum walk, it is possible to retrieve part of the solution and start another MNRS quantum walk using the remaining part of the quantum state as the new starting state. By doing so, we avoid repeating the setup cost for each new quantum walk, and we now benefit from a trade-off.

In particular, using this observation, we tackle the following problem:

Problem 1 (Multiple collision search). Let $f : \{0,1\}^n \to \{0,1\}^m$, $n \leq m \leq 2n$ be a random function. Let $k \leq 2n - m$. Find 2^k *collision pairs*, that is, pairs of distinct x, y such that $f(x) = f(y)$.

The constraints on the input and output domain are such that a significant $(\Theta\left(2^{2n-m}\right))$ number of collisions pairs exist in the random case. This problem has several applications both in asymmetric and symmetric cryptography. For example, the problem of finding multiple vectors close to a target vector, which appears in lattice sieving (as mentioned above) can be seen as a special case. The limited-birthday problem in symmetric cryptanalysis (e.g., impossible differential attacks and rebound distinguishers [16]) is another example.

Lower Bounds. While quantum query lower bounds for the collision problem (with a single solution) had been known for a longer time, Liu and Zhandry proved more recently in [26] a query lower bound in $\Omega\left(2^{2k/3+m/3}\right)$ to find 2^k solutions, which holds for all values of $m \geq n$.

For relatively small values of k and m (precisely, $k \leq 3n - 2m$, as we explain in Section 6), the BHT collision search algorithm [8] allows to reach this bound. Besides this algorithm, Ambainis' algorithm [2] uses a quantum walk to find one collision in time $\widetilde{\mathcal{O}}\left(2^{m/3}\right)$. However, no matching algorithm was known for other values, neither in time nor in queries.

Contributions. Our main contribution in this paper is a *chained quantum walk* algorithm to solve the multiple collision search problem. We formalize the

intuitive idea that the output state of a quantum walk can be *reused*, to some extent, as the starting state of another. For any admissible values of k, n, m such that $k \leq \frac{m}{4}$, our algorithm requires $\mathcal{O}\left(2^{\frac{2}{3}k+\frac{m}{3}}\right)$ queries, and also $\widetilde{\mathcal{O}}\left(2^{\frac{2}{3}k+\frac{m}{3}}\right)$ quantum gates (i.e., time) and space in the qRAM model.

Theorem 4 (Section 4). Let $f : \{0,1\}^n \to \{0,1\}^m$, $n \leq m \leq 2n$ be a random function. Let $k \leq \min(2n - m, m/4)$. There exists a quantum algorithm making $\mathcal{O}\left(2^{2k/3+m/3}\right)$ quantum queries to f and with a gate count $\widetilde{\mathcal{O}}\left(2^{2k/3+m/3}\right)$, that outputs 2^k collision pairs of f.

By combining our algorithm with the BHT approach, we can now meet the lower bound over all values of k, n, m, except a range of (k, m) contained in $\left[\frac{n}{3}, n\right] \times [n, 1.6n]$, as summarized in Figure 1. Nevertheless, our approach also improves the known complexities in this range.

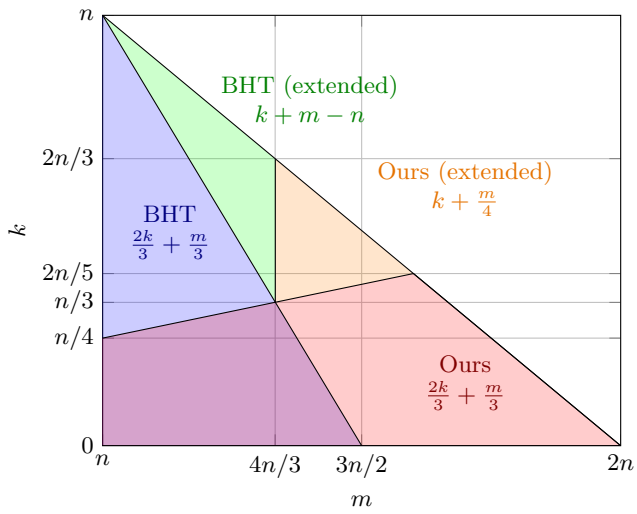

Fig. 1. Gate count exponent in the algorithm depending on the relative values of k, m and n. Both our algorithm and the BHT approach can be extended to the whole triangle, but we show only the one achieving the best complexity. In the purple region (bottom left), both approaches reach the same complexity exponent $\frac{2k}{3} + \frac{m}{3}$.

Using our new algorithm, we improve the state-of-the-art time complexity of quantum sieving to solve the SVP in [9] from $2^{0.2570d+o(d)}$ to $2^{0.2563d+o(d)}$ quantum gates. We also provide time-memory trade-offs that are conjectured to be tight [15]:

Theorem 7 (Section 4). Let $f : \{0,1\}^n \to \{0,1\}^m$, $n \leq m \leq 2n$ be a random function. For all $k \leq \ell \leq \max(2n - m, m/2)$, there ex-

ists an algorithm that computes 2^k collisions using $\widetilde{\mathcal{O}}\left(2^\ell\right)$ qubits and $\widetilde{\mathcal{O}}\left(2^{k+m/2-\ell/2}\right)$ quantum gates and quantum queries to f.

Organization. In Section 2 we provide several technical preliminaries on quantum algorithms, especially Grover's quantum search algorithm. Indeed, an MNRS quantum walk actually emulates a quantum search, and these results are helpful in analyzing the behavior of such a walk. In Section 3, we give important details on the MNRS framework, and in particular, the *vertex-coin encoding*, which is a subtlety often omitted from depictions of the framework in the previous literature. In Section 4 we detail our algorithm assuming a suitable quantum data structure is given, and in Section 5 we detail the *quantum radix trees*. While they were already proposed in [21], we give new (or previously omitted) details relative to the radix tree operations, memory allocation, and how we can efficiently and robustly extract collisions. We give a general summary of the multiple collision search problem in Section 6 and our applications in Section 7.

2 Preliminaries

In this section, we give some preliminaries on collision search, quantum algorithms and Grover search, which are important for the analysis of quantum walks and their data structures.

2.1 Collision Search

In this paper, we study the problem of *collision search* in random functions.

Problem 2. Let $f : \{0,1\}^n \to \{0,1\}^m$ $(n \leq m)$ be a random function. Find a collision of f, that is, a pair $(x, y), x \neq y$ such that $f(x) = f(y)$.

The case $m < n$ can be solved by the same algorithms as the case $m = n$ by reducing f to a subset of its domain. This is why in the following, we focus only on $m \geq n$. The average number of collisions is $\mathcal{O}\left(2^{2n-m}\right)$. When $m \geq 2n$, we can assume that exactly one collision exists, or none. Distinguishing between these two cases is the problem of *element distinctness*, which is solved by searching for the collision. In all cases, the collision problem can be solved in:

- $\Theta(2^{m/2})$ classical time (and queries to f). When $m = n$, the problem is the easiest, as it requires only $\mathcal{O}\left(2^{n/2}\right)$ time and $\mathsf{poly}(n)$ memory using Pollard's rho method. When $m = 2n$, the problem is harder since the best algorithm also uses $\Theta(2^n)$ memory.
- $\Theta(2^{m/3})$ quantum time (and quantum queries to f). A first algorithm was given by Brassard, Høyer and Tapp to reach this for $m = n$ [8], then the lower bound was proven to be $\Omega(2^{m/3})$ [1], and afterwards, Ambainis solved the *element distinctness* problem (the case $m = 2n$) by a quantum walk algorithm [2] which can be adapted for any value of m.

In our case, we want to solve the problem of *multiple collision search*: as there will be expectedly many collisions in the outputs of f, we want to find a significant (exponential in n) number of them.

Problem 3. Let $f : \{0,1\}^n \to \{0,1\}^m$, $n \leq m \leq 2n$, $k \leq 2n-m$. Find 2^k distinct collisions of f.

Here the state of the art differ classically and quantumly:

- Classically, it is well known that the problem can be solved for any m and k in $\Theta(2^{(k+m)/2})$ queries (as long as 2^k does not exceed the average number of collisions of f).
- Quantumly, Liu and Zhandry [26] gave a query lower bound $\Omega(2^{2k/3+m/3})$. However, a matching algorithm is only known for small m. For example, this lower bound is matched for $m = n$ by adapting the BHT algorithm [26,17].

Note that we assume that the collision pairs are fully distinct. In the case $m < n, k \geq m$, there are not enough distinct images, and we only obtain multicollision tuples. The lower bound of [26] does not apply here. If $m < n$ and $k \leq m$, we restrict the inputs of the function to a set of size $\{0,1\}^m$, and this case is covered by a variant of the BHT algorithm. Thus, like in the case of a single collision, we will only consider $n \leq m$.

On the Memory Complexity. For $m = n$, the best known classical algorithm for multiple collision-finding is the parallel collision search (PCS) algorithm by van Oorschot and Wiener [30]. It generalizes Pollard's rho method which finds a single collision in $\mathcal{O}\left(2^{n/2}\right)$ time and $\mathsf{poly}(n)$ memory. Dinur [12] showed that in this regime, the time-space trade-off of the PCS algorithm is optimal. Using a restricted model of computation, it can also be shown optimal for larger values of m.

Quantumly, a time-space lower bound of $T^3S \geq \Omega\left(2^{3k+m}\right)$ has been shown [15]. However, the authors conjecture this bound can be improved to $T^2S \geq \Omega\left(2^{2k+m}\right)$. All known quantum algorithms for collisions, including our new algorithms, match this conjectured lower bound.

2.2 Quantum Algorithms

We refer to [29] for an introduction to quantum computation. We write our quantum algorithms in the standard *quantum circuit model*, where algorithms are written as a sequence of standard *quantum gates*. We are interested in the minimal achievable gate count. This means that we do not consider any parallelization trade-offs, even though there is some literature on the topic for SVP algorithms [24]. By default, we use the (universal) Clifford+T gate set, although our complexity analysis remains asymptotic, and we do not detail our algorithms at the gate level.

Memory Models. Many memory-intensive quantum algorithms require some kind of *quantum random-access model* (qRAM), which can be stronger than the standard quantum circuit model. One can encounter two types of qRAM:

- Classical memory with quantum random access (QRACM): a classical memory of size M can be addressed *in quantum superposition* in $\mathsf{polylog}(M)$ operations.
- Quantum memory with quantum random access (QRAQM): M *qubits* can be addressed *in quantum superposition* in $\mathsf{polylog}(M)$ operations.

The QRAQM model is required by most quantum walk based algorithms for cryptographic problems, e.g., subset-sum [4,5], information set decoding [22] and the most recent quantum algorithm for lattice sieving [9]. It requires to augment the set of gates available with a "qRAM" gate addressing all M memory cells (e.g., individual bits) in superposition. In this paper, we use a definition taken from [2]:

$$|y_1, \ldots, y_M\rangle |x\rangle |i\rangle \xmapsto{\mathsf{qRAM}} |y_1, \ldots, y_{i-1}, x, y_{i+1}, \ldots y_M\rangle |y_i\rangle |i\rangle \quad . \tag{1}$$

This operation implies the ability to *read* in superposition by querying the cell at index i, but also to *write*. This is necessary for efficient data structures such as the ones studied in [2] or the *quantum radix trees* from the literature (see Section 5).

While the qRAM gate can be simulated with $\widetilde{\mathcal{O}}(M)$ Clifford+T gates, in the following, the gate count of our algorithms is given asymptotically on the "Clifford + T + qRAM" gate set, so we assume the qRAM has unit cost, as is required by previous works.

Collision Finding without qRAM. To date, the best quantum algorithms for collision finding, and the ones that reach the query lower bound, require the qRAM model: the BHT algorithm [8] uses QRACM and Ambainis' quantum walk uses QRAQM [2] to define gate-efficient quantum data structures. Initially Ambainis used a *skip list*. We will focus on the more recent *quantum radix tree*, but the QRAQM requirement remains the same.

To some extent, it is possible to get rid of qRAM. For $m = n$, the complexity rises from $\mathcal{O}\left(2^{m/3}\right)$ to $\mathcal{O}\left(2^{2m/5}\right)$ gates [10]. For $m = 2n$, the complexity rises to $\mathcal{O}\left(2^{3m/7}\right)$ [20]. These algorithms can also be adapted for multiple collision finding, where they will outperform the classical ones for some parameter ranges (but not all).

2.3 Grover's Algorithm

In this section, we recall Grover's quantum search algorithm [14] and give a few necessary results for the rest of our analysis. Indeed, as shown in [27], an MNRS quantum walk actually emulates a quantum search, up to some error. If we manage to put this error aside, the analysis of the walk follows from the following lemmas.

Original Quantum Search. In the original setting of Grover's search, we have a function $g : \{0,1\}^n \to \{0,1\}$ and the goal is to find x st. $g(x) = 1$ using queries to g. In the quantum setting, we have access to the unitary $O_g : |x\rangle |b\rangle \to |x\rangle |b \oplus g(x)\rangle$, which is an efficient quantum unitary if g is efficiently computable. In particular we can compute $|\psi_U\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle |g(x)\rangle$ with a single call to O_g. Let $\varepsilon = \frac{|\{x:g(x)=1\}|}{2^n}$. We also define the normalized states

$$|\psi_B\rangle = \frac{1}{\sqrt{(1-\varepsilon)2^n}} \sum_{x:g(x)=0} |x\rangle |g(x)\rangle , \quad |\psi_G\rangle = \frac{1}{\sqrt{\varepsilon 2^n}} \sum_{x:g(x)=1} |x\rangle |g(x)\rangle$$

and $|\psi_U\rangle = \sqrt{1-\varepsilon}\,|\psi_B\rangle + \sqrt{\varepsilon}\,|\psi_G\rangle$. Let $\mathcal{H} = \mathrm{span}(\{|\psi_B\rangle, |\psi_G\rangle\})$. Let Rot_θ be the θ-rotation unitary in $\mathcal{H}$:

$$\mathrm{Rot}_\theta(\cos(\alpha)\,|\phi_B\rangle + \sin(\alpha)\,|\psi_G\rangle) = \cos(\alpha+\theta)\,|\psi_B\rangle + \sin(\alpha+\theta)\,|\psi_G\rangle \quad .$$

For a fixed ε, let $\alpha = \arcsin(\sqrt{\varepsilon})$ so that

$$|\phi_U\rangle = \sqrt{1-\varepsilon}\,|\psi_B\rangle + \sqrt{\varepsilon}\,|\psi_G\rangle = \cos(\alpha)\,|\psi_B\rangle + \sin(\alpha)\,|\psi_G\rangle ,$$

For a state $|\psi\rangle \in \mathcal{H}$, let $\mathrm{Ref}_{|\psi\rangle}$ be the reflection over $|\psi\rangle$ in $\mathcal{H}$:

$$Ref_{|\psi\rangle}\,|\psi\rangle = |\psi\rangle \quad \text{and} \quad Ref_{|\psi\rangle}\left|\psi^\perp\right\rangle = -\left|\psi^\perp\right\rangle$$

where $\left|\psi^\perp\right\rangle$ is any state in $\mathcal{H}$ orthogonal to $|\psi\rangle$[5] We have

$$\mathrm{Ref}_{|\psi_U\rangle}\mathrm{Ref}_{|\psi_B\rangle} = \mathrm{Rot}_{2\alpha} \quad .$$

Assume that we have access to a *checking oracle* O_{check} which performs:

$$\begin{cases} O_{\mathrm{check}}\,|\psi_B\rangle\,|0\rangle & = |\psi_B\rangle\,|0\rangle \\ O_{\mathrm{check}}\,|\psi_G\rangle\,|0\rangle & = |\psi_G\rangle\,|1\rangle \end{cases}$$

In the standard setting described above, this is just copying the last register. Starting from an "initial state" $|\psi_U\rangle$, we apply repeatedly an iterate consisting of a reflection over $|\psi_U\rangle$, and a reflection over $|\psi_B\rangle$. This progressively transforms the current state into the "good state" $|\psi_G\rangle$. Typically $\mathrm{Ref}_{|\psi_U\rangle}$ is constructed from a circuit that computes $|\psi_U\rangle$ and $\mathrm{Ref}_{|\psi_B\rangle}$ is implemented using the checking oracle above: in that case, we are actually performing an *amplitude amplification* [7].

Proposition 1 (Grover's algorithm, known α). *Consider the following algorithm, with $\alpha \le \pi/4$:*

1. *Start from $|\psi_U\rangle$.*
2. *Apply $\mathrm{Rot}_{2\alpha} = \mathrm{Ref}_{|\psi_U\rangle}\mathrm{Ref}_{|\psi_B\rangle}$ N times on $|\psi_U\rangle$ with $N = \lfloor \frac{\pi/2-\alpha}{2\alpha} \rfloor$.*
3. *Apply O_{check} and measure the last qubit.*

[5] For a fixed $|\psi\rangle$, $\left|\psi^\perp\right\rangle$ is actually unique up to a global phase.

This procedure measures 1 *wp. at least* $1-4\alpha^2$ *and the resulting state is* $|\psi_G\rangle$.

Proof. Let us define $\gamma = \alpha + 2N\alpha$. We have

$$(Rot_{2\alpha})^n |\psi_U\rangle = \cos(\alpha+2N\alpha)|\psi_B\rangle + \sin(\alpha+2N\alpha)|\psi_G\rangle = \cos(\gamma)|\psi_B\rangle + \sin(\gamma)|\psi_G\rangle .$$

Notice that we chose N st. $\gamma \leq \frac{\pi}{2} < \gamma + 2\alpha$ so $\frac{\pi}{2} - \gamma \in [0, 2\alpha)$. After applying the checking oracle, we obtain the state

$$\cos(\gamma)|\psi_B\rangle|0\rangle + \sin(\gamma)|\psi_G\rangle|1\rangle .$$

Measuring the last qubit gives outcome 1 with probability $\sin^2(\gamma)$ and the resulting state in the first register is $|\psi_G\rangle$. In order to conclude, we compute

$$\sin^2(\gamma) = \cos^2(\pi/2 - \gamma) \geq \cos^2(2\alpha) \geq 1 - 4\alpha^2. \qquad \square$$

In our algorithms, we will start not from the uniform superposition $|\psi_U\rangle$, but from the *bad subspace* $|\psi_B\rangle$. We show that this makes little difference.

Proposition 2 (Starting from $|\psi_B\rangle$, known α). *Consider the following algorithm, with* $\alpha \leq \pi/4$*:*

1. *Start from* $|\psi_B\rangle$.
2. *Apply* $\mathrm{Rot}_{2\alpha} = \mathrm{Ref}_{|\psi_U\rangle}\mathrm{Ref}_{|\psi_B\rangle}$ N' *times on* $|\psi_B\rangle$ *with* $N' = \lfloor \frac{\pi/2}{2\alpha} \rfloor$.
3. *Apply the checking oracle and measure the last qubit.*

This procedure measures 1 *with probability at least* $1-4\alpha^2$ *and the resulting state is* $|\psi_G\rangle$.

Proof. The proof is essentially the same as the previous one. Let $\gamma' = 2N'\alpha$. We have

$$(Rot_{2\alpha})^{N'} |\psi_B\rangle = \cos(2N'\alpha)|\psi_B\rangle + \sin(2N'\alpha)|\psi_G\rangle = \cos(\gamma')|\psi_B\rangle + \sin(\gamma')|\psi_G\rangle .$$

Notice that we chose N' st. $\gamma' \leq \frac{\pi}{2} < \gamma' + 2\alpha$ so $\frac{\pi}{2} - \gamma' \in [0, 2\alpha)$. After applying the checking oracle, we obtain the state

$$\cos(\gamma')|\psi_B\rangle|0\rangle + \sin(\gamma')|\psi_G\rangle|1\rangle .$$

Measuring the last qubit gives 1 wp. $\sin^2(\gamma')$ and the resulting state in the first register is $|\phi_G\rangle$. In order to conclude, we compute

$$\sin^2(\gamma') = \cos^2(\pi/2 - \gamma') \geq \cos^2(2\alpha) \geq 1 - 4\alpha^2. \qquad \square$$

After applying the check and measuring, if we don't succeed, we obtain the state $|\psi_B\rangle$ again. So we can run the quantum search again.

In Grover's algorithm, we have a procedure to construct $|\psi_U\rangle$ and we use this procedure to initialize the algorithm and to perform the operation $\mathrm{Ref}_{|\psi_U\rangle}$. A quantum walk will have the same general structure as Grover's algorithm, but we will manipulate very large states $|\psi_U\rangle$. Though $|\psi_U\rangle$ is long to construct (the *setup* operation), performing $\mathrm{Ref}_{|\psi_U\rangle}$ will be less costly.

In the MNRS framework, $|\psi_U\rangle$ is chosen as the unique eigenvector of eigenvalue 1 of an operator related to a random walk in a graph. To perform $\mathrm{Ref}_{|\psi_U\rangle}$ efficiently, we use phase estimation on this operator.

3 Quantum Walks for Collision Finding

In this section, we present MNRS quantum walks, which underlie most cryptographic applications of quantum walks to date, and give important details on their actual implementation using a *vertex-coin encoding*.

3.1 Definition and Example

We consider a regular, undirected graph $G = (V, E)$, which in cryptographic applications (e.g., collision search), is usually a Johnson graph (as in this paper) or a product of Johnson graphs (a case detailed e.g. in [22]).

Definition 1 (Johnson graph). *The Johnson graph $J(N, R)$ is a regular, undirected graph whose vertices are the subsets of $[N]$ containing R elements, with an edge between two vertices v and v' iff $|v \cap v'| = R - 1$. In other words, v is adjacent to v' if v' can be obtained from v by removing an element and adding an element from $[N] \backslash v$ in its place.*

In collision search, a vertex in the graph specifies a set of R inputs to the function f under study, where its domain $\{0,1\}^n$ is identified with $[2^n]$. Let $M \subseteq V$ be a set of *marked* vertices, e.g., all the subsets $S \subseteq \{0,1\}^n$ which contain a collision of f: $\exists x, y \in S, x \neq y, f(x) = f(y)$. A classical *random walk* on G finds a marked vertex using Algorithm 1.

Algorithm 1: Classical random walk on G

```
Setup an arbitrary vertex x ∈ V
repeat
    repeat
        Update: move to a random adjacent vertex
    until the current vertex is uniformly random
    Check if the current vertex is marked
until the current vertex is marked
```

The quantum walk is analogous to this process. Let $\varepsilon = \frac{|M|}{|V|}$ be the proportion of marked vertices and δ be the spectral gap of G. Starting from any vertex, after $\mathcal{O}\left(\frac{1}{\delta}\right)$ updates, we sample a vertex of the graph uniformly at random. For a Johnson graph $J(N, R)$, $\delta = \frac{N}{R(N-R)} \simeq \frac{1}{R}$. Let S be the time to **Setup**, U the time to **Update**, C the time to **Check** a given vertex. Then Algorithm 1 finds a marked vertex in time: $\mathcal{O}\left(\mathsf{S} + \frac{1}{\varepsilon}\left(\frac{1}{\delta}\mathsf{U} + \mathsf{C}\right)\right)$. Magniez *et al.* [27] show how to translate this generically in the quantum setting, provided that quantum analogs of these operations (SETUP, UPDATE, CHECK) can be implemented.

Theorem 1 (From [27]). *Assume that quantum algorithms SETUP, UPDATE and CHECK are given. Then there exists a quantum algorithm that finds a*

marked vertex with gate count: $\widetilde{\mathcal{O}}\left(\mathsf{S}+\frac{1}{\sqrt{\varepsilon}}\left(\frac{1}{\sqrt{\delta}}\mathsf{U}+\mathsf{C}\right)\right)$ *instead of* $\mathcal{O}\left(\frac{1}{\sqrt{\varepsilon}}\left(\mathsf{S}+\mathsf{C}\right)\right)$ *with a naive search.*

Using this framework generically, we can recover the complexity of Ambainis' algorithm for collision search: $\widetilde{\mathcal{O}}\left(2^{m/3}\right)$ for any codomain bit-size m. We use the Johnson graph $J(2^n, 2^{m/3})$. Its spectral gap is approximately $2^{-m/3}$. A vertex is marked if and only if it contains a collision, so the probability of being marked is approximately $2^{2m/3-m} = 2^{-m/3}$. Using a quantum data structure for unordered sets, we can implement SETUP in gate count $\widetilde{\mathcal{O}}\left(2^{m/3}\right)$, UPDATE and CHECK in $\mathsf{poly}(n)$. The formula of Theorem 1 gives the complexity $\widetilde{\mathcal{O}}\left(2^{m/3}\right)$.

3.2 Details of the MNRS Framework

In the d-regular graph $G = (V, E)$, for each $x \in V$, let N_x be the set of neighbors of x, of size d. In the case $G = J(N, R)$, we have $d = R(N - R)$. For a vertex x, let $|x\rangle$ be an arbitrary encoding of x as a quantum state, let $D(x)$ be a *data structure* associated to x, and let $|\widehat{x}\rangle = |x\rangle\,|D(x)\rangle$.

Remark 1. The encoding of x is commonly thought of as the set itself, and the data structure as the images of the set by f. But whenever we look at quantum walks from the perspective of gate count (and not query complexity), an efficient quantum data structure is already required for x itself, i.e., an unordered set data structure in the case of a Johnson graph, and one cannot really separate x from $D(x)$. This is why we will favor the notation $|\widehat{x}\rangle$.

For a vertex x, let $|p_x\rangle$ be the uniform superposition over its neighbors: $|p_x\rangle = \frac{1}{\sqrt{d}}\sum_{y\in N_x}|y\rangle$, and: $|\widehat{p_x}\rangle = \frac{1}{\sqrt{d}}\sum_{y\in N_x}|\widehat{y}\rangle$. From now on, we consider a walk on *edges* rather than vertices in the graph, and introduce:

$$\begin{cases} |\psi_U\rangle = \frac{1}{\sqrt{|V|}}\sum_{x\in V}|\widehat{x}\rangle\,|p_x\rangle \text{ the superposition of vertices (and neighbors)} \\ |\psi_M\rangle = \frac{1}{\sqrt{|M|}}\sum_{x\in M}|\widehat{x}\rangle\,|p_x\rangle \text{ the superposition of marked vertices} \\ A = \mathrm{span}\{|\widehat{x}\rangle\,|p_x\rangle\}_{x\in V} \\ B = \mathrm{span}\{|\widehat{p_y}\rangle\,|y\rangle\}_{y\in V} \end{cases}$$

Let Ref_A and Ref_B be respectively the reflection over the space A and the space B. The core of the MNRS framework is to use these operations to emulate a reflection over $|\psi_U\rangle$. By alternating such reflections with reflections over $|\psi_M\rangle$ (using the checking procedure), the quantum walk behaves exactly as a quantum search, and the analysis of Section 2.3 applies.

Proposition 3 (From [27]). *Let $W = \mathrm{Ref}_B\mathrm{Ref}_A$. We have $\langle\psi_U|\,W\,|\psi_U\rangle = 1$. For any other eigenvector $|\psi\rangle$ of W, we have $\langle\psi|\,W\,|\psi\rangle = e^{i\theta}$ with $\theta \in [2\sqrt{\delta}, \pi/2]$.*

To reflect over $|\psi_U\rangle$, we perform a *phase estimation* of the unitary W, which allows to separate the part with eigenvalue 1, from the part with eigenvalue $e^{i\theta}$

with $\theta \in [2\sqrt{\delta}, \pi/2]$. The phase estimation circuit needs to call W a total of $\mathcal{O}\left(\frac{1}{\sqrt{\delta}}\right)$ times to estimate θ up to sufficient precision. It has some error, which can be made insignificant with a polynomial increase in complexity; thus in the following, we will consider the reflection Ref_U to be exact.

To construct W, we need to implement Ref_A and Ref_B. We first remark that:

$$\mathrm{Ref}_B = \mathrm{SWUP} \circ \mathrm{Ref}_A \circ \mathrm{SWUP} \ , \tag{2}$$

where $\mathrm{SWUP} \left|\widehat{x}\right\rangle \left|y\right\rangle = \left|\widehat{y}\right\rangle \left|x\right\rangle$. This SWUP (Swap-Update) operation can furthermore be decomposed into an update of the database (UP_D) followed by a register swap:

$$\left|\widehat{x}\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|D(x)\right\rangle \left|y\right\rangle \xrightarrow{\mathrm{UP}_D} \left|x\right\rangle \left|D(y)\right\rangle \left|y\right\rangle \xrightarrow{\mathrm{Swap}} \left|y\right\rangle \left|D(y)\right\rangle \left|x\right\rangle = \left|\widehat{y}\right\rangle \left|x\right\rangle \ , \tag{3}$$

so $\mathrm{SWUP} = \mathrm{Swap} \circ \mathrm{UP}_D$.

We would then implement Ref_A using an update unitary that, from a vertex x, constructs the uniform superposition of neighbors. However this would require us to write $\log_2(|V|)$ data, and in practice, $|V|$ is doubly exponential (the vertex is represented by an exponential number of bits). Thankfully, in d-regular graphs, and in particular in Johnson graphs, we can avoid this loophole by making the encoding of edges more compact. Instead of storing a pair of vertices (x, y), which will eventually result in having to rewrite entire vertices, we can store a single vertex and a *direction*, or *coin*.

3.3 Vertex-coin Encoding

The encoding is a reversible operation: $O_{\mathrm{Enc}} \left|\widehat{x}\right\rangle \left|y\right\rangle = \left|\widehat{x}\right\rangle \left|c_{x\to y}\right\rangle$, which compresses an edge (x, y) by replacing y by a much smaller register of size $\lceil \log_2(d) \rceil$. Note that we only need the *existence* of such a circuit. We never use it during the algorithms; all operations are directly performed using the compact encoding.

Let $\left|\psi^{coin}_{\mathrm{Unif}}\right\rangle = \frac{1}{\sqrt{d}} \sum_c \left|c\right\rangle$ be the uniform superposition of coins. In the vertex-coin encoding, Ref_A corresponds to $I \otimes Ref_{\left|\psi^{coin}_{\mathrm{Unif}}\right\rangle}$:

$$\mathrm{Ref}_A = O^{-1}_{\mathrm{Enc}} \circ \left(I \otimes \mathrm{Ref}_{\left|\psi^{coin}_{\mathrm{Unif}}\right\rangle}\right) \circ O_{\mathrm{Enc}}.$$

Now, for the SWUP operation, we have to decompose again UP_D and Swap in the encoded space. First, we define UP'_D such that:

$$\left|x\right\rangle \left|D(x)\right\rangle \left|c_{x\to y}\right\rangle \xrightarrow{UP'_D} \left|x\right\rangle \left|D(y)\right\rangle \left|c_{x\to y}\right\rangle .$$

Moreover, we define Swap' such that:

$$\left|x\right\rangle \left|c_{x\to y}\right\rangle \xrightarrow{\mathrm{Swap}'} \left|y\right\rangle \left|c_{y\to x}\right\rangle .$$

and we define $\mathrm{SWUP}' = \mathrm{Swap}' \circ \mathrm{UP}'_D$ (we abuse notation here, by extending Swap' where we apply the identity to the middle register), so:

$$\mathrm{SWUP}' \left|\widehat{x}\right\rangle \left|c_{x\to y}\right\rangle = \left|\widehat{y}\right\rangle \left|c_{y\to x}\right\rangle \ ,$$

and $\mathrm{SWUP}' = O_{\mathrm{Enc}} \circ \mathrm{SWUP} \circ O_{\mathrm{Enc}}^{-1}$. So we define

$$\begin{cases} \mathrm{Ref}'_A = I \otimes \mathrm{Ref}_{\left|\psi_{\mathrm{Unif}}^{coin}\right\rangle} = O_{\mathrm{Enc}} \circ Ref_A \circ O_{\mathrm{Enc}}^{-1} \\ \mathrm{Ref}'_B = \mathrm{SWUP}' \circ \mathrm{Ref}'_A \circ \mathrm{SWUP}' = O_{\mathrm{Enc}} \circ \mathrm{Ref}_B \circ O_{\mathrm{Enc}}^{-1} \\ W' = \mathrm{Ref}'_B \circ \mathrm{Ref}'_A \end{cases} \tag{4}$$

By putting everything together, we have $W' = O_{\mathrm{Enc}} \circ W \circ O_{\mathrm{Enc}}^{-1}$. Since O_{Enc} is a unitary operator, W and W' are unitarily equivalent, i.e., they have the same eigenvalues. Thus, Proposition 3 applies to W' the same as it does to W, and gives its spectral properties. We can perform phase estimation on W', and combine afterwards with Proposition 1. Since constructing the uniform superposition of coins is trivial, all relies on the unitary SWUP'.

Theorem 2 (MNRS, adapted). *Let $|\widehat{x}\rangle$ be an encoding of the vertex x (incl. data structure) and assume that a vertex-coin encoding is given. Let $\alpha = \arcsin\sqrt{\varepsilon}$. Starting from the state: $\frac{1}{\sqrt{|V|}}\sum_{x\in V}|\widehat{x}\rangle\left|\psi_{\mathrm{Unif}}^{coin}\right\rangle$, apply $\left\lfloor\frac{\pi/2-\alpha}{2\alpha}\right\rfloor$ iterates of: • a checking procedure which flips the phase of marked vertices; • a phase estimation of W'; then apply the checking again and measure. With probability at least $1-4\alpha^2$, we measure 1 and collapse on the uniform superposition of marked vertices.*

Finally, we can adapt this analysis by starting from the bad vertices, with a proof that is the same as Proposition 2. This will be the main building block of our new algorithm.

Theorem 3 (MNRS, starting from bad vertices). *Starting from the state: $\frac{1}{\sqrt{|V|-|M|}}\sum_{x\in V\setminus M}|\widehat{x}\rangle\left|\psi_{\mathrm{Unif}}^{coin}\right\rangle$ (the superposition of unmarked vertices), apply $\left\lfloor\frac{\pi/2}{2\alpha}\right\rfloor$ iterates of: • a checking procedure which flips the phase of marked vertices; • a phase estimation of W'; then apply the checking again and measure. With probability at least $1-4\alpha^2$, we measure 1 and collapse on the uniform superposition of marked vertices. Otherwise, we collapse on the uniform superposition of unmarked vertices.*

Coins for a Johnson Graph. In a Johnson graph $J(N,R)$, a coin $c=(j,z)$ is a pair where:

- $j \in [R]$ is the index of the element that will be removed from the current vertex (given an arbitrary ordering, e.g. the lexicographic ordering of bitstrings).
- $z \in [N-R]$ is the index of an element that does not belong to the current vertex, and will be added as a replacement.

4 A Chained Quantum Walk to Find Many Collisions

In this section, we prove our main result.

Theorem 4. *Let $f : \{0,1\}^n \to \{0,1\}^m$, $n \leq m \leq 2n$ be a random function. Let $k \leq \min(2n-m, m/4)$. There exists a quantum algorithm making $\mathcal{O}\left(2^{2k/3+m/3}\right)$ quantum queries to f and using $\widetilde{\mathcal{O}}\left(2^{2k/3+m/3}\right)$ Clifford+T+qRAM gates, that outputs 2^k collision pairs of f.*

Our new algorithm, which is detailed in Section 4.1 and Section 4.2, solves the case $k \leq \frac{m}{4}$. The case $k \leq 2n - m$ was already solved by adapting the BHT algorithm, as detailed in Section 6.

Note that if we are only interested in the query complexity, our technique is still necessary to improve over previous results, but the radix tree data structure that we detail in Section 5 can be replaced by a simple ordered list with expensive update operations (see [19]).

4.1 New Algorithm

We detail here our *chained quantum walk* algorithm. Recall that the Johnson graph $J(N, R)$ is the regular, undirected graph whose vertices are subsets of size R of $[N]$, and edges connect each pair of vertices which differ in exactly one element. We identify $[N]$ with $\{0,1\}^n$, the domain of f.

We assume that an efficient quantum unordered set data structure is given, which makes vertices in the Johnson graph correspond to quantum states, while allowing to implement efficiently the operations required for the MNRS quantum walks. It will be detailed in Section 5. In the following we write $|S\rangle$ for the quantum state corresponding to a set S.

Idea of Our Algorithm. After running a quantum walk on a Johnson graph, we obtain a superposition of vertices which contain a collision. We remove the collision from the vertex, and we measure the elements that form this collision: we still obtain a superposition of sets, which might be exploited for the next walk. The sets in this superposition have a very important property: because we just removed the collision (more generally, we will remove all collisions that the vertex contains), they actually *do not* contain one with certainty. Thus, we do not have the uniform superposition of vertices of our next MNRS walk, but the uniform superposition of *unmarked* vertices. However, we have seen that this made little difference, and we can continue using Theorem 3. When we measure the result of a walk step, it will succeed with at least constant probability. In the case of failure, we collapse on the superposition of unmarked vertices again, which means we simply have to restart the walk. The extraction of collisions modifies the walk parameters (vertex size, graph, marked vertices) in a way that we track throughout the algorithm, and is detailed below.

Technical Details. Let C be a table in classical memory of all the multi-collisions found so far. This table contains entries of the form: $u : (x_1, \ldots, x_r)$ where $f(x_1) = \ldots = f(x_r) = u$ forms a multicollision of f, indexed by the image. We

define the *size* of C, its set of *preimages* and its set of *images*:

$$\begin{cases} \mathsf{Preim}(C) := \bigcup_{u:(x_1,\ldots,x_r)\in C} \{x_1,\ldots,x_r\} \\ \mathsf{Im}(C) := \bigcup_{u:(x_1,\ldots,x_r)\in C} \{u\} \end{cases} \tag{5}$$

Given the table C, given a size parameter R, we define the two sets of sets:

$$\begin{cases} V_R^C & := \{S \subseteq (\{0,1\}^n \backslash \mathsf{Preim}(C)), |S| = R\} \\ M_R^C & := \{S \subseteq (\{0,1\}^n \backslash \mathsf{Preim}(C)), |S| = R, \\ & \qquad (\exists x \neq y \in S, f(x) = f(y) \vee \exists z \in S, f(z) \in \mathsf{Im}(C))\} \end{cases} \tag{6}$$

The first one will be the set of vertices for the current walk, and the second one its set of *marked* vertices. As we can see, the current walk excludes a set of previously measured inputs, and a vertex is marked if it leads to a new collision, *or* to a preimage of one of the previously measured images. The second case simply extends one of the currently known multicollision tuples. The probability for a vertex to be marked can be easily computed, and we just need to bound it as follows:

$$\max\left(\frac{R|\mathsf{Im}(C)|}{2^m}, \frac{R(R-1)}{2^{m+1}}\right) \leq \varepsilon_{R,C} \leq \frac{R|\mathsf{Im}(C)|}{2^m} + \frac{R(R-1)}{2^{m+1}} \ ,$$

since any vertex containing a collision, or a preimage from the table C, is marked.

In Section 5, we will show that with an appropriate data structure, there exists an *extraction* algorithm EXTRACT which does the following:

$$\text{EXTRACT} : C, R, \frac{1}{\sqrt{|M_R^C|}} \sum_{S\in M_R^C} |S\rangle \mapsto C', R', \frac{1}{\sqrt{|V_{R'}^{C'} \backslash M_{R'}^{C'}|}} \sum_{S\in V_{R'}^{C'}\backslash M_{R'}^{C'}} |S\rangle \ ,$$

where $R' = R - r$ for some value r, and C' contains exactly r new elements (collisions adding new entries, or preimages going into previous entries). Thus, EXTRACT transforms the output of a successful walk into the set of *unmarked vertices* for the next walk.

We can now give Algorithm 2, depending on a tunable parameter ℓ.

4.2 Complexity Analysis

Theorem 5 (Time-memory tradeoff). *For all $k \leq \ell \leq \min(2k/3+m/3, m/2)$, Algorithm 2 computes 2^k collisions using $\widetilde{\mathcal{O}}\left(2^\ell\right)$ qubits and $\widetilde{\mathcal{O}}\left(2^{k+m/2-\ell/2}\right)$ Clifford+T+qRAM gates.*

Proof. We start by noticing that although Algorithm 2 outputs a set of multicollisions rather than collisions, the number of collisions and multicollisions that are actually obtained are closely related. Indeed, for a function from $[2^n]$ to $[2^n]$, there is a polynomial (in n) limit to the width of multicollisions that can appear for a non-negligible fraction of the functions. Indeed, by Theorem 4

Algorithm 2: Chained quantum walk algorithm for multiple collisions.

Input: quantum access to $f : \{0,1\}^n \to \{0,1\}^m$, parameter k
Output: a table of multicollisions C such that $|\mathsf{Im}(C)| \geq 2^k$
$C \leftarrow \emptyset$, $R \leftarrow 2^\ell$ `/* Initialize an empty table */`
$|\psi\rangle \leftarrow \sum_{S \in V_{2^\ell}^C} |S\rangle$ `/* SETUP */`

while $|\mathsf{Im}(C)| < 2^k$ **do**
  Run the quantum walk:
  - Starting state: $|\psi\rangle = \sum_{S \in V_R^C \setminus M_R^C} |S\rangle$
  - Graph: $J(\{0,1\}^n \setminus \mathsf{Preim}(C), R)$ (Johnson graph with vertices of size R, excluding the preimages of C)
  - Marked vertices: M_R^C
  - Iterates: $\lfloor (\pi/2)/(2\alpha) \rfloor$, where $\alpha = \arcsin \sqrt{\varepsilon_{R,C}}$
  - Spectral gap: $\delta \simeq \frac{1}{R}$

  Apply CHECK and measure the result: let flag be the output
  if *flag is true* **then**
    `/* The state collapses on:` $\sum_{S \in M_R^C} |S\rangle$ `*/`
    Apply EXTRACT (contains measurements)
    - Update the table C
    - Update the current width R
    - Update the state: $|\psi\rangle = \sum_{S \in V_R^C \setminus M_R^C} |S\rangle$

  `/* Otherwise, the state collapses on:` $\sum_{S \in V_R^C \setminus M_R^C} |S\rangle$ `for the previous` R `and` C`. There is nothing to extract from it,` C `and` R `remain unchanged. */`
return C

in [13], the average number of r-collisions in such a random function is $\frac{2^n e^{-1}}{r!}$. Thus, there exists a universal constant c such that with probability $1 - o(2^{-n})$, such a random function does not have any r-collision with $r \geq cn$.

This means that regardless of the state of the current table C, we have:

$$|\mathsf{Im}(C)| \leq |\mathsf{Preim}(C)| \leq cn|\mathsf{Im}(C)| \ .$$

In particular, by taking 2^ℓ greater than $cn2^{k+1}$, we ensure that during the algorithm, $R > 2^{\ell-1}$. This means that we never run out of elements.

Secondly, we can bound $\varepsilon_{R,C} \geq \frac{R(R-1)}{2^{m+1}}$. This allows to upper bound easily the time complexity of any of the walks: if the current vertex size is R then it runs for $\mathcal{O}\left(2^{m/2}/R\right)$ iterates, and each iterate contains $\widetilde{\mathcal{O}}\left(\sqrt{R}\right)$ operations. The constants in the $\mathcal{O}$ are the same throughout the algorithm. This means that we can upper bound the complexity of each walk by $\widetilde{\mathcal{O}}\left(2^{m/2}/\sqrt{R}\right) \leq \widetilde{\mathcal{O}}\left(2^{m/2-\ell/2}\right)$.

By Theorem 3, the success probability of this walk is bigger than $1 - 4\varepsilon_{R,C}$. If we do not succeed, the CHECK followed by a measurement make the current state collapse again on the superposition of unmarked vertices, and we run the

exact same walk again. Note that for this algorithm to work, we must have $\varepsilon_{R,C} < 0.5$. This corresponds to the probability that the list contains a collision, or a new preimage of $\mathsf{Im}(C)$, which is $\widetilde{\mathcal{O}}\left(2^{2\ell-m}\right)$. Hence, we must have $\ell \leq m/2$.

Then, as $\ell \leq 2k/3 + m/3$, the final complexity of the algorithm is

$$\widetilde{\mathcal{O}}\left(2^{\ell} + 2^{k} 2^{m/2-\ell/2}\right) = \widetilde{\mathcal{O}}\left(2^{k+m/2-\ell/2}\right) \ ,$$

where 2^{ℓ} is the cost of the SETUP, and the second term accounts for all the walk steps. □

5 Quantum Radix Trees and Extractions

In this section, we detail the *quantum radix tree* data structure, a history-independent unordered set data structure introduced in [21]. We show that it allows to perform, exactly and in a polynomial number of Clifford+T+qRAM gates, the two main operations required for our walk: SWUP′ and EXTRACT. We describe these operations in pseudocode, while ensuring that they are reversible and polynomial.

5.1 Logical Level

Following [21], the *quantum radix tree* is an implementation of a radix tree storing an unordered set S of n-bit strings. It has one additional property: its concrete memory layout is history-independent. Indeed, there are many ways to encode a radix tree in memory, and as elements are inserted and removed, we cannot have a unique bit-string $T(S)$ representing a set S. We use instead a uniform superposition of all memory layouts of the tree, which makes the *quantum state* $|T(S)\rangle$ unique, and independent of the order in which the elements were inserted or removed. Only the entry point (the root) has a fixed position.

We separate the encoding of S into $|T(S)\rangle$ in two levels: first, a *logical level*, in which S is encoded as a unique radix tree $R(S)$; second, a *physical level*, in which $R(S)$ is encoded into a quantum state $|T(S)\rangle$. The logical mapping $S \to R(S)$ is standard.

Definition 2 (From [21]). *Let S be a set of n-bit strings. The radix tree $R(S)$ is a binary tree in which each leaf is labeled with an element of S, and each edge with a substring, so that the concatenation of all substrings on the path from the root to the leaf yields the corresponding element. Furthermore, the labels of two children of any non-leaf node start with different bits.*

By convention, we put the "0" bit on the left, and "1" on the right. In addition to the n-bit strings stored by the tree, we append to each node the value of an ℓ-bit *invariant* which can be computed from its children, and depends only on the logical structure of the radix tree, not its physical structure. Typically the invariant can count the number of elements in the tree.

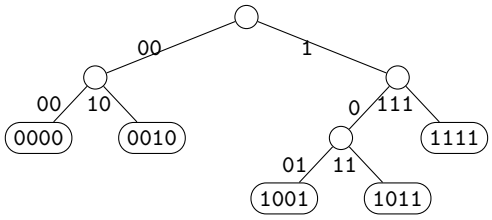

Fig. 2. Tree $R(S)$ representing the set $S = \{0000, 0010, 1001, 1011, 1111\}$ (the example is taken from [21]).

5.2 Memory Representation

We now detail the correspondence from $R(S)$ to $|T(S)\rangle$. We suppose that a quantum *bit-string data structure* is given, that handles bit-strings of length between 0 and n and performs operations such as concatenation, computing shared prefixes, testing if the bit-string has a given prefix, in time $\mathsf{poly}(n)$.

State of the Memory. We suppose that $\mathcal{O}(Mn)$ qubits of memory are given, where $M \geq R$ will be set later on. We divide these qubits into M *cells* of $\mathcal{O}(n)$ qubits each, which we index from 0 to $M-1$. We encode cell addresses on $m = \lceil \log_2 M \rceil + 1$ bits, and we also define an "empty" address $\perp$. Each cell will be either empty, or contain a node of the radix tree, encoded as a tuple $(i, a_l, a_r, \ell_l, \ell_r)$ where:

- i is the value of the invariant
- a_l and a_r (m-bit strings) are respectively the memory addresses of the cells holding the left and right children, either valid indices or $\perp$. A node with $a_l = a_r = \perp$ is a leaf.
- ℓ_l and ℓ_r are the labels of the left and right edges. (ε if the node is a leaf, where ε is the empty string).

In other words, we have added to the tree $R(S)$ a choice of memory locations for the nodes, which we name informally the *memory layout* of the tree. The structure of $R(S)$ itself remains independent on its memory layout.

The root of the tree is stored in cell number 0. In Figure 3, we give an example of a memory representation of the tree $R(S)$ of Figure 2. We take as invariant the number of leaves which, at the root, gives the number of elements in the set. It is important to note that memory cells have an "empty" default state, which allows the radix tree to support size changes. Whether a cell is empty or not depends on the memory layout.

A radix tree encoding a set of size R contains $2R-1$ nodes (including the root), which means that we need (a priori) no more than $M = 2R-1$ cells in our memory. In addition to the bit-strings x, we could add any data d_x to which x serves as a unique index. (This means adding another register which is non-empty for leaf nodes only). Finally, it is possible to account for multiplicity of elements in the tree by adding multiplicity counters, but since this is unnecessary for our applications, we will stick to the case of unique indices.

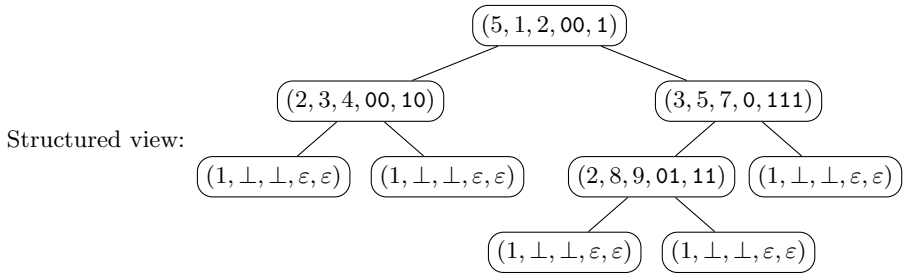

Fig. 3. Example of memory layout for the tree of Figure 2, holding the set $S = \{\texttt{0000}, \texttt{0010}, \texttt{1001}, \texttt{1011}, \texttt{1111}\}$.

Definition. Let S be a set of size R, encoded in a radix tree with $2R-1$ nodes. We can always take an arbitrary ordering of the nodes in the tree, for example the lexicographic ordering of the paths to the root (left = 0, right = 1). This means that, for any sequence of non-repeating cell addresses I, of length $2R-1$, we can define a mapping: $S, I \mapsto T_I(S)$ which specifies the writing of the tree in memory, by choosing the addresses $I = (i_1 = 0, \ldots, i_{2R-1})$ for the elements. For example, the tree of Figure 3 would correspond to the sequence $(0, 1, 3, 4, 2, 5, 8, 9, 7)$. We can then define the *quantum radix tree encoding* S as the quantum state:

$$|T(S)\rangle = \sum_{\text{valid sequences } I} |T_I(S)\rangle \quad , \tag{7}$$

where we take a uniform superposition over all valid memory layouts.

For two different sets S and S', and for any pair I, I' (even if $I' = I$), we have $T_{I'}(S) \neq T_I(S')$: the encodings always differ. This means that, as expected, we have $\langle T(S)|T(S')\rangle = 0$.

Memory Allocator. In order to maintain this uniform superposition over all possible memory layouts, we need an implementation of a *memory allocator*. This unitary ALLOC takes as input the current state of the memory, and returns a uniform superposition over the indices of all currently unoccupied cells. Possible implementations of ALLOC are detailed in Section 5.4.

5.3 Basic Operations

We show how to operate on the quantum radix trees in $\mathsf{poly}(n)$ Clifford+T +qRAM gates. We start with the basics: lookup, insertion and deletion.

Lookup. We define a unitary LOOKUP which, given S and a new element x, returns whether x belongs to S:

$$\text{LOOKUP} \ : |x\rangle\,|T(S)\rangle\,|0\rangle \mapsto |x\rangle\,|T(S)\rangle\,|x \in S\rangle \quad . \tag{8}$$

Algorithm 3: LOOKUP as a classical algorithm.

Input: element x, quantum radix tree $T(S)$
Output: whether $x \in S$
$(i, a_l, a_r, \ell_l, \ell_r) \leftarrow$ root
$y \leftarrow \varepsilon$ (empty string)
while $a_l \neq \perp$ *(node is not a leaf)* **do**
  if $y||\ell_l$ *is a prefix of* x **then**
    $y \leftarrow y||a_l$
    $(i, a_l, a_r, \ell_l, \ell_r) \leftarrow$ node at address a_l
  else if $y||\ell_r$ *is a prefix of* x **then**
    $y \leftarrow y||a_l$
    $(i, a_l, a_r, \ell_l, \ell_r) \leftarrow$ node at address a_r
  else
    Break (not a solution)
return *true if* $y = x$

We implement LOOKUP by descending in the radix tree $R(S)$; the pseudocode is given in Algorithm 3. Since the "while" loop contains at most n iterates, quantumly these n iterates are performed controlled on a flag that says whether the computation already ended. After obtaining the result, they are recomputed to erase the intermediate registers.

Insertion. We define a unitary INSERT, which, given a new element x, inserts x in the set S. If x already belongs to S, its behavior is unspecified.

$$\text{INSERT} \ : |x\rangle\,|T(S)\rangle \mapsto |x\rangle\,|T(S \cup \{x\})\rangle \quad . \tag{9}$$

The implementation of INSERT is more complex, but the operation is still reversible. The pseudocode is given in Algorithm 4. At first, we find the point of insertion in the tree, then we call ALLOC twice to obtain new memory addresses for two new nodes. We modify locally the layout to insert these new nodes, including a new leaf for the new element x. Then, we update the invariant on the path to the new leaf. Finally, we uncompute the path to the new leaf (all the addresses of the nodes on this path). To do so, we perform a loop similar to LOOKUP, given the knowledge of the newly inserted element x.

Deletion. The deletion can be implemented by uncomputing INSERT, since it is a reversible operation. It performs:

$$\text{INSERT}^\dagger \ : |x\rangle\,|T(S \cup \{x\})\rangle \mapsto |x\rangle\,|T(S)\rangle \quad . \tag{10}$$

The deletion of an element that is not in S is unspecified.

Quantum Lookup. We can implement a "quantum lookup" unitary QLOOKUP which produces a uniform superposition of elements in S having a specific property P. The only requirement is that the invariant of nodes has to store the

Algorithm 4: INSERT as a classical algorithm.

Input: element x, quantum radix tree $T(S)$
Output: element x, quantum radix tree $T(S \cup \{x\})$
Find the first node $j_1 : (i, a_l, a_r, \ell_l, \ell_r)$ such that y is a prefix of x, $y||\ell_l$ is not a prefix of x and $y||\ell_r$ is not a prefix of x either. Write all the addresses of the nodes on the path from the root to j_1
/* If at this point we have found that the element belongs to S instead, then the rest of the computation is meaningless. */
/* By construction ℓ_l starts with 0 and ℓ_r starts with 1. One of them shares a non-empty prefix z with the remaining part of x. Without loss of generality, we assume that it is ℓ_l. */
Let $\ell_l = z||t$ and $x = y||z||x'$
Call ALLOC to obtain an address j_2
Replace a_l with j_2 in the node $j_1 : (i, a_l, a_r, \ell_l, \ell_r)$ (move a_l to a temporary register)
Call ALLOC to obtain an address j_3
Write at address j_3: $(*, \perp, \perp, \varepsilon, \varepsilon)$
/* Information at this point: x, a_l, j_2, j_3, the path to j_1 and the tree */
if *t starts with 0* **then**
    Move a_l and cut ℓ_l to modify the two nodes in positions j_1 and j_2 as follows: $j_1 : (i, j_2, a_r, z, \ell_r)$ and $j_2 : (*, a_l, j_3, t, x')$.
else
    Move a_l and cut ℓ_l to modify the two nodes in positions j_1 and j_2 as follows: $j_1 : (i, j_2, a_r, z, \ell_r)$ and $j_2 : (*, j_3, a_l, t, x')$.
/* We make this choice so that the left edge is always labeled starting with a 0 and the right edge with a 1 */
/* Since we have moved j_3 and a_l, the remaining information is: x, the modified tree, j_2 and the path to j_1 (actually the path to x in the new tree) */
Recompute the invariants on the path to x, in reverse order (starting from the address j_2).
/* The recomputation of the invariants is reversible (but we still know the path to x) */
Do a lookup of x to uncompute the path to x.
/* Now the only information that remains is $x, T(S \cup \{x\})$. */

number of nodes in the subtree having this property (and so, leaf nodes will indicate if the given x satisfies $P(x)$ or not).

$$\text{QLOOKUP} \ : |T(S)\rangle\,|0\rangle \mapsto |T(S)\rangle \sum_{x\in S|P(x)} |x\rangle \ . \tag{11}$$

This unitary is implemented by descending in the tree coherently (i.e., in superposition over the left and right paths) with a weight that depends on the number of solutions in the left and right subtrees. First, we initialize an address register $|a\rangle$ to the root. Then, for n times (the maximal depth of the tree), we update the current address register as follows:

- We count the number of solutions in the left and right subtrees of the node at address $|a\rangle$ (say, t_l and t_r).
- We map $|a\rangle$ to $|a\rangle \left(\sqrt{\frac{t_l}{t_l+t_r}}\,|\text{left child of } a\rangle + \sqrt{\frac{t_r}{t_l+t_r}}\,|\text{right child of } a\rangle\right)$. (We do nothing if $|a\rangle$ is a leaf).

In the end, we obtain a uniform superposition of the paths to all elements satisfying P. We can query these elements, then uncompute the paths using an inverse LOOKUP. Likewise, we can also perform a quantum lookup of pairs satisfying a given property, e.g., retrieve a uniform superposition of all collision pairs in S.

5.4 Quantum Memory Allocators

We now define the unitary ALLOC, which given the current state of the memory, creates the uniform superposition of unallocated cells:

$$\text{ALLOC} \ : |\text{current memory}\rangle\,|0\rangle \mapsto |\text{current memory}\rangle \sum_{i \text{ unoccupied}} |i\rangle \ . \tag{12}$$

We do not need to define a different unitary for un-allocation; we only have to recompute ALLOC to erase the addresses of cells that we are currently cleaning. To implement ALLOC, we add to each memory cell a flag indicating if it is allocated. We propose two approaches.

Quantum search allocation. Classically, we can allocate new cells by simply choosing addresses at random and checking if they are already allocated or not. Quantumly, we can follow this approach using a *quantum search* over all the cells for unallocated ones. Obviously, for this approach to be efficient, we need the proportion of unallocated cells to be always constant. Besides, if we keep a counter of the number of allocated cells (which does not vary during our quantum walk steps anyway), we can make this operation exact using Amplitude Amplification (Theorem 4 in [7]). Indeed, this counter gives the proportion of allocated cells, so we know exactly the probability of success of the amplified algorithm.

We can implement this procedure with a single iteration of quantum search as long as we have a 33% overhead on the maximal number of allocated cells (similarly to the case of searching with a single query studied in [11]).

Quantum tree allocation. A more standard, but less time-efficient approach to implement ALLOC is to organize the memory cells in a complete binary tree (a heap), so that each node of the tree stores the number of unallocated cells in its children. This tree is not a quantum radix tree, since its size never changes, and no elements are inserted or removed. In order to obtain the uniform superposition of free cell addresses, we mimic the approach of QLOOKUP.

5.5 Higher-level Operations for Collision Walks

We now implement efficiently the higher-level operations required by our algorithms: setting up the initial vertex (SETUP), performing a quantum walk update (SWUP′), looking for collisions (CHECK) and extracting them (EXTRACT).

Representation. We consider the case of (multi-)collision search. Here the set S is a subset of $[N] = \{0,1\}^n$, but we also need to store the images of these elements by the function $f : \{0,1\}^n \to \{0,1\}^m$. Let $F = \{f(x)||x, x \in S\}$. A collision of f is a pair $(f(x)||x), (f(y)||y)$ such that $f(x) = f(y)$, i.e., the bit-strings have the same value on the first m bits.

Since our goal is to retrieve efficiently the collision pairs, we will store both a radix tree $T(S)$ to keep track of the elements, and $T(F)$ to keep track of the collisions. One should note that the sets F and S have the same size. When inserting or deleting elements, we insert and delete both in $T(S)$ and $T(F)$. These trees are stored in two separate chunks of memory cells.

SETUP. The unitary SETUP starts from an empty state $|0\rangle$ and initializes the tree to a uniform superposition of subsets of a given set. As long as sampling uniformly at random from this set is efficient, we can implement SETUP using a sequence of insertions in a tree that starts empty.

SWUP'. We show an efficient implementation of the unitary SWUP′:

$$\text{SWUP}' \left|T(S)\right\rangle \left|T(F)\right\rangle \left|c_{S\to S'}\right\rangle = \left|T(S')\right\rangle \left|T(F')\right\rangle \left|c_{S'\to S}\right\rangle \tag{13}$$

Where $c_{S\to S'}$ is the *coin register* which contains information on the transition of a set S to a set S'. As we have detailed before, the coin is encoded as a pair (j, z) where $j \in [R]$ is the index of an element in S, which has to be removed, and $z \in [N - R]$ is the index of an element in $\{0,1\}^n \backslash S$, which has to be inserted. We implement SWUP′ as follows:

1. First, we convert the coin register to a pair x, y where: • y is the z-th element of $\{0,1\}^n$ which is not in S and • x is the j-th element of S (according to the lexicographic ordering of bit-strings). For the first, we need a specific algorithm detailed in Appendix B, which accesses the tree $T(S)$. The second can be done easily if the invariant of each node stores the number of leaves in its subtree. Note that both the mapping from z to y, and from j to x, are reversible. At this point the state is $\left|T(S)\right\rangle \left|T(F)\right\rangle \left|x, y\right\rangle$.

2. We use INSERT† to delete x from $T(S)$, and delete $f(x)||x$ from $T(F)$.
3. We use INSERT to insert y in $T(S)$ and $f(y)||y$ in $T(F)$. At this point the state is: $|T(S')\rangle\,|T(F')\rangle\,|x,y\rangle$ where $S' = (S\backslash\{x\}) \cup \{y\}$ and F' is the set of corresponding images.
4. Finally, we convert the pair x, y back to a coin register.

Remark 2 (Walking in a reduced set). In our walk, we actually reduce the set of possible elements, due to the previously measured collisions. So the coin does not encode an element of $\{0,1\}^n\backslash S$, but of $\{0,1\}^n\backslash S\backslash\mathsf{Preim}(C)$, where C is our current table of multicollisions. An adapted algorithm is also given in Appendix B for this case.

Checking. We make the CHECK operation trivial, by defining an appropriate invariant of the tree $T(F)$. For each node in the tree, we count the number of multicollisions and preimages of $\mathsf{Im}(C)$ that the subtree rooted at this node contains. Then, the unitary CHECK simply tests whether the invariant at the root is zero.

During the operations of insertion and deletion in the tree, the invariant can be updated appropriately. Besides checking if the inserted element creates a new collision (resp., the deleted element removes one), we also need to check whether the image belongs to the set $\mathsf{Im}(C)$. During the run of the algorithm, $\mathsf{Im}(C)$ is classical, and can be stored in quantum-accessible classical memory.

Extracting. The most important property for our chained quantum walk is the capacity to *extract* multicollisions from the radix tree, in a way that preserves the rest of the state, and allows to reuse a superposition of *marked* vertices for the current walk, as a superposition of *unmarked* vertices for the next one. Recall from Section 4.1 that we have defined a table of multicollisions C, a set V_R^C of sets of size R in $\{0,1\}^n\backslash\mathsf{Preim}(C)$, and a set $M_R^C \subseteq V_R^C$ of *marked vertices*, which contain either a new element mapping to $\mathsf{Im}(C)$, or a new collision. Recall also from the proof of Theorem 5 that a random function, with probability $1-o(2^{-n})$, does not admit an r-collision $(x_1,\ldots,x_r)$ with $r = \mathcal{O}(n)$ for some appropriate constant. This limit on the size of multicollisions ensures that the extraction does not reduce too much the size of the current vertex.

The operation EXTRACT does:

$$\text{EXTRACT} : C, R, \frac{1}{\sqrt{|M_R^C|}} \sum_{S\in M_R^C} |S\rangle \mapsto C', R', \frac{1}{\sqrt{|V_{R'}^{C'}\backslash M_{R'}^{C'}|}} \sum_{S\in V_{R'}^{C'}\backslash M_{R'}^{C'}} |S\rangle \quad ,$$

i.e., it updates the current vertex state, but also reduces R to a smaller value R', and updates the table C into a bigger table C'. It is implemented as Algorithm 5. Although it is not strictly necessary, we have separated the subroutine CHECK into: CHECKP, which finds whether the set contains a new preimage of C, and CHECKC, which finds whether there is a new collision.

We now prove the correctness of Algorithm 5. We start with the uniform superposition of marked vertices, i.e., sets $S \subseteq \{0,1\}^n\backslash\mathsf{Preim}(C)$ of size R, which

Algorithm 5: Multicollision extraction: EXTRACT.

Input: C, R, uniform superposition over M_R^C
Output: C' R', uniform superposition over $V_{R'}^{C'} \backslash M_{R'}^{C'}$
$\mathsf{flag} \leftarrow \mathsf{true}$
$C' \leftarrow C$, $R' \leftarrow R$
Apply CHECKP and measure the result: let flag be the output
while *flag is true* **do**
  Perform a "quantum lookup" of the solution (new preimage)
  Select one uniformly at random, denote it x
  Copy x outside the tree; apply INSERT† to remove it; measure x
  $R' \leftarrow R - 1$
  Insert x in C', at the index of its image $f(x)$
  Apply CHECKP and measure the result: let flag be the output
Apply CHECKC and measure the result: let flag be the output
while *flag is true* **do**
  Perform a "quantum lookup" of the solution (new collision)
  Select one uniformly at random, denote it $(x_1, \ldots, x_r)$
  Write r in a new register
  Copy $(x_1, \ldots, x_r)$ outside the tree
  Apply INSERT† a total of $\mathcal{O}(n)$ times, in a controlled way depending on the exact value of r, to remove $x_1, \ldots, x_r$
  Measure r and $x_1, \ldots, x_r$
  $R' \leftarrow R - r$
  Insert a new entry $(x_1, \ldots, x_r)$ in C'
  Apply CHECKC and measure the result: let flag be the output

contain *at least* a solution tuple $x_1, \ldots, x_r$ which is either a (multi)-collision, or a new preimage.

The first loop removes all new preimages. Each time we measure an element, we collapse on the superposition of sets which contained it. After CHECKP returns 0 for the first time, the state collapses on the uniform superposition of all sets S such that:

$$S \subseteq \left(\{0,1\}^n \backslash \mathsf{Preim}(C')\right), |S| = R' = R - t, \left(\forall z \in S, f(z) \notin \mathsf{Im}(C')\right) \ ,$$

where t is the number of iterates of the loop that we had to perform. There is a variable number of such iterates but we expect only one to occur on average, since the typical case is for vertices to contain only one solution.

The second loop will run until there are no collisions anymore. New preimages cannot appear since we extract entire multicollision tuples. At the first loop iterate, assuming that CHECKC returns 1, we collapse on the uniform superposition of sets:

$$S \subseteq \left(\{0,1\}^n \backslash \mathsf{Preim}(C')\right), |S| = R', \left(\forall z \in S, f(z) \notin \mathsf{Im}(C') \wedge \exists x, y \in S, x \neq y, f(x) = f(y)\right) \ .$$

We select one of the solutions $(x_1, \ldots, x_r)$ at random, remove it, and measure the tuple $x_1, \ldots, x_r$. Let $u = h(x_1)$. After measurement, the state collapses on *all sets that do not contain* $x_1, \ldots, x_r$*, and contain no preimage of* u.

Since we update R' and C' accordingly, we obtain the sets:

$$S \subseteq (\{0,1\}^n \backslash \mathsf{Preim}(C')), |S| = R', \left(\forall z \in S, f(z) \notin \mathsf{Im}(C')\right) .$$

After repeatedly calling CHECKC and measuring, we will continue extracting collisions until CHECKC returns 0, i.e., we have collapsed on the sets which *do not* contain a collision. At this point, we have a uniform superposition of:

$$S \subseteq (\{0,1\}^n \backslash \mathsf{Preim}(C')), |S| = R', \left(\forall z \in S, f(z) \notin \mathsf{Im}(C') \wedge \forall x, y \in S, f(x) \neq f(y)\right) .$$

This is, by definition, the set of unmarked vertices (see Equation (6)).

Note that for this algorithm to work, we need to maintain invariants of the number of solutions (new preimages and multicollisions) that any subtree contains. These invariants only decrease during the loop iterates, and they are updated accordingly when we remove the solutions from the tree.

6 Searching for Many Collisions, in General

As we have seen, our new algorithm is valid (and tight) for all values of n, m and $k \leq 2n - m$ such that $k \leq \frac{m}{4}$. Two approaches can be used for higher k.

BHT. A standard approach to find multiple collisions, which works when m is small, is to extend the BHT algorithm [8]. We select a parameter ℓ, then make 2^ℓ queries, and look for 2^k collisions on this list of queries. This is done by a quantum search on $\{0,1\}^n$ for an input colliding with the list.

There are on average 2^{2n-m} collision pairs in the function, so a random element of $\{0,1\}^n$ has a probability $\mathcal{O}\left(2^{n-m}\right)$ to be in a collision pair. This gives $\mathcal{O}\left(2^{\ell-m+n}\right)$ collision pairs for the initial list.

Thus, a search for a collision with the list has $\mathcal{O}\left(2^{\ell-m+n}\right)$ solutions in a search space of size 2^n, and it requires $\sqrt{\frac{2^n}{2^{\ell+m-n}}} = 2^{(m-\ell)/2}$ iterates.

If this procedure is to output 2^k collisions, we need ℓ such that $2^{\ell-m+n} \geq 2^k$ i.e. $\ell - m + n \geq k$. By trying to equalize the complexity of the two steps, we obtain: $\ell = k + \frac{m-\ell}{2} \implies \ell = \frac{2k}{3} + \frac{m}{3}$ which is only valid for $k \leq 3n - 2m$. For a bigger k, we can repeat this. We find 2^{3n-2m} collisions in time (and memory) 2^{2n-m}, and we do this $2^{k-(3n-2m)}$ times, for a total time $\widetilde{\mathcal{O}}\left(2^{k+m-n}\right)$. If we want to restrict the memory then we obtain the tradeoff of $\widetilde{\mathcal{O}}\left(2^{k+m/2-\ell/2}\right)$ time using $\mathcal{O}\left(2^\ell\right)$ memory.

Using our method. If $k > m/4$, then the memory limitation in Theorem 5 on ℓ becomes relevant. In that case, as we are restricted to $\ell \leq m/2$, the minimal achievable time is $\widetilde{\mathcal{O}}\left(2^{k+m/2-\ell/2}\right) = \widetilde{\mathcal{O}}\left(2^{k+m/4}\right)$.

Conclusion. The time and memory complexities of the problem are the following (in $\log_2$ and without polynomial factors):

- If $k \leq 3n - 2m$: $\frac{2k}{3} + \frac{m}{3}$ time and memory using BHT
- Otherwise, if $k \leq \frac{m}{4}$: $\frac{2k}{3} + \frac{m}{3}$ time and memory using our algorithm
- Otherwise, if $m \leq \frac{4}{3}n$: $k + m - n$ time and $2n - m$ memory using BHT
- Otherwise, if $m \geq \frac{4}{3}n$: $k + \frac{m}{4}$ time and $\frac{m}{2}$ memory using our algorithm

This situation is summarized in Figure 1, and it allows us to conclude:

Theorem 6. *Let $f : \{0,1\}^n \to \{0,1\}^m$, $n \leq m \leq 2n$ be a random function. Let $k \leq 2n - m$. There exists an algorithm finding 2^k collisions in $\widetilde{\mathcal{O}}\left(2^{C(k,m,n)}\right)$ Clifford+T+qRAM gates, and using $\widetilde{\mathcal{O}}\left(2^{C(k,m,n)}\right)$ quantum queries to f, where:*

$$C(k,m,n) = \max\left(\frac{2k}{3} + \frac{m}{3}, k + \min\left(m - n, \frac{m}{4}\right)\right) \quad . \tag{14}$$

Proof. We check that: $k \leq 3n - 2m \iff \frac{2k}{3} + \frac{m}{3} \geq k + m - n$ and $k \leq \frac{m}{4} \iff \frac{2k}{3} + \frac{m}{3} \geq k + \frac{m}{4}$. □

We conjecture that the best achievable complexity is, in fact, $C(k,m,n) = \frac{2k}{3} + \frac{m}{3}$ for any admissible values of k, m and n. It would however require a non-trivial extension of our algorithm, capable of outputting collisions at a higher rate than what we currently achieve.

In terms of time-memory trade-offs, we can summarize the results as:

Theorem 7 (General Time-memory tradeoff). *For all $k \leq \ell \leq \min(2k/3 + m/3, \max(2n - m, m/2))$, there exists an algorithm that computes 2^k collisions using $\widetilde{\mathcal{O}}\left(2^{\ell}\right)$ qubits and $\widetilde{\mathcal{O}}\left(2^{k+m/2-\ell/2}\right)$ Clifford+T+qRAM gates and quantum queries to f.*

Similarly, as in [15], we conjecture that the trade-off should be achievable for all $\ell \leq 2k/3 + m/3$.

7 Applications

In this section, we show how our algorithm can be used as a building block for lattice sieving and to solve the limited birthday problem. We also discuss the problem of multicollision search.

7.1 Improvements in quantum sieving for solving the Shortest Vector Problem

In this section, we present the improvement of our reusable quantum walks to lattice sieving algorithms. A lattice $\mathcal{L} = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_d) := \{\sum_{i=1}^{d} z_i \mathbf{b}_i : z_i \in \mathbb{Z}\}$ is the set of all integer combinations of linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_d \in$

$\mathbb{R}^d$. We call d the *rank* of the lattice and $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ a *basis* of the lattice. The most important computational problem on lattices is the Shortest Vector Problem (SVP). Given a basis for a lattice $\mathcal{L} \subseteq \mathbb{R}^d$, SVP asks to compute a non-zero vector in $\mathcal{L}$ with the smallest Euclidean norm. The main lattice reduction algorithm used for lattice-based cryptanalysis is the famous BKZ algorithm [31]. It internally uses an algorithm for solving (near) exact SVP in lower-dimensional lattices. Therefore, finding faster algorithms to solve exact SVP is critical to choosing security parameters of cryptographic primitives.

Previously, the fastest quantum algorithm solved SVP under heuristic assumptions in $2^{0.2570d+o(d)}$ time [9]. It applies the MNRS quantum walk technique to the state-of-the-art classical algorithm called lattice sieving, where we combine close vectors together to obtain shorter vectors at each step. It was noted in [9] that the algorithm could be slightly improved if we could find many marked vertices in a quantum walk without repaying the setup each time, which is exactly what we showed in Section 4. We redid the analysis of [9] with this improvement and show the following

Proposition 4. *There exists a quantum algorithm that solves SVP under heuristic assumptions in* $2^{0.2563d+o(d)}$

Proving this statement requires to restate the whole framework and analysis of [9], which we do in Appendix C.

7.2 Solving the Limited Birthday Problem

The following problem is very common in symmetric cryptanalysis. It appears for example in impossible differential attacks [6], but also in rebound distinguishers [16]. In the former case we use generic algorithms to solve the problem for a black-box E, and in the latter, a valid distinguisher for E is defined as an algorithm outputting the pairs faster than the generic one.

Problem 4 (Limited Birthday). Given access to a black-box permutation $E : \{0,1\}^n \to \{0,1\}^n$ and possibly its inverse E^{-1}, given two vector spaces $\mathcal{D}_{\text{in}}$ and $\mathcal{D}_{\text{out}}$ of sizes $2^{\Delta_{\text{in}}}$ and $2^{\Delta_{\text{out}}}$ respectively, find 2^k pairs x, x' such that $x \neq x', x \oplus x' \in \mathcal{D}_{\text{in}}, E(x) \oplus E(x') \in \mathcal{D}_{\text{out}}$.

For simplicity, we will focus only on the time complexity of the problem, although some parameter choices require a large memory as well. Classically the best known time complexity is given in [6]:

$$\max\left(\min_{\Delta \in \{\Delta_{\text{in}}, \Delta_{\text{out}}\}}\left(\sqrt{2^{k+n+1-\Delta}}\right), 2^{k+n+1-\Delta_{\text{in}}-\Delta_{\text{out}}}\right) . \tag{15}$$

This complexity is known to be tight for $2^k = 1$ [16].

In the quantum setting, we need to consider superposition access to E and possibly E^{-1} to have a speedup on this problem. Previously the methods used [23] involved only individual calls to Ambainis' algorithm (when there are few solutions) or an adaptation of the BHT algorithm (when there are many solutions).

The quantum algorithm, as the classical one, relies on the definition of *structures* of size $2^{\Delta_{\text{in}}}$, which are subsets of the inputs of the form $T_x = \{x \oplus v, v \in \mathcal{D}_{\text{in}}\}$ for a fixed x. For a given structure T_x, we can define a function $h_x : \{0,1\}^{\Delta_{\text{in}}} \to \{0,1\}^{n-\Delta_{\text{out}}}$ such that any collision of h_x yields a pair solution to the limited birthday problem. The expected number of collisions of a single h_x is $C := 2^{2\Delta_{\text{in}}+\Delta_{\text{out}}-n}$, and there are three cases:

1. $C < 1$: we follow the approach of [23], which is to repeat 2^k times a Grover search among structures, to find one that contains a pair (this is done with Ambainis' algorithm). The time exponent is $k + \frac{n-\Delta_{\text{out}}}{2} - \frac{\Delta_{\text{in}}}{3}$.
2. $1 < C < 2^k$: we need to consider several structures and to extract all of their collision pairs. Using Theorem 6 this gives a time exponent:
$$\max\left(k + \frac{2}{3}(n - \Delta_{\text{in}} - \Delta_{\text{out}}), k + \min\left(n - \Delta_{\text{out}} - \Delta_{\text{in}}, \frac{n - \Delta_{\text{out}}}{4}\right)\right)$$
3. $2^k < C$: we need only one structure. To recover 2^k pairs, we need a time exponent (by Theorem 6):
$$\max\left(\frac{2k}{3} + \frac{n - \Delta_{\text{out}}}{3}, k + \min\left(n - \Delta_{\text{out}} - \Delta_{\text{in}}, \frac{n - \Delta_{\text{out}}}{4}\right)\right)$$

Finally, we can swap the roles of Δ_{in} and Δ_{out} and take the minimum. Unfortunately this does not lead to an equation as simple as Equation (15).

7.3 On multicollision-finding

A natural extension of this work would be to look for multicollisions.

Problem 5 (r-collision search). Let $f : \{0,1\}^n \to \{0,1\}^m$ be a random function. Find an r-collision of f, that is, a tuple $(x_1, \ldots, x_r)$ of distinct elements such that $f(x_1) = \ldots = f(x_r)$.

As with collisions, the lower bound by Liu and Zhandry [26] is known to be tight when $m \leq n$. The corresponding algorithm is an extension of the BHT algorithm which constructs increasingly smaller lists of i-collisions, starting with 1-collisions (evaluations of the function f on arbitrary points) and ending with a list of r-collisions.

This algorithm, given in [17,18], finds 2^k r-collisions in time and memory:

$$\widetilde{\mathcal{O}}\left(2^{k\frac{2^{(r-1)}}{2^r-1}} 2^{m\frac{2^{(r-1)}-1}{2^r-1}}\right) .$$

As with 2-collisions, it is possible to extend it when $m > n$. Of course, there's a constraint: the list i must contain more tuples that are part of an $i+1$-collision than the size of the list $i+1$.

The size of each i-collision list is $N_i = 2^{k\frac{2^r-2^{r-i}}{2^r-1}} 2^{m\frac{2^{r-i}-1}{2^r-1}}$. The probability that an i-collision extends to an $i+1$-collision is of order 2^{n-m}. Hence, for the algorithm to work, we must have, for all i, $N_{i+1}/N_i \leq 2^{n-m}$. This means:

$$k\frac{2^{r-i-1}}{2^r-1} - m\frac{2^{r-i-1}}{2^r-1} \leq n-m \ .$$

This constraint is the most restrictive for the largest possible i, $r-1$. We obtain the following constraint, which subsumes the others:

$$k\frac{1}{2^r-1} + m\left(1-\frac{1}{2^r-1}\right) \leq n \ .$$

This gives the point up to which this algorithm meets the lower bound. We could use our new algorithm as a subroutine in this one, to find 2-collisions, and this would allow to relax the constraint over N_2/N_1. Unfortunately, this cannot help to find multicollisions, as the other constraints are more restrictive. More generally, these constraints show that it is not possible to increase the range of the BHT-like r-collision algorithm solely by using an $r-i$-collision algorithm with an increased range.

Acknowledgments. A.S. wants to thank Nicolas David and María Naya-Plasencia for discussions on the limited birthday problem. A.S. has been supported by ERC-ADG-ALGSTRONGCRYPTO (project 740972). Y.S. is supported by EPSRC grant EP/S02087X/1 and EP/W02778X/1. This work received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PETQ-0007 EPiQ and ANR-22-PETQ-0008 PQ-TLS. All authors would like to thank Schloss Dagstuhl and the organizers of the Dagstuhl Seminar 21421 "Quantum Cryptanalysis" where this work was initiated, and the reviewers of EUROCRYPT 2023 for helpful comments.

References

1. Aaronson, S., Shi, Y.: Quantum lower bounds for the collision and the element distinctness problems. J. ACM **51**(4), 595–605 (2004)
2. Ambainis, A.: Quantum Walk Algorithm for Element Distinctness. SIAM J. Comput. **37**(1), 210–239 (2007)
3. Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms. p. 10–24. SODA '16, Society for Industrial and Applied Mathematics, USA (2016)
4. Bernstein, D.J., Jeffery, S., Lange, T., Meurer, A.: Quantum algorithms for the subset-sum problem. In: PQCrypto. LNCS, vol. 7932, pp. 16–33. Springer (2013)
5. Bonnetain, X., Bricout, R., Schrottenloher, A., Shen, Y.: Improved classical and quantum algorithms for subset-sum. In: ASIACRYPT (2). Lecture Notes in Computer Science, vol. 12492, pp. 633–666. Springer (2020)

6. Boura, C., Naya-Plasencia, M., Suder, V.: Scrutinizing and improving impossible differential attacks: Applications to clefia, camellia, lblock and simon. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 8873, pp. 179–199. Springer (2014)
7. Brassard, G., Hoyer, P., Mosca, M., Tapp, A.: Quantum amplitude amplification and estimation. Contemporary Mathematics **305**, 53–74 (2002)
8. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: LATIN. Lecture Notes in Computer Science, vol. 1380, pp. 163–169. Springer (1998)
9. Chailloux, A., Loyer, J.: Lattice sieving via quantum random walks. In: ASIACRYPT (4). Lecture Notes in Computer Science, vol. 13093, pp. 63–91. Springer (2021)
10. Chailloux, A., Naya-Plasencia, M., Schrottenloher, A.: An efficient quantum collision search algorithm and implications on symmetric cryptography. In: ASIACRYPT (2). Lecture Notes in Computer Science, vol. 10625, pp. 211–240. Springer (2017)
11. Chi, D.P., Kim, J.: Quantum database search by a single query. In: QCQC 1998. LNCS, vol. 1509, pp. 148–151. Springer, Heidelberg (1999)
12. Dinur, I.: Tight time-space lower bounds for finding multiple collision pairs and their applications. In: EUROCRYPT (1). Lecture Notes in Computer Science, vol. 12105, pp. 405–434. Springer (2020)
13. Flajolet, P., Odlyzko, A.M.: Random mapping statistics. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 434, pp. 329–354. Springer (1989)
14. Grover, L.K.: A Fast Quantum Mechanical Algorithm for Database Search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing 1996. pp. 212–219. ACM (1996)
15. Hamoudi, Y., Magniez, F.: Quantum time-space tradeoff for finding multiple collision pairs. In: TQC. LIPIcs, vol. 197, pp. 1:1–1:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
16. Hosoyamada, A., Naya-Plasencia, M., Sasaki, Y.: Improved attacks on sliscp permutation and tight bound of limited birthday distinguishers. IACR Trans. Symmetric Cryptol. **2020**(4), 147–172 (2020)
17. Hosoyamada, A., Sasaki, Y., Tani, S., Xagawa, K.: Improved quantum multicollision-finding algorithm. In: PQCrypto. Lecture Notes in Computer Science, vol. 11505, pp. 350–367. Springer (2019)
18. Hosoyamada, A., Sasaki, Y., Tani, S., Xagawa, K.: Quantum algorithm for the multicollision problem. Theor. Comput. Sci. **842**, 100–117 (2020)
19. Jaques, S., Schanck, J.M.: Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 11692, pp. 32–61. Springer (2019)
20. Jaques, S., Schrottenloher, A.: Low-gate quantum golden collision finding. In: SAC. Lecture Notes in Computer Science, vol. 12804, pp. 329–359. Springer (2020)
21. Jeffery, S.: Frameworks for Quantum Algorithms. Ph.D. thesis, University of Waterloo, Ontario, Canada (2014), http://hdl.handle.net/10012/8710
22. Kachigar, G., Tillich, J.: Quantum information set decoding algorithms. In: PQCrypto. LNCS, vol. 10346, pp. 69–89. Springer (2017)
23. Kaplan, M., Leurent, G., Leverrier, A., Naya-Plasencia, M.: Quantum differential and linear cryptanalysis. IACR Trans. Symmetric Cryptol. **2016**(1), 71–94 (2016)
24. Kirshanova, E., Mårtensson, E., Postlethwaite, E.W., Moulik, S.R.: Quantum algorithms for the approximate k-list problem and their application to lattice siev-

ing. In: ASIACRYPT. LNCS, vol. 11921, pp. 521–551. Springer (2019). https://doi.org/10.1007/978-3-030-34578-5_19
25. Laarhoven, T.: Search problems in cryptography: from fingerprinting to lattice sieving. Ph.D. thesis, Mathematics and Computer Science (Feb 2016), proefschrift
26. Liu, Q., Zhandry, M.: On finding quantum multi-collisions. In: EUROCRYPT (3). Lecture Notes in Computer Science, vol. 11478, pp. 189–218. Springer (2019)
27. Magniez, F., Nayak, A., Roland, J., Santha, M.: Search via quantum walk. SIAM J. Comput. **40**(1), 142–164 (2011)
28. Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. Journal of Mathematical Cryptology **2**(2), 181–207 (2008). https://doi.org/doi:10.1515/JMC.2008.009, https://doi.org/10.1515/JMC.2008.009
29. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information (2002)
30. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. J. Cryptol. **12**(1), 1–28 (1999)
31. Schnorr, C.: A hierarchy of polynomial time lattice basis reduction algorithms. Theor. Comput. Sci. **53**, 201–224 (1987). https://doi.org/10.1016/0304-3975(87)90064-8, https://doi.org/10.1016/0304-3975(87)90064-8

Appendix

A Extraction without Measurement

Our presentation of the chained quantum walks in Section 4 and Section 5 relies on measurements to extract the collisions, but in some cases, we might want to perform this operation reversibly, i.e., without any measurement. In this section, we explain how this can be done.

Suppose that we want to extract K collisions. The internal state of our algorithm will contain both a vertex in (one step of) our walk, the current parameters of the walk (vertex size, number of solutions...) and $\mathcal{O}(K)$ registers to store the produced collisions. We can then create a unitary operator that on the current state:

1. Applies CHECK and stores its result in a new qubit.
2. Controlled on the result of CHECK (i.e., if a collision exists), finds a collision, removes it from the vertex, *and* updates the parameters of the walk, which determine the number of iterates it has to make.
3. Controlled negatively on the result of CHECK (i.e., if there is no collision), performs a quantum walk with the current parameters.

In this unitary operator, we set the precision of the phase estimation circuit (in the walk step) at the highest level required, i.e., depending on the initial vertex size, and it will work correctly for all walks.

After running a sequence of $2K(1 + o(1))$ such operators, the state becomes a superposition which contains, with high probability, at least K collisions. The full algorithm is measurement-free and can be uncomputed by re-inserting the extracted elements and performing the quantum walks backwards, depending on the stored CHECK results.

B z-th Element Outside the Radix Tree

In this section, we solve the following problem:

> Find the value y of the z-th element of $\{0,1\}^n$ which is not in a given subset of $\{0,1\}^n$ represented by a radix tree T.

We need the following invariant in the tree: each node n of T stores the number of leaves in the subtree rooted at n. We denote this quantity by $\text{leaves}(n)$. See Figure 4 for an example.

Assume that T has R elements in $\{0,1\}^n$, let $N = 2^n$. Assume here that we have some easily computable order on $\{0,1\}^n$, represented by a map $\phi : \{0,1\}^n \to [N]$ that assigns to each bit-string its index, and its inverse ϕ^{-1} also easily computable. Given $i \in [N - R]$, the goal is to find the i^{th} element in $\{0,1\}^n \setminus T$.

Theorem 8. `FindNthNotInTree`(i, T) *returns the* i^{th} *element in* $\{0,1\}^n \setminus T$ *in* $\text{poly}(n)$ *time.*

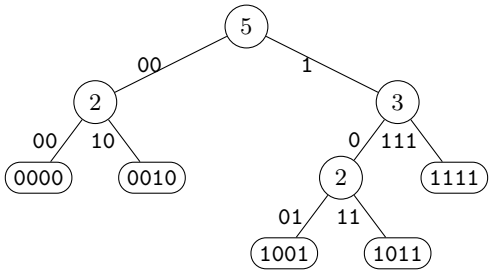

Fig. 4. Example of tree where each node stores the number of leaves in the subtree. We omit this quantity (which is 1) on the leaves themselves for readability.

Algorithm 6: FindNthNotInSubtree(i, T, `node`)

Input: index i, radix tree T, node `node`
Output: i^{th} element in $\{x \in \{0,1\}^n : \phi(x) \geqslant \phi(\ell)\} \setminus$ `subtree(`T`,`*`node`*`)` where ℓ is the bit-string of the left-most leaf in the subtree rooted at `node`
if *`node` is a leaf* **then**
  Let x be the bit-string of `node`
  return $\phi^{-1}(\phi(x) + i)$
else
  `/* Here node must have two children left(`*`node`*`) and right(`*`node`*`) */`
  Let x be the bit-string of the left-most leaf in subtree rooted at `node`
  Let y be the bit-string of the left-most leaf in subtree rooted at `right(`*`node`*`)`
  `/* compute the number of elements in` $[x, y) \setminus T$ `*/`
  $\delta \leftarrow \phi(y) - \phi(x) -$ `leaves(left(`*`node`*`))`
  if $i \geqslant \delta$ **then**
    return `FindNthNotInSubtree(`$i - \delta, T,$`right(`*`node`*`))`
  else
    return `FindNthNotInSubtree(`$i, T,$`left(`*`node`*`))`

Algorithm 7: FindNthNotInTree(i, T)

Input: index i, radix tree T
Output: i^{th} element in $\{0,1\}^n \setminus T$
Compute the bit-string x of the leftmost leaf of the tree T
if $i < \phi(x)$ **then**
  return $\phi^{-1}(i)$
else
  return `FindNthNotInSubtree(`$i - \phi(x)$, T, `root(`T`))`

We now consider the same problem where we have two trees T and T' and we want to find the i^{th} element in $\{0,1\}^n$ which is not in T and not in T'. We assume that T and T' have **disjoint** leaves. This problem appears in our chained quantum walk in Section 4, where T' is a classical radix tree in which we store the collision pairs that were found in previous walk steps, and T is the radix tree inside one node of the current quantum walk. Indeed, in order to walk to a new vertex, we need to exclude both the elements that were previously measured and those that belong to the current vertex.

Algorithm 8: CountInIntervalNotSubtree(u, v, T, `node`)

Input: bit-strings u and v, radix tree T, a node `node` of T
Output: size of $[u,v) \setminus \{$all the leaves in the subtree root at `node`$\}$
Let x be the bit-string of the left-most leaf in subtree rooted at `node`
Let y be the bit-string of the right-most leaf in subtree rooted at `node`
`/* when the interval` $[u,v)$ `entirely covers the subtree */`
if $\phi(u) \leqslant \phi(x)$ *and* $\phi(y) < \phi(v)$ **then**
  return $\phi(v) - \phi(u) -$ `leaves(`*node*`)`
`/* when the interval` $[u,v)$ `is disjoint from the subtree */`
if $\phi(v) \leqslant \phi(x)$ *or* $\phi(y) < \phi(u)$ **then**
  return $\phi(v) - \phi(u)$
`/* if we are here, node cannot be a leaf */`
Let z be the bit-string of the left-most leaf in subtree rooted at `right(`*node*`)`
if $\phi(v) \leqslant \phi(z)$ **then**
  `/* when` $[u,v)$ `only intersects the left subtree */`
  return `CountInIntervalNotTree(`u, v, T, `left(`*node*`))`
else if $\phi(z) \leqslant \phi(u)$ **then**
  `/* when` $[u,v)$ `only intersects the right subtree */`
  return `CountInIntervalNotTree(`u, v, T, `right(`*node*`))`
else
  return `CountInIntervalNotTree(`u, z, T, `left(`*node*`))`
   $+$`CountInIntervalNotTree(`z, v, T, `right(`*node*`))`

Algorithm 9: CountInIntervalNotTree(u, v, T)

Input: bit-strings u and v, radix tree T
Output: size of $[u,v) \setminus T$
return `CountInIntervalNotSubtree(`$u, v, T,$ `root(`T`))`

Theorem 9. `FindNthNotInTwoTrees`(i, T, T') *returns the* i^{th} *element in* $\{0,1\}^n \setminus (T \cup T')$ *in* $\mathrm{poly}(n)$ *time.*

Algorithm 10: FindNthNotInTwoSubtrees(i, T, T', `node`)

Input: index i, radix trees T and T' with disjoint leaves, node `node` of T
Output: i^{th} element in $\{x \in \{0,1\}^n : \phi(x) \geqslant \phi(\ell)\} \setminus (\texttt{subtree}(T,\textit{node}) \cup T')$ where ℓ is the bit-string of the left-most leaf in the subtree of T rooted at `node`

```
if node is a leaf then
    Let x be the bit-string of node
    δ ← CountInIntervalNotTree(0^n, x, T')
    return FindNthNotInSubtree(i + δ + 1, T')
else
    /* Here node must have two children left(node) and right(node)
       */
    Let x be the bit-string of the left-most leaf in subtree rooted at node
    Let y be the bit-string of the left-most leaf in subtree rooted at
     right(node)
    /* compute the number of elements in [x, y) \ (T ∪ T')          */
    δ ← CountInIntervalNotTree(x,y,T') − leaves(left(node))
    if i ⩾ δ then
        return FindNthNotInTwoSubtrees(i − δ,T,T',right(node))
    else
        return FindNthNotInTwoSubtrees(i,T,T',left(node))
```

Algorithm 11: FindNthNotInTwoTrees(i, T, T')

Input: index i, radix trees T and T' with disjoint leaves
Output: i^{th} element in $\{0,1\}^n \setminus (T \cup T')$

```
Compute the bit-string x of the leftmost leaf of the tree T
/* compute the number of elements on the left of T that are not in
   T'                                                               */
δ ← CountInIntervalNotTree(0^n, x, T')
if i < δ then
    return FindNthNotInTree(i, T')
else
    return FindNthNotInTwoSubtrees(i − δ, T, T', root(T))
```

C Redoing the Analysis of [9] with our Chained Quantum Walk

Without going through the whole framework of the [9] algorithm, we present its main parameters and ideas, and how our quantum walks improves it. The sieving algorithm works as follows: we start from N lattice points $x_1, \ldots, x_N$ on the d-dimensional sphere of some norm R and we want to find N pairs (x_i, x_j) (with $i \neq j$) such that the norm of $x_i - x_j$ is slightly smaller than R. This gives us N new lattice points of slightly smaller norm. This is one sieving step and the full algorithm performs the above sieving step $poly(d)$ times, until we find a small vector.

The original NV-sieve [28] uses the heuristic that lattice points constructed this way behave like random points on the sphere. For a fixed radius R, we say that a pair of points (x_i, x_j) reduces if $||x_i - x_j||_2 < R$. The probability that a pair of random points reduces is known to be

$$V_d(\pi/3) := \text{poly}(d) \sin^d(\pi/3) \ .$$

Since we have $O(N^2)$ different pairs (x_i, x_j), we take $N = \Theta(\frac{1}{V_d(\pi/3)}) = 2^{0.2075d+o(d)}$ to ensure that we can construct N different new points at each sieving step. The NV-sieve consists, at each sieving step, in checking all the pairs (x_i, x_j) in order to find all reducible pairs. This takes time $O(N^2) = 2^{0.415d+o(d)}$ using a classical computer and $O(N^{3/2}) = 2^{0.301d+o(d)}$ using a quantum computer that performs Grover's search.

Locality sensitive filtering. The idea of locality sensitive filtering for sieving is the following: instead of checking all the pairs (x_i, x_j), we check in priority those that have a higher chance of being close by using locality sensitive functions. There has been an extensive literature on the subject, see for instance [25]. In the most recent iterations [3], we start from a code C which behaves as a random code but which can be efficiently list-decoded. Then, for each lattice point, we compute the closest codewords (that is, we decode the lattice point), and we split the set of lattice points into buckets identified by each close codeword. Now, we only consider pairs of vectors that are in the same bucket. Of course, two vectors that lie in different buckets might form a reducible pair. Hence, this procedure has to be repeated with new random codes in order to get almost all reducible pairs.

In order to find good pairs, we typically check all pairs within each bucket. In the quantum setting, one can use Grover's algorithm, which corresponds to Laarhoven's algorithm [25] or a quantum walk, which corresponds to the [9] algorithm.

C.1 The [9] quantum algorithm and our improvement

We need the following definitions. We work on the d-dimensional sphere of radius R. For a point $\mathbf{v}$ and angle α, the spherical cap $\mathcal{H}_{\mathbf{v},\alpha}$ is the set of points of the

sphere at angle at most α from $\mathbf{v}$. For an angle $\alpha \in [0, \pi/2]$, $V_d(\alpha)$ is the ratio of the volume of a spherical cap of angle α to the volume of the d-dimensional sphere. We have

$$V_d(\alpha) = \text{poly}(d) \sin^d(\alpha) \ .$$

We restate (slightly informally) the quantum algorithm for lattice sieving, as presented in [9]. We fix a dimension d.

Algorithm 12: Sieving algorithms of [9] using LSF with parameter $c \in (0, 1)$

Input: a list L of N lattice vectors of norm R, a constant $\gamma < 1$ and parameter $c \in (0, 1)$.
Output: a list L' of N lattice vectors of norm at most γR.

```
L' := ∅
while |L'| ≤ N do
    Sample a random product code C = c_1, ..., c_{N^{1-c}} of size N^{1-c}. Let α st.
      V(α) = 1/N^{1-c}.
    For i ∈ [N^{1-c}], prepare f_α(c_i) = ∅.
    for v ∈ L do
        Compute T_v = H_{v,α} ∩ C.
        for c ∈ T_v do
            f_α(c) := f_α(c) ∪ {v}
        /* at the end of the for v ∈ L loop, f_α(c) contains all
           points v ∈ L at angle at most α from c                    */
    for i ∈ [N^{1-c}] do
        S ← FAS_1(f_α(c_i)) /* FAS_1(f_α(c_i)) finds a constant fraction of
           solution pairs (x, y) for x, y ∈ f_α(c_i)                 */
        L' := L' ∪ S
return L'
```

Algorithm 12 has the following running time:

Proposition 5 ([9]). *Algorithm 12 has the following asymptotic running time:*

$$T = \max\{1, N^{c-\zeta}\} \cdot (N + N^{1-c} FAS_1). \tag{16}$$

where

- $N = \frac{1}{V_d(\pi/3)} \approx 2^{0.2075d + o(d)}$.
- α *st.* $V_d(\alpha) = N^{-(1-c)}$.
- $\theta^*_\alpha = 2 \arcsin(\frac{1}{2\sin(\alpha)})$.
- ζ *st.* $N^\zeta = N^{2c} V_d(\theta^*_\alpha)$.
- FAS_1 *is the running time of the* FAS_1 *subroutine.*

The authors of [9] use a quantum walk in order to solve the FAS_1 problem.

Proposition 6 ([9]). *For a parameter c_1, let*

- *β st. $V_d(\beta) = \frac{1}{N^{c_1}}$.*
- *ρ_0 st. $N^{\rho_0} = \frac{V_d(\beta)}{W_d(\beta,\theta^*_\alpha)}$, where $W_d(\beta,\theta^*_\alpha) = \mathrm{poly}(d) \cdot \left(1 - \frac{2\cos^2(\beta)}{1+\cos(\theta^*_\alpha)}\right)^{d/2}$.*

In order to solve FAS_1 with parameter c_1, it is enough to perform N^{ρ_0} quantum walks on a Johnson graph $J(c, c_1)$ and, each time, and to find a constant fraction of its marked vertices. This quantum walk has the following properties

$$\mathsf{S} = N^{c_1},\ \delta = N^{-c_1},\ \varepsilon = N^{2c_1-\rho_0} V_{d-1}(\theta^*_\alpha),\ \mathsf{U} = 1,\ \mathsf{C} = 1.$$

from which we can deduce that the number of marked vertices is $N^{\zeta-\rho_0}$. The total running time of FAS_1 is therefore

$$FAS_1 = N^{\rho_0} \cdot N^{\zeta-\rho_0} \left(\mathsf{S} + \frac{1}{\sqrt{\varepsilon}} \left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right). \tag{17}$$

In the above proposition, for each of the quantum walks, there is the multiplicative term $N^{\zeta-\rho_0}$ which comes from the fact that we want to find $N^{\zeta-\rho_0}$ marked elements in each quantum walk. However, with our new results, we are able to avoid this multiplicative cost in front of the setup. Using Theorem 4, we can replace Equation (17) with

$$FAS_1 = N^{\rho_0} \cdot \left(\mathsf{S} + \frac{N^{\zeta-\rho_0}}{\sqrt{\varepsilon}} \left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right). \tag{18}$$

This is justified because the quantum walks described above are actually walks for collision finding (with a different condition on marked elements), and the optimized parameters of the walk we obtain are, as we will show, consistent with the requirements of Theorem 4.

With this improvement, we redid the optimization of [9] and obtained the following new running time for quantum sieving. We take the following set of parameters: $c \approx 0.3875, c_1 \approx 0.27$ which gives $\zeta \approx 0.1568$ and $\rho_0 \approx 0.1214$. Notice that with these parameters, we are indeed in the range of Theorem 4 since the number of solutions we extract is $2^k = N^{\zeta-\rho_0} \approx N^{0.0354}$ and the range of the function f on which we collision is $2^m = 2^{c_1} \approx N^{0.27}$ (the number of points in the code), so we indeed have $k \leq \frac{m}{4}$. The parameters of the quantum walk become:

$$\mathsf{S} \approx N^{0.27},\ \varepsilon \approx N^{-0.2},\ \delta \approx N^{-0.27},\ \mathsf{U} = \mathsf{C} = 1\ .$$

This gives $FAS_1 \approx N^{0.27}$. Plugging this into Equation (16), we get a total running time of $N^{1.2347}$ which is equal to $2^{0.2563d+o(d)}$ improving slightly the previous running time of $2^{0.2570d+o(d)}$.