Improved Quantum Algorithms for the k-XOR Problem

André Schrottenloher [*]

Cryptology Group, CWI, Amsterdam, The Netherlands
`firstname.lastname@m4x.org`

Abstract. The k-XOR problem can be generically formulated as the following: given many n-bit strings generated uniformly at random, find k distinct of them which XOR to zero. This generalizes collision search (two equal elements) to a k-tuple of inputs.
This problem has become ubiquitous in cryptanalytic algorithms, including variants in which the XOR operation is replaced by a modular addition (k-SUM) or other non-commutative operations (e.g., the composition of permutations). The case where a single solution exists on average is of special importance.
At EUROCRYPT 2020, Naya-Plasencia and Schrottenloher defined a class of *quantum merging algorithms* for the k-XOR problem, obtained by combining quantum search. They represented these algorithms by a set of *merging trees* and obtained the best ones through linear optimization of their parameters.
In this paper, we give a simplified representation of merging trees that makes their analysis easier. We give better quantum algorithms for the Single-solution k-XOR problem by relaxing one of the previous constraints, and making use of quantum walks. Our algorithms subsume or improve over all previous quantum algorithms for Single-solution k-XOR. For example, we give an algorithm for 4-XOR (or 4-SUM) in quantum time $\widetilde{\mathcal{O}}(2^{7n/24})$.

Keywords: Quantum algorithms, merging algorithms, k-XOR, k-SUM, bicomposite search.

1 Introduction

The *collision search problem* for a random function can be formulated as follows: given a random $h : \{0,1\}^n \to \{0,1\}^n$, find a pair of distinct inputs (x,y) such that $h(x) = h(y)$. This problem is ubiquitous in cryptography and collision search algorithms have been well studied. It is well known that, as formulated here, it can be solved in about $\mathcal{O}\big(2^{n/2}\big)$ classical queries to h and time. Using Floyd's cycle-finding algorithm, we need only polynomial memory.

A possible generalization would be to look for more than two elements having the same image: the problem (*multicollision* search) then becomes harder.

[*] Part of this work was done while the author was at Inria, France.

Another would be to have *more than two elements collide* in the sense that they sum to zero, or that their combination satisfies some constraint. This leads to the *Generalized Birthday Problem*, or k-XOR for us, formulated by Wagner [31]:

> Given k lists of random n-bit strings: $\mathcal{L}_1, \dots, \mathcal{L}_k$ which can be extended at will, find a k-tuple $(y_1, \dots, y_k) \in (\mathcal{L}_1 \times \dots \times \mathcal{L}_k)$ such that $y_1 \oplus \dots \oplus y_k = 0$.

In [31], Wagner gave an algorithm to solve k-XOR for any k, based on the *merging* building block. Although the idea of merging had been around for a longer time, with examples like [10], this was the first generic k-list merging algorithm. Later on, many works have either pursued the generic direction [28, 12], or the optimization of more specific algorithms. For example, the best algorithms for randomized instances of subset-sum [19, 3, 6] actually solve k-list problems with additional constraints, and use *merging* as an algorithmic subroutine.

Quantum k-XOR Algorithms. Obviously, *quantum* k-XOR algorithms can be used as replacements for classical ones in the context of quantum cryptanalysis. But our need for understanding the quantum speedups for k-XOR goes further, as quantum k-list algorithms of similar shapes have played a role in generic decoding [21] or in lattice sieving [22]. Knowing and improving the "generic" advantage of k-XOR algorithms may help for further improvements in these specific settings.

Grassi et al. [17] tackled the Many-solutions case (the case initially studied by Wagner) for a generic k. A more complete picture was obtained in [27]. Quantum algorithms for k-XOR were extended to a whole family derived from classical merging strategies, among which some appear to be optimal. These *quantum merging algorithms* were represented syntactically as *merging trees*, with some parameters to optimize linearly. Besides, this was the first study of the Single-solution case for a generic k.

Contrary to what occurs classically, the Single-solution k-XOR problem has a quantum time complexity advantage when k increases. For example, the Single-solution 2-XOR problem has a quantum time complexity $\widetilde{\mathcal{O}}(2^{n/3})$ [1], and an algorithm of time complexity $\widetilde{\mathcal{O}}(2^{0.3n})$ for the 4-XOR problem has been given in [5]. In [27], a closed formula for the time complexity exponent, depending on k, was obtained. Though the algorithms differed from [5] (as they did not use quantum walks), their complexity exponent also reached 0.3 at best.

Contributions. In this paper we give a simplified definition of *merging trees*, with a better emphasis on the correspondence between classical and quantum merging algorithms. In the Many-solutions case, we recover the algorithms of [27] and obtain simpler proofs of their optimality in the class of merging trees. In the Single-solution case, we simplify the presentation of [27] and modify one of its constraints[1]. We obtain a new closed formula with a convergence towards

[1] Our code implementing the new Mixed Integer Linear Program is available at: `https://github.com/Aschtlr/quantum-kxor` As in [27], we use the MILP solver of the SCIP suite [15, 16].

2/7 instead of 0.3. Finally, we introduce *quantum walks* as a new building block in these algorithms. They allow to reduce further the exponents, although not below 2/7. In particular, we solve 4-SUM in quantum time $\widetilde{\mathcal{O}}\left(2^{7n/24}\right)$, below the previous $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$ [5].

Organization of the Paper. We define the problem and present *classical* merging algorithms in Section 2. In Section 3, we give some brief preliminaries of quantum computing. In Section 4, we introduce our new definition of *merging trees.* In Section 5, we explain how the trees are extended to the Single-solution case, and we give some of our new results. Next, in Section 6, we show how to obtain our best exponents with quantum walk algorithms for claw-finding.

2 Classical Algorithms for Many-solutions k-XOR

In this paper, we use the term "k-XOR" to refer to a simple variant of Wagner's Generalized Birthday Problem, where the data is generated by a single random function h. Note that, since h is random, a solution might not exist. We include this as a case of failure in our algorithms, as we only require them to succeed on average. We name "k-SUM" the problem where the n-bit bitwise XOR ($\oplus$) is replaced by addition modulo 2^n ($+$). Other extensions are possible provided that *merging* is properly defined, as shown by Wagner [31].

Problem 1 (Many-solutions k-XOR). Given oracle access to a random function $h : \{0,1\}^n \to \{0,1\}^n$, find distinct inputs $(x_1, \ldots, x_k)$ such that $h(x_1) \oplus \ldots \oplus h(x_k) = 0$.

We will assume that *quantum* access to h is given. By restricting the domain of h to $\{0,1\}^{n/k}$, we obtain the *Single-solution* case (a single solution on average). Here having quantum access to h is not a strong restriction, because the time complexity of the best algorithms will exceed $2^{n/k}$, so we can query the whole function classically, store its table, and emulate quantum access to its contents.

Query Complexity. The classical query complexity of the k-XOR problem, Single- or Many-solutions, is $\Omega(2^{n/k})$. The quantum query complexity was determined to be $\Omega(2^{n/(k+1)})$ by Belovs and Spalek [4] in the Single-solution case and by Zhandry [32] in the Many-solutions case.

Time Complexity. The time complexity of the k-XOR problem is also exponential in k. We will write it in the form $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ or $\mathcal{O}(2^{\alpha_k n})$ where the exponent α_k depends only on k. The polynomial factors will be constant or logarithmic. All the quantum algorithms that we will present are composing Grover's quantum search algorithm [18] and MNRS quantum walks [25], which achieve at most a quadratic speedup. So this is the best we can expect.

2.1 Classical Merging

We adopt the following conventions: lists named $\mathcal{L}_i$ have corresponding sizes $L_i = 2^{\ell_i n}$ (up to a constant). We write for simplicity that $\mathcal{L}_i$ "has size ℓ_i". All these parameters ℓ_i are constants.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two lists of n-bit strings selected uniformly and independently at random, of respective sizes $L_1 \simeq 2^{\ell_1 n}$ and $L_2 \simeq 2^{\ell_2 n}$. We assume that they are sorted. We select a prefix t of un bits ($u < 1$), where un is approximated to an integer. By *merging* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *with prefix* t, we say that we compute the *join* list $\mathcal{L}_1 \bowtie_t \mathcal{L}_2$ of pairs (x_1, x_2) such that $x_1 \in \mathcal{L}_1, x_2 \in \mathcal{L}_2, x_1 \oplus x_2 = t|*$. We say that such x_1 and x_2 *partially collide* on un bits. The join list is expected to keep track of the values of x_1 and x_2 that led to $x_1 \oplus x_2$, but we omit them for clarity.

Until Section 4 included, the prefixes will have arbitrary values. In that case, we care only about the parameter u and we use the notation $\bowtie_u$. The notation $\bowtie_t$, with the actual value of the prefix, will be used in Section 5 and Section 6.

The merging operation is efficiently computed by iterating through the lists to retrieve the partial collision pairs. The result is a list of *average* size $\frac{L_1 L_2}{2^{un}}$. Indeed, when $x_1 \in \mathcal{L}_1$ and $x_2 \in \mathcal{L}_2$ are selected uniformly at random, then $\Pr(x_1 \oplus x_2 = t|*) = 2^{-un}$. By linearity of the expectation, the average time complexity of algorithms based on merging is easy to compute. The variance is a more difficult problem, which was first studied by Minder and Sinclair [26, Section 4].

In this paper, we consider the following heuristic, which is enough to ensure the correctness of our algorithms. We show how to remove it from our algorithms in the XOR case, up to a polynomial increase in time, in Appendix B.

Heuristic 1. *If $\mathcal{L}_1$ and $\mathcal{L}_2$ have uniformly random elements, then so does the join $\mathcal{L}_u$ (with the constraint on un bits).*

Lemma 1 (Classical merging, adapted from [31]). *The join list $\mathcal{L}_u = \mathcal{L}_1 \bowtie_u \mathcal{L}_2$ can be computed in time $\max(\ell_1 + \ell_2 - u, \min(\ell_1, \ell_2))$ (in $\log_2$). This list is of size L_u, which has an expectation: $\mathbb{E}(L_u) = \frac{L_1 L_2}{2^{un}}$. Under Heuristic 1, the deviation from $\mathbb{E}(L_u)$ is exponentially small.*

2.2 Wagner's Algorithm

Wagner's algorithm starts from lists of pairs $(x, h(x))$ for many arbitrary values of x, and merges recursively the lists pairwise with increasing zero-prefixes, until a tuple of k elements with a full-zero sum of images is found. This *merging strategy* is best represented as a *merging tree*. It is a binary tree where each node represents an intermediate list of ℓ-tuples with a given size and prefix constraint on the sum. The example of $k = 4$ is given in Figure 1.

We name *merging algorithms* the class of classical algorithms that are represented by valid merging trees. That is, the root node should have prefix length n and expected size 1, and all intermediate nodes have parameters constrained by the formula of Lemma 1. For any merging tree, there exists a k-XOR algorithm

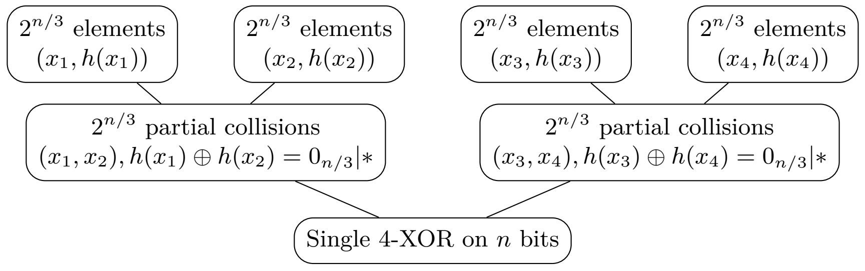


Fig. 1. Structure of Wagner's 4-XOR tree.

with time and memory complexities equal to the maximum of list sizes in the tree.

In the context of Wagner's algorithm, if k is not a power of 2, $k - 2^{\lfloor \log_2(k) \rfloor}$ degrees of freedom are left unused. The tree has $2^{\lfloor \log_2(k) \rfloor}$ prefixless leaves of size $2^{\frac{n}{\lfloor \log_2(k) \rfloor + 1}}$ (single elements obtained by querying h). At subsequent levels, lists are merged pairwise on $\frac{n}{\lfloor \log_2(k) \rfloor + 1}$ bits, so they remain of size $2^{\frac{n}{\lfloor \log_2(k) \rfloor + 1}}$. The final level merges on $\frac{2n}{\lfloor \log_2(k) \rfloor + 1}$ bits to obtain a single solution on average. The total complexity exponent is $\frac{1}{\lfloor \log_2(k) \rfloor + 1}$.

3 Quantum Preliminaries

In this paper, we assume basic knowledge of quantum computing, such as qubits, quantum states, ket notations $|\cdot\rangle$. However, we stress that we will only use well-known algorithmic tools such as quantum search in a black-box way (especially since we consider asymptotic complexities).

Aside from a few exceptions, the known quantum speedups for k-XOR [17, 27], including those of this paper, require some *quantum RAM* (qRAM) model. We will use:

- Classical memory with quantum random-access (QRACM): it contains classical data, but *superposition access* is allowed. Assuming that the data bits are indexed by $1 \leq i \leq 2^m - 1$, a unit cost qRAM gate is given:

$$|i\rangle |y\rangle \xmapsto{\mathsf{qRAM}} |i\rangle |y \oplus M_i\rangle \quad , \text{ where } M_i \text{ is the data at index } i.$$

  That is, all memory cells can be accessed simultaneously in superposition.
- Quantum memory with quantum random-access (QRAQM): it also allows superposition access, but the data can be a quantum state:

$$|i\rangle |y\rangle |M_0 \cdots M_{2^m-1}\rangle \xmapsto{\mathsf{qRAM}} |i\rangle |y \oplus M_i\rangle |M_0 \cdots M_{2^m-1}\rangle \quad .$$

The QRACM/QRAQM terminology is borrowed from [23], and corresponds to QACM/QAQM in [27]. Both are ubiquitous in quantum algorithms, although

QRACM is often regarded as much more reasonable than QRAQM [23]. The qRAM gate is defined in [1, Section 6.1]. We will also briefly consider algorithms *without* qRAM, using purely classical storage instead. A memory of size M is then accessed in time $\widetilde{\mathcal{O}}(M)$ using a sequential circuit.

Quantum Search. Grover's quantum search [18] is one of the most well-known quantum algorithms. We will actually make use of Amplitude Amplification, a powerful generalization proposed by Brassard et al. [7]. It speeds up the search for a "good" output of *any* probabilistic algorithm.

Theorem 1 ([7], Theorem 2). *Let $\mathcal{A}$ be a quantum algorithm that uses no intermediate measurements, let $f : X \to \{0,1\}$ be a boolean function that tests if an output of $\mathcal{A}$ is "good" and assume that a quantum oracle O_f for f is given: $|x\rangle|0\rangle \xmapsto{O_f} |x\rangle|f(x)\rangle$. Let $\theta_a = \arcsin\sqrt{a}$. Then there exists an algorithm running in time: $\left\lfloor \frac{\pi}{4\theta_a} \right\rfloor (2|\mathcal{A}| + |O_f| + \mathcal{O}(\log|X|))$ that obtains a good result with success probability greater than $\max(1-a, a)$.*

We can define a *quantum sampling* black-box, analogous to a classical algorithm which would sample uniformly at random from some well-defined set. We use the **Sample** keyword to write down such quantum algorithms in a simple way, by using recursively the conversion given by Theorem 1. We just have to define a search space and a testing function (the inside of the **Sample** block, which may itself contain another **Sample**).

Definition 1. *Let X be a set. A quantum sampling algorithm for X (denoted $\mathsf{qSample}(X)$) is a quantum algorithm that takes no input and creates the uniform superposition of elements of X (that is, of basis states uniquely representing the elements of X).*

4 Quantum Algorithms for Many-solutions k-XOR

The representation of Wagner's algorithm as a *merging tree* does not make any assumption on the *order* in which the algorithm computes the lists. The tree can be traversed breadth-first, in which case the merging algorithm computes all leaves, then all nodes of depth $\lfloor \log_2(k) \rfloor - 1$, then all nodes of depth $\lfloor \log_2(k) \rfloor - 2$, *etc.* A more interesting option is to traverse it *depth-first.* This well-known technique reduces the storage from $2^{\lfloor \log_2(k) \rfloor}$ to $\lfloor \log_2(k) \rfloor$ lists.

This depth-first traversal actually rewrites the k-XOR algorithm as a sequence of classical Sample procedures. If the list $\mathcal{L}$ is a leaf node, then $\mathsf{Sample}(\mathcal{L})$ consists in making an arbitrary query to h and returning $(x, h(x))$. Otherwise, we use recursively a result equivalent to Lemma 1:

Lemma 2. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two lists of respective sizes $2^{\ell_1 n}$ and $2^{\ell_2 n}$, with $\mathcal{L}_2$ stored in memory, and $\mathcal{L}_u = \mathcal{L}_1 \bowtie_u \mathcal{L}_2$ be the join list with an arbitrary prefix of un bits. Let $\mathsf{Sample}(\mathcal{L}_1)$ be a sampling algorithm for $\mathcal{L}_1$. Then there exists a sampling algorithm $\mathsf{Sample}(\mathcal{L}_u)$ with average complexity:*

$$\mathsf{T_c}(\mathsf{Sample}(\mathcal{L}_u)) = \mathsf{T_c}(\mathsf{Sample}(\mathcal{L}_1) + \mathcal{O}(n)) \cdot \max(2^{(u-\ell_2)}, 1) \ . \tag{1}$$

Proof. The algorithm consists in sampling $x_1 \in \mathcal{L}_1$, and searching an element $x_2 \in \mathcal{L}_2$ such that $x_1 \oplus x_2$ has the right prefix. We repeat this until such an element is found. □

Although this rewriting does not change the classical time complexity, nor the correctness of the algorithm, it leads to the definition of *quantum merging algorithms* in [27]: each Sample can be replaced by a quantum algorithm $\mathsf{qSample}$, using quantum search. Now, any merging tree does not only represent a classical algorithm for k-XOR, but also a quantum one. Unfortunately, the trees defined in [27] are multiary and more complex than those used classically. We will provide in Section 4.2 a simpler definition that goes back to these binary trees.

4.1 Merging in the Quantum Setting

Quantum merging algorithms are based on a result analogous to Lemma 2: if the list $\mathcal{L}_2$ is given, then from a quantum algorithm that samples from the list $\mathcal{L}_1$, we can create another that samples from $\mathcal{L}_u$.

Lemma 3 (Quantum merging). *Let t be an arbitrary prefix of un bits. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two lists of respective sizes $2^{\ell_1 n}$ and $2^{\ell_2 n}$. Assume that $\mathcal{L}_2$ is stored either in QRACM or in classical memory.*

Assume that we are given a quantum sampling algorithm $\mathsf{qSample}(\mathcal{L}_1)$ for $\mathcal{L}_1$. Then there exists a quantum sampling for $\mathcal{L}_u = \mathcal{L}_1 \bowtie_u \mathcal{L}_2$ with quantum time complexity:

$$\mathsf{T_q}(\mathsf{qSample}(\mathcal{L}_u)) = \begin{cases} (\mathsf{T_q}(\mathsf{qSample}(\mathcal{L}_1)) + \mathcal{O}(n)) \cdot \max(2^{\frac{(u-\ell_2)}{2}n}, 1) \text{ \textit{with QRACM}} \\ \left(\mathsf{T_q}(\mathsf{qSample}(\mathcal{L}_1)) + 2^{\ell_2 n}\right) \cdot \max(2^{\frac{(u-\ell_2)}{2}n}, 1) \text{ \textit{without}} \end{cases} \quad (2)$$

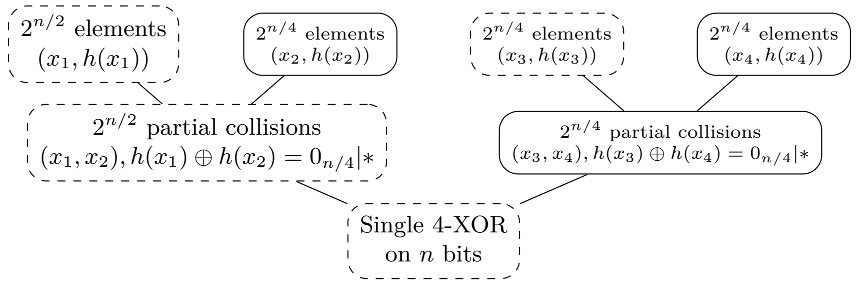
in qRAM gates and n-qubit register operations.

Proof. We use an Amplitude Amplification, where the amplified algorithm consists in sampling $\mathcal{L}_1$, finding whether there is a match of the given prefix in $\mathcal{L}_2$, and returning the pair if it exists. Using Heuristic 1 ensures an exponentially low error for the full procedure. Indeed, this error depends on the difference between the average number of solutions (which dictates the number of search iterations) and the actual one.

To obtain the time complexity, we separate two cases: if $u > \ell_2$, then the amplification really needs to take place, and it has $2^{(u-\ell_2)n/2}$ iterations up to a constant. Each iteration calls $\mathsf{qSample}(\mathcal{L}_1)$ and queries the memory. Without QRACM, we use a circuit that performs a sequence of $2^{\ell_2 n}$ classically controlled comparisons (hence the additional term).

If $u < \ell_2$, then a given element $x_1 \in \mathcal{L}_1$ will have on average exponentially many $x_2 \in \mathcal{L}_2$ such that $x_1 \oplus x_2 = t|*$. It is possible to return the superposition of them at no greater time cost, by organizing the QRACM in a radix tree.

Each node of the tree is indexed by a prefix, with the node of prefix t having $t|0$ and $t|1$ as children, and leaves are the actual elements stored. To query



$2^{n/2}$ elements $(x_1, h(x_1))$
$2^{n/4}$ elements $(x_2, h(x_2))$
$2^{n/4}$ elements $(x_3, h(x_3))$
$2^{n/4}$ elements $(x_4, h(x_4))$
$2^{n/2}$ partial collisions $(x_1, x_2), h(x_1) \oplus h(x_2) = 0_{n/4}|*$
$2^{n/4}$ partial collisions $(x_3, x_4), h(x_3) \oplus h(x_4) = 0_{n/4}|*$
Single 4-XOR on n bits

Fig. 2. Re-optimization of 4-XOR merging (from [27]). Plain lines indicate the lists actually stored in QRACM.

an element, we move down the tree with $\mathcal{O}(n)$ qRAM gates. To construct the uniform superposition over a whole subtree, we move in superposition in both the left and right branches (weighted by the number of leaves in both subtrees). □

Because we are now using quantum search, the balanced trees such as Wagner's are not suitable anymore, and we must re-optimize the parameters. The example of 4-XOR, which reaches a time complexity $\mathcal{O}\left(2^{n/4}\right)$, is displayed in Figure 2. In this algorithm, we first built the two intermediate lists of size $2^{n/4}$, then find the 4-XOR with an exhaustive search in the $2^{n/2}$-sized leaf list. This list is not written down, as it only corresponds to a search space efficiently sampled. Given a random $(x, h(x))$, we find a partial collision with the first intermediate list, moving to the next level. Then we try to match against the second intermediate list. Both operations require memory access to the lists, which becomes QRACM access when the search is done quantumly.

4.2 Definition of Merging Trees

The goal of *merging trees* is to represent quantum merging strategies for k-XOR in a purely syntactical way. Though we use the same name as [27], our definition will largely differ.

Definition 2 (Merging trees). *A k-merging tree $\mathcal{T}_k$ is a binary tree defined recursively as follows:*

- *A node is either labeled "Sample" (S-node) or "List" (L-node)*
- *If $k = 1$, this is a leaf node $\mathcal{T}_1$*
- *If $k > 1$, $\mathcal{T}_k$ has two children: an S-node $\mathcal{T}_{k_l}$ and an L-node $\mathcal{T}_{k_r}$, where $k_l + k_r = k$.*

It follows inductively that a k-merging tree has k leaf nodes. Intuitively, an S-node represents a procedure that samples from a given list and an L-node represents a list stored in memory, constructed with exponentially many samples.

By convention, we draw Sample nodes (dashed) on the left and List nodes (plain) on the right, as in Figure 2. To each node $\mathcal{T}$ corresponds a list $\mathcal{L}$ which is either *built* or *sampled*. Since the trees are binary, we adopt a simple numbering of lists $\mathcal{L}_i^j$. The root node, at level 0 in the tree, is $\mathcal{L}_0^0$, and the two children of $\mathcal{L}_i^j$ are numbered respectively $\mathcal{L}_{2i}^{j+1}$ for the sampled one and $\mathcal{L}_{2i+1}^{j+1}$ for the list one. We label a node with the following variables describing $\mathcal{L}_i^j$:

- The *width* k_i^j
- The number u_i^j of bits set to zero (relatively to n)
- The size ℓ_i^j of this list: by our conventions, ℓ_i^j represents a size of $2^{\ell_i^j n}$

Thus, $\mathcal{L}_i^j$ is a list of k_i^j-tuples $(x_1, \ldots, x_{k_i^j})$ such that $x_1 \oplus \ldots \oplus x_{k_i^j} = 0_{u_i^j n}|*$, of size $2^{\ell_i^j n}$, which is only stored in memory if i is odd, and otherwise, represents a *search space*.

Merging strategy and constraints. We constrain the variables ℓ_i^j and u_i^j in order to represent a valid merging strategy. First, we want a solution to the k-XOR problem.

Constraint 1 (Root node). *At the root node: $u_0^0 = 1$ and $\ell_0^0 = 0$.*

As each node results from the merging of its two children, the number of zeros increases. Furthermore, two siblings shall have the same number of zeros: $u_{2i}^j = u_{2i+1}^j$. Otherwise, we could reduce this parameter to $\min(u_{2i}^j, u_{2i+1}^j)$.

Constraint 2 (Zero-prefixes). $\forall i, j \geq 1, u_{2i}^j = u_{2i+1}^j$ *and* $u_i^{j-1} \geq u_{2i}^j$.

Finally, the size of a list is constrained by the sizes of its predecessors and the new constraint ($(u_i^{j-1} - u_{2i+1}^j)n$ more bits to put to zero).

Constraint 3 (Size of a list). $\forall i, j \geq 1, \ell_i^{j-1} = \ell_{2i}^j + \ell_{2i+1}^j - (u_i^{j-1} - u_{2i+1}^j)$.

Possible extensions of this framework are discussed in [27]. None of the classical techniques of [12, 2, 13, 28, 26] seem to bring further improvements to the k-XOR problem in the quantum setting.

4.3 From Trees to Algorithms

We attach to each node another parameter t, which represents the *sample time*. Our intuition is that the time to sample from the list $\mathcal{L}_i^j$ represented by this node will be $\widetilde{\mathcal{O}}(2^{nt})$.

Constraint 4 (Sampling). *Let $\mathcal{T}_i^j$ be a node in the tree, either an S-node or an L-node. If $\mathcal{T}_i^j$ is a leaf, $t_i^j = \frac{u_i^j}{2}$. Otherwise, $\mathcal{T}_i^j$ has an S-child $\mathcal{T}_{2i}^{j+1}$ and an L-child $\mathcal{T}_{2i+1}^{j+1}$, and:*

$$t_i^j = \begin{cases} t_{2i}^{j+1} + \frac{1}{2}\max\left(u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0\right) & \textit{with QRACM} \\ \max(t_{2i}^{j+1}, \ell_{2i+1}^{j+1}) + \frac{1}{2}\max\left((u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0\right) & \textit{without} \\ t_{2i}^{j+1} + \max\left(u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0\right) & \textit{classically} \end{cases} \quad (3)$$

If the node is a leaf, then we simply run Grover's algorithm multiple times. Equation (3) is simply a translation of (2) in the case of a specific node. The third option needs to be added when QRACM is not available, in order to model a situation where the best thing to do is to sample the list classically. If we do that, then the whole branch (from this node to the root) becomes classical. Next, we can deduce the time complexity exponent of a tree.

Definition 3. *Let $\mathcal{T}_k$ be a k-merging tree. We define $\mathsf{T_q}(\mathcal{T}_k)$ and $\mathsf{M}(\mathcal{T}_k)$ as:*

$$\mathsf{T_q}(\mathcal{T}_k) = \max\left(\max_{\textit{List nodes}}\left(t_i^j + \ell_i^j\right), t_0^0\right) \text{ and } \mathsf{M}(\mathcal{T}_k) \quad = \max_{\textit{List nodes}}\left(\ell_i^j\right) \; .$$

It should be noted that the list size of Sample nodes plays only a role in the structural constraints, not in the time complexity. They should simply have a size sufficient to ensure the existence of a solution in the tree. Thanks to Lemma 3, we can prove that to any merging tree satisfying the constraints, there corresponds a quantum merging algorithm.

Theorem 2 (Quantum merging strategies). *Let $\mathcal{T}_k$ be a k-merging tree and $\mathsf{T_q}(\mathcal{T}_k)$ computed as in Definition 3. Then there exists a* quantum merging algorithm *that, given access to a quantum oracle for h, finds a k-XOR.*

Under Heuristic 1, this algorithm succeeds with probability more than $1 - e^{-an}$ for some constant $a > 0$. It runs in time $\mathcal{O}\left(n2^{\mathsf{T_q}(\mathcal{T}_k)n}\right)$, makes the same number of queries to h. It requires only $\mathcal{O}(n)$ computing qubits. It uses a memory $\mathcal{O}\left(2^{\mathsf{M}(\mathcal{T}_k)n}\right)$, counted in n-bit registers (either classical or QRACM).

Proof. We define recursively the correspondence $\mathcal{T}_k \overset{\mathcal{A}}{\longmapsto} \mathcal{A}(\mathcal{T}_k)$ from a merging tree $\mathcal{T}_k$ to a k-XOR algorithm $\mathcal{A}(\mathcal{T}_k)$. The complexities follow from Lemma 3 and simplifying $\mathcal{O}(2^{\alpha n}) + \mathcal{O}\left(2^{\beta n}\right) = \mathcal{O}\left(2^{\max(\alpha,\beta)n}\right)$. A global factor $\mathcal{O}(n)$ comes from the memory operations.

Let $N(k, u, \ell)$ be the root node of $\mathcal{T}_k$ and $S(k', u', \ell')$ and $L(k'', u'', \ell'')$ its two children, if they exist.

- If it is a Sample leaf, then $\mathcal{A}(\mathcal{T}_k)$ simply consists in running Grover's algorithm in time $\mathcal{O}\left(2^{un/2}\right)$.
- Otherwise, if it is a Sample: • we sample from the child L with $\mathcal{A}(L)$, and build the list in time $\mathcal{O}\left(2^{(\mathsf{T_q}(L)+\ell'')n}\right)$. • we apply Lemma 3, using $\mathcal{A}(S)$ as a sample for the child S.
- If it is a List, the situation is the same, except that we repeat the operation exponentially many times. □

4.4 Optimal Trees for Many-solutions k-XOR

Now that we have defined the set of merging trees, we can explore this space to search for the trees $\mathcal{T}_k$ that minimize $\mathsf{T_q}(\mathcal{T}_k)$.

Given a tree $\mathcal{T}_k$, its time and memory complexity exponents $\mathsf{T_q}(\mathcal{T}_k)$ and $\mathsf{M}(\mathcal{T}_k)$ are defined as the maximums of linear combinations of the parameters

ℓ_i^j, u_i^j. Thus, there exists a choice of these parameters that minimizes $\mathsf{T_q}(\mathcal{T}_k)$, under Constraint 1, 2, 3 and 4. As remarked in [27], this is a linear problem, solvable with Mixed Integer Linear Programming (MILP). Given k, we try all possible tree structures and find the optimal one. Note that thanks to our new definition of merging trees, we have a much smaller set of tree shapes to explore than in [27].

There always exists an optimal tree $\mathcal{T}_k$ that achieves the best time complexity exponent. For a given k, there is sometimes more than one, but we find that it is reached by a family of *balanced* trees T_k. When k is a power of 2, T_k is Wagner's balanced binary tree.

Definition 4 (Trees T_k). *If $k = 1$, then T_k is simply a leaf node. If $k = 2k'$, then the "Sample" child of T_k is $T_{k'}$ and the "List" child is $T_{k'}$. If $k = 2k' + 1$, then the "Sample" child of T_k is $T_{k'+1}$ and the "List" child is $T_{k'}$.*

When QRACM is available, the authors of [27] find a complexity exponent $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$ for any k, where $\kappa = \lfloor \log_2(k) \rfloor$. When QRACM is not available, and $k \geq 8$, they find $\beta_k = \frac{1}{\kappa+1}$ if $k < 2^\kappa + 2^{\kappa-1}$ and $\beta_k = \frac{2}{2\kappa+3}$ if $k \geq 2^\kappa + 2^{\kappa-1}$. In the latter case, the strategies for 2, 3, 5, 7 reach respectively $\beta_2 = \frac{2}{5}$ (see [11]), $\beta_3 = \frac{5}{14}$ (see [17]), $\beta_5 = \frac{14}{45}$ and $\beta_7 = \frac{2}{7}$. Thanks to our rewriting of the constraints, we are able to improve to $\beta_5 = \frac{40}{129}$ and $\beta_7 = \frac{15}{53}$. Our new optimality proofs are given in Appendix C and D.

5 Quantum Algorithms for Single-solution k-XOR

The algorithms of Section 4 solve the Many-solutions case (Problem 1). Following again the study in [27], we extend the merging trees to target the Single-solution case. In this section, we assume QRAQM.

When only a few solutions are to be found, *merging* does not seem to help at first sight, since it puts more constraints on the solution tuples. However, an interesting idea is to merge with arbitrary constraints, e.g., by choosing a prefix t, and to repeat this for every value of t. Obviously, the set of all merging trees obtained by looping on the value of t contains all k-tuples of elements, so the solution cannot be missed.

This is the core idea of Schroeppel and Shamir's 4-SUM algorithm [30] (see Algorithm 1 and Figure 3) and more generally, the *Dissection* algorithms of [13, Section 3]. In the quantum setting, it encompasses some proposed algorithms such as the element distinctness (Single-solution 2-XOR) algorithm of [9].

The classical algorithms are intended to decrease the memory usage while keeping the time equal or close to the classical birthday bound $\mathcal{O}(2^{n/2})$. In contrast, the quantum algorithms allow to *decrease* the time complexity with respect to the quantum birthday bound $\mathcal{O}(2^{n/3})$, as shown in [5, 27].

Algorithm 1 Schroeppel and Shamir's algorithm, based on a repetition loop.

```
    Input: oracle access to $h\ : \{0,1\}^{n/4} \to \{0,1\}^n$
    Output: $x_1, x_2, x_3, x_4$ such that $h(x_1) \oplus h(x_2) \oplus h(x_3) \oplus h(x_4) = 0$
1: Create 4 lists $\mathcal{L}_i, 0 \le i \le 3$, of size $2^{n/4}$, where pairs $x, h(x)$ have arbitrary indices
2: for all Prefix $s$ of $\frac{n}{4}$ bits do
3:     $\mathcal{L}_{01}^s \leftarrow$ MERGE($\mathcal{L}_0, \mathcal{L}_1, s$)        ▷ $\mathcal{L}_{101}^s$ is of average size $\frac{2^{n/4} \times 2^{n/4}}{2^{n/4}} = 2^{n/4}$
4:     $\mathcal{L}_{23}^s \leftarrow$ MERGE($\mathcal{L}_2, \mathcal{L}_3, s$)
5:     if there is a collision between $\mathcal{L}_{01}$ and $\mathcal{L}_{23}$ then
                                    ▷ Happens for a single $s$ (or with probability $2^{-n/4}$)
6:         return The collision
```

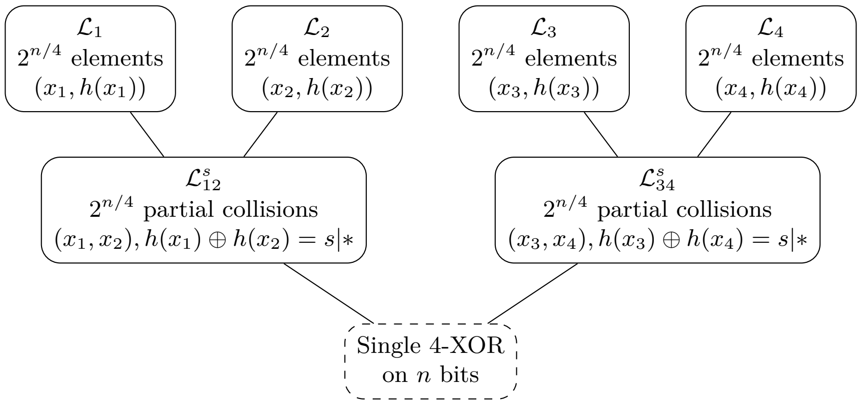


Fig. 3. Structure of Schroeppel and Shamir's merging tree.

5.1 Extended Merging Trees

The *extended merging trees* that we use in this paper subsume those given in [27], with a technical trick that will allow smaller complexities. The optimal strategies turn out to have a very simple description. Thus, we defer their derivation to Appendix E.1, and we focus here on the actual algorithms.

A merging tree is now extended with *repetition loops*. We make the selection of some arbitrary prefixes, or more generally, sublists of list nodes. This choice defines *a subset of the merging tree*. We complete the merging process. If a solution is obtained, then this choice of subset was good. These repetition loops are performed with quantum searches.

Note that in Section 4, we only needed QRACM, as all intermediate lists could be written down classically, and quantum access was only necessary to sample their elements. Here, we need to write down lists *under a quantum search*, which is why QRAQM is necessary.

Remark 1 (Amending the constraints). Our improved complexities with respect to [27] rely on the following idea. A subtree $\mathcal{T}^j$ of width k^j can cost 0 inside the repetitions if a global cost $2^{\frac{k^j}{k}n}$ (in time and memory) has been already paid. Indeed, when $\mathcal{T}^j$ is of width 1, a full lookup table of h can be prepared beforehand and reused instead of having to rebuild the tree in each search iteration. Likewise, we can prepare the sorted list of all k^j-tuples (which is of size $\left(2^{\frac{n}{k}}\right)^{k^j}$) in order to retrieve quickly those having a wanted prefix.

5.2 New Results for Single-solution k-XOR

Remark 1 allows us to reach better exponents than [27], and to break the previous lower bound of 0.3 for k-list merging.

Theorem 3 (New trees for single-solution k-XOR). *Let $k > 2$ be an integer. The best extended merging tree (with our definition) finds a k-XOR in time $\mathcal{O}(2^{\gamma_k n})$ where:*

$$\gamma_k = \frac{k + \left\lfloor \frac{k+6}{7} \right\rfloor + \left\lfloor \frac{k+1}{7} \right\rfloor - \left\lfloor \frac{k}{7} \right\rfloor}{4k} \quad . \tag{4}$$

In particular, γ_k converges towards a minimum $\frac{2}{7}$, reached by multiples of 7.

The proof of this optimality is given in Appendix E. The formula of Theorem 3 comes from the reduction of the constraints to a simple linear optimization problem with two integer variables. These variables are sufficient to describe the shape of the corresponding tree.

Optimal Trees. For $k \leq 5$, the results of Theorem 3 and [27] coincide and we can refer to [27]. The novelty of Theorem 3 appears with Algorithm 2 (Figure 4), whose total time complexity is, up to a constant:

$$\underbrace{2^{2n/7}}_{\substack{\text{Building } \mathcal{L}_{34} \\ \text{and } \mathcal{L}_{67}}} + \underbrace{2^{n/7}}_{\text{Search of } s} \left(\underbrace{2^{n/7}}_{\text{Computing } \mathcal{L}_{567}} + \underbrace{2^{n/7}}_{\text{Search in } \mathcal{L}_{12}} \right) = \mathcal{O}\left(2^{2n/7}\right) \quad .$$

It benefits from computing some products of lists *outside the loops.* Interestingly, this also modifies the memory requirements: only $2^{n/7}$ QRAQM is required, in order to hold $\mathcal{L}_{567}$, and $2^{2n/7}$ *QRACM* is needed for $\mathcal{L}_{34}$ and $\mathcal{L}_{67}$.

The optimal strategy for a bigger k actually mimics the 7-XOR example. We introduce two integer variables k_1 and k_2 with the values:

$$k_1 = \left\lfloor \frac{3k}{7} \right\rceil \ , \qquad k_2 = \left\lfloor \frac{2k}{7} \right\rceil - \left\lfloor \frac{k-1}{7} \right\rceil + \left\lfloor \frac{k-2}{7} \right\rceil \ , \tag{5}$$

where $\left\lfloor \frac{3k}{7} \right\rceil$ is the integer closest to $\frac{3k}{7}$, and we perform Algorithm 3. The tree structure (Figure 5) is overly simple: there are four subtrees, each of which is a trivial product of lists (a merge with empty prefixes). There is only a single repetition loop, and the whole algorithm contains only two levels of quantum search. Intuitively, the subtrees end up being "trivial" because enforcing a prefix of length un would induce a new quantum search loop with $2^{un/2}$ iterates (all prefixes must be searched). Instead, a better strategy is to forget the prefix and pay the cost $2^{un/2}$ later on, when sampling the parent. A formal proof is given in Appendix E. The fact that this choice of structure matches the complexity given by Theorem 3 also follows from it.

Memory Complexity. Our algorithms for single-solution k-XOR reach the best time complexity $\mathcal{O}\big(2^{2n/7}\big)$ when k is a multiple of 7, but at these points, they require a QRACM of size $2^{2n/7}$. This is suboptimal with respect to the time-memory product. By optimizing for it, we obtained the same results as [27]. The experiments suggest that the list sizes in the tree never exceed $2^{n/k}$ in that case. For trees with a list size fixed to $2^{n/k}$, we observe that the best time-memory product decreases for small k, reaches a minimum at $k = 17$ with $\widetilde{\mathcal{O}}(2^{\frac{7}{17}n})$, and increases again later, as it behaves like $(k - \mathcal{O}(\sqrt{k}))/2$. More details are provided in Appendix F.

On Memory Models. The balance between QRACM and QRAQM is interesting here, since in general, we will use more QRACM than QRAQM. An interesting question is whether we can completely eliminate QRAQM. In this setting, the best procedure remains to cut the lists in three complete products of equal size, and do a quantum search on two groups for a match on the third one. This converges towards $\widetilde{\mathcal{O}}\big(2^{n/3}\big)$ and this complexity is reached for multiples of 3.

Another interesting question is whether one can get rid of quantum random access, and use only plain quantum circuits. In this setting, the best algorithm for Single-solution 2-XOR runs in time $\widetilde{\mathcal{O}}\big(2^{3n/7}\big)$ using $\mathcal{O}\big(2^{n/7}\big)$ qubits [20]. We can propose an improved complexity for 4-XOR with the following: we use Schroeppel and Shamir's merging tree. We perform a quantum search on the right prefix s ($2^{n/8}$ iterates). At each iterate, we compute the merging tree breadth-first, without qRAM gates, using sorting networks for the merging operations. This costs time $\widetilde{\mathcal{O}}\big(2^{n/4}\big)$ (with a polynomial factor from sorting networks). The total time is $\widetilde{\mathcal{O}}\big(2^{3n/8}\big)$ which is smaller than $\mathcal{O}\big(2^{3n/7}\big)$. It might be possible to improve on this with a more generic method.

Algorithm 2 New Single-solution 7-XOR algorithm. As a quantum algorithm, each **Sample** becomes a quantum search.

```
    Input: 7 lists $\mathcal{L}_i$ of size $2^{n/7}$
    Output: a 7-tuple $(x_i) \in \prod_i \mathcal{L}_i$ that XORs to 0
 1: Build $\mathcal{L}_{67} = \mathcal{L}_6 \bowtie_0 \mathcal{L}_7$ (all sums between these two lists)
 2: Build $\mathcal{L}_{34} = \mathcal{L}_3 \bowtie_0 \mathcal{L}_4$ (all sums between these two lists)
 3: Sample $s \in \{0,1\}^{2n/7}$                         ▷ $2^{n/7}$ quantum search iterates
 4:     Build $\mathcal{L}_{567} = \mathcal{L}_5 \bowtie_s \mathcal{L}_{67}$           ▷ Time $2^{n/7}$, which is the size of the list
 5:     Sample $x \in \mathcal{L}_1 \times \mathcal{L}_2$                    ▷ $2^{n/7}$ quantum search iterates
 6:         Find $y \in \mathcal{L}_{34}$ such that $x \oplus y = s|*$
 7:         Find $z \in \mathcal{L}_{567}$ such that $x \oplus y \oplus z = 0_{3n/7}|*$
 8:         return “good” if $x \oplus y \oplus z = 0$, “not good” otherwise
 9:     EndSample
10:     return “good” if there is a solution $x$, “not good” otherwise
11: EndSample
```

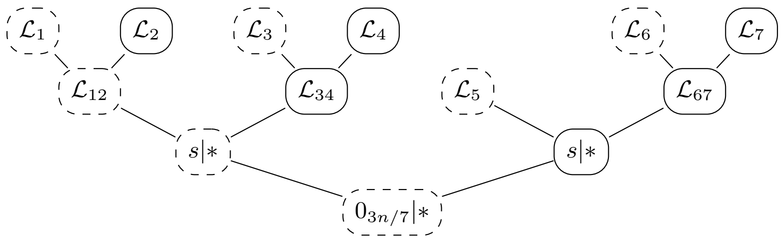


Fig. 4. Single-solution 7-XOR merging tree of Algorithm 2.

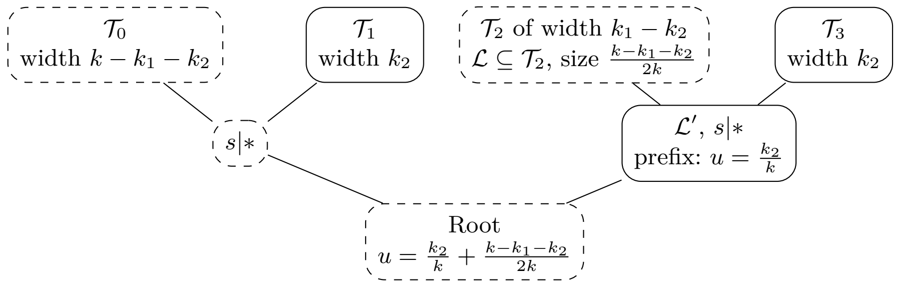

$\mathcal{T}_0$ width $k - k_1 - k_2$
$\mathcal{T}_1$ width k_2
$\mathcal{T}_2$ of width $k_1 - k_2$ $\mathcal{L} \subseteq \mathcal{T}_2$, size $\frac{k-k_1-k_2}{2k}$
$\mathcal{T}_3$ width k_2
$s|*$
$\mathcal{L}'$, $s|*$ prefix: $u = \frac{k_2}{k}$
Root $u = \frac{k_2}{k} + \frac{k-k_1-k_2}{2k}$

Fig. 5. Generic merging tree that reaches the optimal complexity for Single-solution k-XOR (see Algorithm 3).

Algorithm 3 Generic Single-solution k-XOR algorithm. As a quantum algorithm, each **Sample** becomes a quantum search.

```
    Input: $k$ lists $\mathcal{L}_i$ of size $2^{n/k}$
    Output: a $k$-tuple $(x_i) \in \prod_i \mathcal{L}_i$ that XORs to 0
 1: Select $k_1, k_2$ by Equation 5
 2: Build $\mathcal{T}_1$ and $\mathcal{T}_3$, each with the product of $k_2$ lists
 3: Sample $s \in \{0,1\}^{\frac{k_2}{k}n}$
 4:     Sample Sublists $\mathcal{L}$ of $\mathcal{T}_2$ of size $\frac{k-k_1-k_2}{2k}$
 5:         Merge $\mathcal{L}$ with $\mathcal{T}_3$ to obtain a list $\mathcal{L}'$ with prefix $s$ and size $\frac{k-k_1-k_2}{2k}$
 6:         Sample $x \in \mathcal{T}_0$                    $\triangleright$ $2^{\frac{k-k_1-k_2}{2k}n}$ quantum search iterates
 7:             Find $y \in \mathcal{T}_1$ such that $x \oplus y = s|*$
 8:             return "good" if there is a collision with $\mathcal{L}'$, "not good" otherwise
 9:         EndSample
10:         return "good" if there is a solution, "not good" otherwise
11:     EndSample
12:     return "good" if there is a solution, "not good" otherwise
13: EndSample
```

6 Extension with Quantum Walks

The algorithms presented so far are the best ones achievable *in the restricted model of quantum merging trees.* One of the open questions left in [27] was whether it was possible to improve generically the time complexity using quantum walks. We find that this is the case, yielding a better curve than Theorem 3 that we will now explicit. In particular, we obtain the first 4-SUM algorithm with complexity below $\mathcal{O}\left(2^{0.3n}\right)$ (obtained in [5] with a quantum walk).

Theorem 4 (Single-solution k-XOR with quantum walks). *Let $k > 2$ be an integer. There exists a quantum Single-solution k-XOR algorithm running in time $\widetilde{\mathcal{O}}(2^{\gamma_k n})$ where:*

$$\gamma_k = \frac{2k - \left\lfloor \frac{k}{7} \right\rfloor - \left\lfloor \frac{k+3}{7} \right\rfloor}{6k} \quad . \tag{6}$$

In particular, γ_k converges towards a minimum $\frac{2}{7}$, reached by multiples of 7.

6.1 Preliminaries

In this paper, we only need quantum walks to solve the following problem.

Problem 2 (Single claw-finding). Let f, g be two functions of different domains $\{0,1\}^{\ell_1 n}, \{0,1\}^{\ell_2 n}$, that we can query quantumly, with the promise that there exists either a single *claw* (x, y) such that $f(x) = g(y)$, or none. Determine the case and find the claw.

This is an extension of the *element distinctness* problem, or Single-solution 2-XOR, and it can be solved by similar algorithms. In particular, we will consider Ambainis' algorithm [1] which is a quantum walk for element distinctness running in time $\mathcal{O}\left(2^{2\ell n/3}\right)$ when $\ell_1 = \ell_2 = \ell$. We will give some high-level ideas and refer to [1, 5, 21] for applications of quantum walks to k-SUM algorithms.

When there is a single function h, Ambainis' algorithm is a walk on a *Johnson graph*, where a vertex represents a subset of 2^{nr} elements, for some parameter r. We move randomly on the walk by replacing elements, until the vertex contains the wanted collision. The classical time complexity of such a random walk (up to a logarithmic factor) is: $\left(2^{rn} + \frac{2^{2\ell n}}{2^{2rn}}\left(2^{rn}\right)\right)$, where $\frac{2^{2\ell n}}{2^{2rn}}$ is the number of "walk steps" that one should do classically before finding a marked vertex, and 2^r is the number of vertex updates before arriving to a new uniformly random vertex. The corresponding quantum walk algorithm, either in the specific example of Ambainis [1], or the more generic MNRS framework [25], achieves: $2^{rn} + \sqrt{\frac{2^{2\ell n}}{2^{2rn}}}\left(\sqrt{2^{rn}}\right)$, using a quantum memory (QRAQM) of size 2^{rn} and the same number of quantum queries to h.

When there are two functions f, g with domains of different size, we will use a random walk on a *product Johnson graph*, as in [21]. We choose two parameters r_1, r_2; the vertices now contain $2^{r_1 n}$ elements queried to f and $2^{r_2 n}$ elements queried to g, with $r_1 \leq \ell_1$ and $r_2 \leq \ell_2$. The quantum time complexity becomes:

$$2^{r_1 n} + 2^{r_2 n} + \sqrt{\frac{2^{(\ell_1+\ell_2)n}}{2^{(r_1+r_2)n}}}\left(2^{r_1 n/2} + 2^{r_2 n/2}\right) \quad .$$

By symmetry between r_1 and r_2, we can choose $r_1 = r_2 = r$ and restrict ourselves to a single parameter.

Theorem 5 (Adaptation of [1, 21]). *There exists a quantum algorithm solving the single claw-finding problem with domains $\ell_1 n$ and $\ell_2 n$, in time $\widetilde{\mathcal{O}}(2^{\tau n})$ and memory $\mathcal{O}(2^{rn})$, where: $\tau = \max\left(r, \frac{\ell_1+\ell_2-r}{2}\right)$, for any r such that $r \leq \ell_1, r \leq \ell_2, r \geq 0$.*

This algorithm succeeds with constant probability. Up to a polynomial factor, it can be boosted to any probability exponentially close to 1, and thus, used as a subroutine in a quantum search.

6.2 Using Quantum Walks in a Merging Tree

Since we did not include quantum walks in our merging tree framework, it remains an open question whether the algorithms obtained here are the best possible. Our goal is merely to improve on what we presented above, using Theorem 5 as a building block.

We reuse the tree structure of Figure 5, where $\mathcal{T}_0$ to $\mathcal{T}_3$ are the nodes at level 2, which are products of base lists. Thus, we reuse most of the structure of Algorithm 3, except that the parameters will be re-optimized and that the two innermost **Sample** loops are replaced by a single call to **Claw-finding**. This is why we reach an improved time complexity. The new choice of k_1 and k_2 is:

$$k_1 = \begin{cases} \lfloor \frac{k+1}{7} \rfloor + \lfloor \frac{k+4}{7} \rfloor + \lfloor \frac{k+6}{7} \rfloor & \text{for } k \geq 4 \\ 1, 1, 2 \text{ for } k = 2, 3, 4 \text{ respectively} \end{cases} \quad , \qquad k_2 = \left\lfloor \frac{k}{7} \right\rfloor + \left\lfloor \frac{k+4}{7} \right\rfloor \quad . \quad (7)$$

Algorithm 4 Single-solution k-XOR algorithm with a quantum walk.

```
    Input: $k$ lists $\mathcal{L}_i$
    Output: a $k$-tuple $(x_i) \in \prod_i \mathcal{L}_i$ that XORs to 0
1: Select $k_1, k_2$ by Equation (7)
2: Build $\mathcal{T}_1$ and $\mathcal{T}_3$, each with the product of $k_2$ lists
3: Sample $s \in \{0,1\}^{\frac{k_2}{k}n}$
4:     Apply Claw-finding between the lists $\mathcal{T}_0 \bowtie_s \mathcal{T}_1$ and $\mathcal{T}_2 \bowtie_s \mathcal{T}_3$
5:     return "good" if a claw is found, "not good" otherwise
6: EndSample
```

The key idea of Algorithm 4 is that the knowledge of $\mathcal{T}_1$ and $\mathcal{T}_3$, and the constraints of merging, make sure that we can run the quantum walk as expected. That is, we can query $\mathcal{T}_0 \bowtie_s \mathcal{T}_1$ and $\mathcal{T}_2 \bowtie_s \mathcal{T}_3$ in time $\mathcal{O}(1)$.

By definition of k_1 and k_2, the product lists $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ have respective widths $(k - k_1 - k_2)$, k_2, $(k_1 - k_2)$, k_2. Thus, taking into account the quantum search on the right prefix s, and using Theorem 5, we compute the following time complexity for Algorithm 4:

$$2^{\frac{k_2}{2k}n}\left(2^{rn} + 2^{\left(\frac{k-k_1-k_2}{2k} + \frac{k_1-k_2}{2k} - r\right)n} \times 2^{rn/2}\right) + 2^{\frac{k_2}{k}n} \quad , \tag{8}$$

where r is the parameter specifying the size of the vertex. The corresponding QRACM used is $2^{\frac{k_2}{k}n}$, the corresponding QRAQM (for the walks) is 2^{rn}, and the total memory is the maximum between both.

Thus, when k_1, k_2 are free, the time complexity exponent t of Algorithm 4 is solution to the following optimization problem:

$$\begin{array}{lll} \text{(C1) } t \geq \frac{k_2}{2k} + r & \text{(C2) } t \geq \frac{k_2}{k} & \text{(C3) } t \geq \frac{k-k_2}{2k} - \frac{r}{2} \\ \text{(C4) } r \leq \frac{k-k_1-k_2}{k} & \text{(C5) } r \leq \frac{k_1-k_2}{k} & \end{array}$$

Here (C1) and (C3) correspond to the walks, (C2) to the computation of lists $\mathcal{L}_1$ and $\mathcal{L}_3$ *outside* the main loop. (C4) and (C5) are the constraints imposed on our choice of r. Solving this optimization problem gives us the choice of k_1 and k_2 specified by Equation (7), and the time complexity exponent of Theorem 4.

6.3 Results

In Figure 6, we compare Algorithm 4 with the previous work of [27] (where the formula was $\gamma_k = \frac{1}{k}\frac{k+\lceil k/5 \rceil}{4}$) and to the intermediate result of Theorem 3. Numerical results are given in Table 1 in Appendix. Our curve improves or subsumes all previous works on k-XOR (including the special cases of Ambainis' algorithm for $k = 2$ and [5] for 4-SUM).

The algorithm for 4-SUM is very simple. We start from 4 lists. Two of them are stored in QRACM. Then, we do a quantum search over a prefix of $\frac{n}{4}$ bits. In order to find the good one, we search for a claw between the two level-1 lists of size $2^{\frac{n}{4}}$. Thus the complexity is of order: $\sqrt{2^{\frac{n}{4}}} \times 2^{\frac{n}{4} \times \frac{2}{3}} = 2^{7n/24}$.

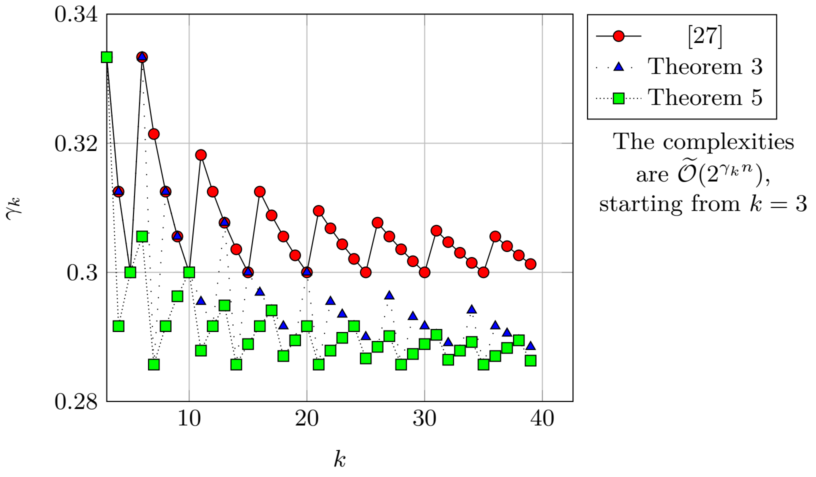


Fig. 6. Single-solution k-XOR time complexity, and comparison with [27].

6.4 Applications

Similarly as those of [27], the algorithms of this paper apply to the class of *bicomposite* problems studied by Dinur et al. [13]. This correspondence is actually easier to see than in [27], because our algorithms have a simple description.

A prominent example of bicomposite search is multiple-encryption, where we search for the key used by a block cipher made of a sequential composition of independent block ciphers.

Problem 3 (r-encryption). Let $E^1, \ldots, E^r$ be r random block ciphers on n bits, indexed by key spaces of the same size 2^n. Assume that we are given r plaintext-ciphertext pairs (p_i, c_i), encrypted by the composition of the E^i under a sequence of independent keys $k_1, \ldots, k_r$:

$$\forall i, c_i = \left(E^r_{k_r} \circ \ldots \circ E^1_{k_1}\right)(p_i), \qquad \text{then retrieve } k_1, \ldots, k_r.$$

Theorem 6. *For any $r \geq 2$, let γ_r be the time complexity exponent given by Theorem 4. Then there exists a quantum algorithm for r-encryption, of time complexity $\mathcal{O}(2^{\gamma_r r n})$.*

In particular, we obtain an algorithm for 4-encryption (Algorithm 5) that runs in time $\widetilde{\mathcal{O}}\left(2^{7n/6}\right)$ for $4n$ bits of key, improving the previous $\widetilde{\mathcal{O}}\left(2^{5n/4}\right)$ [27].

7 Conclusion

In this paper, we simplified the analysis of quantum k-XOR algorithms and improved the previous results for the single-solution case, leading to the best known quantum algorithms for *bicomposite search* and multiple-encryption.

Algorithm 5 4-encryption algorithm.

```
   Input: 4 plaintext-ciphertext pairs $(p_i, c_i)$
   Output: the sequence of 4 keys $k_1, k_2, k_3, k_4$
1: Build the list $\mathcal{L}_1$: $\{E^1_{k_1}(p_1), k_1 \in \{0,1\}^n\}$
2: Build the list $\mathcal{L}_4$: $\{(E^4_{k_4})^{-1}(c_4), k_4 \in \{0,1\}^n\}$
3: Sample $s \in \{0,1\}^n$
4:     Define: $\mathcal{L}_{12}$ the list of all $(k_1, k_2)$ such that $E^1_{k_1}(p_1) = (E^2_{k_2})^{-1}(s)$
               ▷ It is easy to sample from $\mathcal{L}_{12}$, by taking a random key $k_2$, computing
   $(E^2_{k_2})^{-1}(s)$, and looking in $\mathcal{L}_1$ for a match
5:     Define: $\mathcal{L}_{34}$ the list of all $(k_3, k_4)$ such that $E^3_{k_3}(s) = (E^4_{k_4})^{-1}(c_4)$
6:     Search a claw between $\mathcal{L}_{12}$ and $\mathcal{L}_{34}$, if it exists: a pair $(k_1, k_2), (k_3, k_4)$ such that
   all $p_i$ encrypt to all $c_i$
7:     return "good" if a claw exists, "not good" otherwise
8: EndSample
```

We have found significant advantage in combining *merging trees* and *quantum walks*, such as improving the previous best algorithm for 4-SUM. However, this advantage vanishes in the long run, and both methods converge towards the same exponent $\frac{2}{7}$. For now, a problem that can be reduced to k-SUM for any k (such as subset-sum) does not see any improvement from using walks.

It is possible, although we have not attempted, to define a bigger class of merging tree algorithms built entirely over quantum walks, possibly with nested walks. This would be much more technical, and it is difficult to estimate whether one would gain a significant advantage. Whether this might improve the exponent $\frac{2}{7}$ is an interesting open question.

Acknowledgements. The author would like to thank André Chailloux and María Naya-Plasencia for many discussions and comments on intermediate versions of this work, as well as Pierre Briaud, Johanna Loyer and the anonymous reviewers of SAC for their comments. This work has been supported by the European Union's H2020 project No. 714294 (QUASYModo) and by ERC-ADG-ALSTRONGCRYPTO (project 740972).

References

[1] Ambainis, A.: Quantum walk algorithm for element distinctness. SIAM J. Comput. 37(1), 210–239 (2007)

[2] Bai, S., Galbraith, S.D., Li, L., Sheffield, D.: Improved combinatorial algorithms for the inhomogeneous short integer solution problem. J. Cryptology 32(1), 35–83 (2019)

[3] Becker, A., Coron, J., Joux, A.: Improved generic algorithms for hard knapsacks. In: EUROCRYPT. LNCS, vol. 6632, pp. 364–385. Springer (2011)

[4] Belovs, A., Spalek, R.: Adversary lower bound for the k-SUM problem. In: ITCS. pp. 323–328. ACM (2013)

[5] Bernstein, D.J., Jeffery, S., Lange, T., Meurer, A.: Quantum algorithms for the subset-sum problem. In: PQCrypto. Lecture Notes in Computer Science, vol. 7932, pp. 16–33. Springer (2013)

[6] 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)

[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. LNCS, vol. 1380, pp. 163–169. Springer (1998)

[9] Buhrman, H., Dürr, C., Heiligman, M., Høyer, P., Magniez, F., Santha, M., de Wolf, R.: Quantum algorithms for element distinctness. SIAM J. Comput. 34(6), 1324–1330 (2005)

[10] Camion, P., Patarin, J.: The knapsack hash function proposed at crypto'89 can be broken. In: EUROCRYPT. LNCS, vol. 547, pp. 39–53. Springer (1991)

[11] Chailloux, A., Naya-Plasencia, M., Schrottenloher, A.: An efficient quantum collision search algorithm and implications on symmetric cryptography. In: ASIACRYPT (2). LNCS, vol. 10625, pp. 211–240. Springer (2017)

[12] Dinur, I.: An algorithmic framework for the generalized birthday problem. Des. Codes Cryptogr. 87(8), 1897–1926 (2019)

[13] Dinur, I., Dunkelman, O., Keller, N., Shamir, A.: Efficient dissection of composite problems, with applications to cryptanalysis, knapsacks, and combinatorial search problems. In: CRYPTO. LNCS, vol. 7417, pp. 719–740. Springer (2012)

[14] Flajolet, P., Odlyzko, A.M.: Random mapping statistics. In: EUROCRYPT. LNCS, vol. 434, pp. 329–354. Springer (1989)

[15] Gleixner, A., Bastubbe, M., Eifler, L., Gally, T., Gamrath, G., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Lübbecke, M.E., Maher, S.J., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Schubert, C., Serrano, F., Shinano, Y., Viernickel, J.M., Walter, M., Wegscheider, F., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 6.0. Technical report, Optimization Online (2018), `http://www.optimization-online.org/DB_HTML/2018/07/6692.html`

[16] Gleixner, A., Bastubbe, M., Eifler, L., Gally, T., Gamrath, G., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Lübbecke, M.E., Maher, S.J., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Schubert, C., Serrano, F., Shinano, Y., Viernickel, J.M., Walter, M., Wegscheider, F., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin (2018), `http://nbn-resolving.de/urn:nbn:de:0297-zib-69361`

[17] Grassi, L., Naya-Plasencia, M., Schrottenloher, A.: Quantum algorithms for the k-XOR problem. In: ASIACRYPT 2018. LNCS, vol. 11272, pp. 527–559. Springer (2018)

[18] Grover, L.K.: A fast quantum mechanical algorithm for database search. In: STOC. pp. 212–219. ACM (1996)

[19] Howgrave-Graham, N., Joux, A.: New generic algorithms for hard knapsacks. In: EUROCRYPT. LNCS, vol. 6110, pp. 235–256. Springer (2010)

[20] Jaques, S., Schrottenloher, A.: Low-gate quantum golden collision finding. In: SAC. LNCS, Springer (2020), `https://eprint.iacr.org/2020/424`

[21] Kachigar, G., Tillich, J.: Quantum information set decoding algorithms. In: PQCrypto. LNCS, vol. 10346, pp. 69–89. Springer (2017)

[22] Kirshanova, E., Mårtensson, E., Postlethwaite, E.W., Moulik, S.R.: Quantum algorithms for the approximate k-list problem and their application to lattice sieving. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 11921, pp. 521–551. Springer (2019)

[23] Kuperberg, G.: Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In: TQC. LIPIcs, vol. 22, pp. 20–34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013)

[24] Lyubashevsky, V.: The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In: APPROX-RANDOM. LNCS, vol. 3624, pp. 378–389. Springer (2005)

[25] Magniez, F., Nayak, A., Roland, J., Santha, M.: Search via quantum walk. SIAM J. Comput. 40(1), 142–164 (2011)

[26] Minder, L., Sinclair, A.: The extended k-tree algorithm. J. Cryptology 25(2), 349–382 (2012)

[27] Naya-Plasencia, M., Schrottenloher, A.: Optimal merging in quantum k-XOR and k-SUM algorithms. In: EUROCRYPT (2). pp. 311–340. LNCS Vol. 12106, Springer (2020)

[28] Nikolic, I., Sasaki, Y.: Refinements of the k-tree algorithm for the generalized birthday problem. In: ASIACRYPT (2). LNCS, vol. 9453, pp. 683–703. Springer (2015)

[29] Panconesi, A., Srinivasan, A.: Randomized distributed edge coloring via an extension of the chernoff-hoeffding bounds. SIAM J. Comput. 26(2), 350–368 (1997)

[30] Schroeppel, R., Shamir, A.: A $t = \mathcal{O}(2^{n/2}), s = \mathcal{O}(2^{n/4})$ algorithm for certain np-complete problems. SIAM J. Comput. 10(3), 456–464 (1981)

[31] Wagner, D.A.: A generalized birthday problem. In: CRYPTO. LNCS, vol. 2442, pp. 288–303. Springer (2002)

[32] Zhandry, M.: How to record quantum queries, and applications to quantum indifferentiability. In: CRYPTO (2). LNCS, vol. 11693, pp. 239–268. Springer (2019)

A Results for Single-solution k-XOR

Table 1. Quantum time and memory complexity exponents for Single-solution k-XOR obtained with Algorithm 4. The time exponent is the best known for all values of k, and subsumes all previous works.

	Time		QRACM		QRAQM		Parameters	
k	Rounded	As fraction	Rounded	As fraction	Rounded	As fraction	k_1	k_2
2	0.3333	1/3	0.0	0	0.3333	1/3	1	0
3	0.3333	1/3	0.3333	1/3	0.0	0	1	1
4	0.2917	7/24	0.25	1/4	0.1667	1/6	2	1
5	0.3	3/10	0.2	1/5	0.2	1/5	2	1
6	0.3056	11/36	0.1667	1/6	0.2222	2/9	3	1
7	0.2857	2/7	0.2857	2/7	0.1429	1/7	3	2
8	0.2917	7/24	0.25	1/4	0.1667	1/6	4	2
9	0.2963	8/27	0.2222	2/9	0.1852	5/27	4	2
10	0.3	3/10	0.3	3/10	0.15	3/20	5	3
11	0.2879	19/66	0.2727	3/11	0.1515	5/33	5	3
12	0.2917	7/24	0.25	1/4	0.1667	1/6	5	3
13	0.2949	23/78	0.2308	3/13	0.1795	7/39	6	3
14	0.2857	2/7	0.2857	2/7	0.1429	1/7	6	4
15	0.2889	13/45	0.2667	4/15	0.1556	7/45	7	4
16	0.2917	7/24	0.25	1/4	0.1667	1/6	7	4
17	0.2941	5/17	0.2941	5/17	0.1471	5/34	8	5
18	0.287	31/108	0.2778	5/18	0.1481	4/27	8	5
19	0.2895	11/38	0.2632	5/19	0.1579	3/19	8	5
20	0.2917	7/24	0.25	1/4	0.1667	1/6	9	5
21	0.2857	2/7	0.2857	2/7	0.1429	1/7	9	6
22	0.2879	19/66	0.2727	3/11	0.1515	5/33	10	6
23	0.2899	20/69	0.2609	6/23	0.1594	11/69	10	6
24	0.2917	7/24	0.2917	7/24	0.1458	7/48	11	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

B List Sizes and Heuristics

In this section, we prove that the list sizes do not deviate "too much" from their expectation in the merging algorithms studied in this paper, and we show that Heuristic 1 is not required in our quantum algorithms.

Let us consider two lists $\mathcal{L}_1, \mathcal{L}_2$ merged into $\mathcal{L}_u$, with some prefix t of length un. We start with the case where $\mathcal{L}_u$ is smaller.

Lemma 4. *If $\ell_u \leq \max(\ell_1, \ell_2)$, there exists two constants $a, b > 0$ such that with probability $1 - e^{-an}$, a proportion b of the elements of $\mathcal{L}_u$ are drawn uniformly at random from all n-bit strings with prefix t.*

Proof. As an example of non-independence between the output pairs, let us consider $x_1, y_1 \in \mathcal{L}_1$ and $x_2, y_2 \in \mathcal{L}_2$, then the events $x_1 \oplus x_2 = t|*$, $y_1 \oplus x_2 = t|*$, $x_1 \oplus y_2 = t|*$ and $y_1 \oplus y_2 = t|*$ are not independent. In order to recover independence when $\ell_u \leq \max(\ell_1, \ell_2)$, we will use an argument similar to [24]. We first need a technical result, which is a property of random mappings.

Lemma 5. *Let $h : \{0,1\}^n \to \{0,1\}^n$ be a random function. Let $Y(h)$ be the number of elements in $\{0,1\}^n$ without a preimage. Then:*

$$\Pr(Y(h) > 0.4 \cdot 2^n) \leq 0.9987^{2^n} \quad . \tag{9}$$

Proof. We write $Y(h) = \sum_{i \in \{0,1\}^n} Y_i(h)$, where $Y_i(h)$ is 1 if i has no preimage by h. The $Y_i(h)$ are not independent, but they are negatively correlated: knowing that x has no preimage only decreases the probability that this is the case for $x' \neq x$. In that case, a Chernoff bound applies [29], and for any $\delta > 0$:

$$\Pr\left(Y(h) \geq \frac{(1+\delta)2^n}{e}\right) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\frac{2^n}{e}} \quad ,$$

where $\frac{2^n}{e}$ is the average of $Y(h)$, which is a standard result of random mappings [14]. We then choose $\delta = 0.4e - 1 \simeq 0.087$ and obtain the claimed bound by rounding the term on the left side. □

Assume without loss of generality that $\ell_2 \leq \ell_1$. The idea is that since $\ell_u \leq \ell_1$, a given element of $\mathcal{L}_1$ intervenes in one pair on average, which ensures the independence of the pairs. We assume that $\ell_2 - u > 0$.

First of all, we cut $\mathcal{L}_2$ into sublists $\mathcal{L}_2^x$ depending on un-bit prefixes x. Then any element in $\mathcal{L}_u$ is the sum of an element x_1 in $\mathcal{L}_1$ of prefix x, and an element x_2 of $\mathcal{L}_2^x$. We use Chernoff bounds to show that the individual sizes of the $\mathcal{L}_2^x$ do not deviate much from their expectation $\mathbb{E}\left(L_2^x\right) = 2^{n(\ell_2 - u)}$. We have:

$$\forall x, \forall \delta \leq 1, \Pr(|L_2^x - \mathbb{E}\left(L_2^x\right)| \geq \delta\, \mathbb{E}\left(L_2^x\right)) \leq 2e^{-\frac{\delta^2\, \mathbb{E}\left(L_2^x\right)}{3}}, \tag{10}$$

and by taking $\delta = (\mathbb{E}\left(L_2^x\right))^{1/3}$, since $\mathbb{E}\left(L_2^x\right)$ is exponential in n, taking a union bound over all prefixes x does not change that the probability to deviate is vanishingly small:

$$\Pr\left(\exists x, |L_2^x - \mathbb{E}\left(L_2^x\right)| \geq \mathbb{E}\left(L_2^x\right)^{2/3}\right) \leq 2^{un+1} e^{-\frac{\mathbb{E}\left(L_2^x\right)^{1/3}}{3}} \quad . \tag{11}$$

Focusing on $\mathcal{L}_1$, we extract a sublist $\mathcal{L}_1'$ of its elements having *distinct* prefixes of un bits, and we show that the size of $\mathcal{L}_1'$ is only smaller by a constant. This comes from Lemma 5. If we index elements of $\mathcal{L}_1$ by their $\ell_1 n$-bit prefix, then at least a constant proportion of these prefixes are occupied, with probability exponentially close to 1. Combining this with Equation (11), we bound the deviation of the merged list size from its expectation.

It remains to observe that $\mathcal{L}_1' \bowtie \mathcal{L}_2 \subseteq \mathcal{L}_u$ contains independent sums, since each element of $\mathcal{L}_2$ appears at most once. In the case where $\ell_1 = \ell_2 = u$, taking unique prefixes for both lists $\mathcal{L}_1$ and $\mathcal{L}_2$ gives the same result. □

Lemma 4 allows to show that if all list sizes in a merging tree are decreasing, then with probability exponentially close to 1, its time complexity is, up to a constant, equal to the average. Indeed, we will simply use the Lemma for each merging step individually. This includes Wagner's algorithm [31] as a special case.

In a generic merging tree, however, the list sizes do not always decrease. In fact, when the initial lists are too small, the first levels of the tree will make them *increase*, as remarked by Minder and Sinclair [26]. They will decrease afterwards, because the complete merging tree ends with a single solution.

This case seems problematic at first sight, because the bigger lists of the middle levels are not statistically close to lists of uniform bit-strings. For example, when we take a complete cross-product of two lists, the $2^{(\ell_1+\ell_2)n}$ resulting elements were obtained from $2^{\ell_1 n} + 2^{\ell_2 n}$, and there are many relations between them. But despite this "loss of randomness", the list behaves nicely for the subsequent merging steps. The following lemma aims at capturing this intuition.

Lemma 6. *Consider t lists $\mathcal{L}_1, \ldots, \mathcal{L}_t$ and their product $\mathcal{L} = \mathcal{L}_1 \times \ldots \times \mathcal{L}_t$. Assume that the $\mathcal{L}_i$ are of exact size $2^{n\ell}$ and contain uniformly drawn n-bit strings. Then there exists two constants $a, b > 0$ such that with probability $1 - e^{-an}$, $\mathcal{L}$ meets a proportion b of $n\ell t$-bit prefixes.*

Proof. We detail the proof for a pair of lists $(\mathcal{L}_1, \mathcal{L}_2)$, but the extension to t lists is easy. We consider two independent, arbitrary ranges of ℓn bits, "range 1" and "range 2".

By Lemma 5, $\mathcal{L}_1$ meets a proportion b_1 of bit-strings in range 1, with probability $1 - e^{a_1 n}$ for some $a_1, b_1 > 0$. We will assume that all prefixes are met (for simplicity), but in general we must always reason with some of them missing.

We can define a random variable $X_1(i)$ that given a range-1 value $i \in \{0,1\}^{n\ell}$, gives the value of the $(1-\ell)n$ remaining bits of the corresponding element in $\mathcal{L}_1$. We define $X_2(i)$ similarly.

The cross-product $\mathcal{L}_1 \times \mathcal{L}_2$ can be partitioned into $2^{\ell n}$ bins depending on the value in range 1. Bin i contains all the $X_1(j) \oplus X_2(i \oplus j)$ for $j \in \{0,1\}^n$. But then, a given $X_1(j)$ or $X_2(j)$ intervenes only in one element: because they are independent, we can use Lemma 5 again. This time, we show that for a given value of i, all (up to a constant) values in *range 2* are met. Because we used a Chernoff bound, this can hold simultaneously for all range-1 values.

In the end, a constant number of range-1 values are met, and for each of them, a constant number of range-2 values are met as well. Thus a constant proportion of 2ℓ-bit prefixes are met in total. $\square$

Lemma 6 contains all that we need for more general merging trees. At a given merging step, we will not have necessarily a complete cross-product, as we might have merged for some prefix value. But the merged list is then a sublist of the cross-product, depending on the prefixes, and we can also bound its size.

In practice, we need to use this lemma only for our Single-solution k-XOR algorithms, which have a very specific shape. They have three levels (see Algorithm 3):

- Level 2: 4 complete cross-products (of different sizes);
- Level 1: merging with a decreasing size;
- Level 0: merging with a single solution at most (often none).

The tree is parameterized by some guess, that will only be right once. So we only need our algorithm to succeed once, on this guess. By Lemma 6, we guarantee the size of the level-2 lists, but also of the level-1 lists (since there are enough elements with the given prefixes). Then it amounts to find a collision between them.

Quantum Complexities. If we remove Heuristic 1, the "quantum merging" lemma (Lemma 3) is not true anymore. We cannot guarantee that the list sizes are exponentially close to their average and, in particular, all our QSample procedures may now have constant probabilities of error.

However, fixing the Many-solutions and Single-solution k-XOR algorithms presented in this paper is easy. In the Many-Solutions case, the optimal algorithms use only a single level of Amplitude Amplification: they do not amplify non-exact procedures. In fact, they perform only quantum searches in some lists at the lowest level. The guarantee on all list sizes entails that these searches will succeed with constant probability. Since the results (partial k-XORs) can always be checked, there is only a constant increase in time complexity.

In the Single-solution case, the algorithms use a single level of quantum search (for an intermediate prefix, and possibly, a sublist), followed by either a single level of quantum search, or a quantum walk, that solves a Single claw-finding problem. This "inner" procedure has a constant probability of success due to our loose guarantees on the list sizes: we can repeat it $\mathcal{O}(n)$ times to make its failure probability exponentially low. This ensures that when the solution occurs, the "inner" procedure always finds it, and that the "good choice" for the outer search is flagged without error. This ensures a constant probability of success.

C Proof of Optimality in the QRACM Setting

In the QRACM model, we give a new proof of the following result from [27], based on our new definition of merging trees:

> For any integer k and $c > 0$, the best quantum merging procedure that *samples* 2^{cn} times a k-XOR on n bits has a time complexity exponent $\max(\alpha_k(1+2c), c)$, where $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$ where $\kappa = \lfloor \log_2(k) \rfloor$.

Proof. We use a recurrence on k (this general statement with the parameter c is required for a nice recurrence hypothesis). For $k = 2$, we have $\kappa = 1$ and $\alpha_2 = \frac{1}{3}$. Finding 2^{cn} collisions $(x_1, x_2, h(x_1) \oplus h(x_2) = 0)$, or *sampling a collision* 2^{cn} *times* can be done in time $2^{\left(\frac{2c}{3} + \frac{1}{3}\right)n}$, using a re-optimization of the steps in BHT collision search [8]. This works as long as $c \leq 1$, i.e., $2c + 1 \leq 3$. Thus, the theorem is true for $k = 2$.

Let us consider a merging tree $\mathcal{T}_k$ for some $k > 2$, with a list size $c > 0$ at the root. The root node has two subtrees: the "list" one $\mathcal{T}_r$, on the right, and the "sampled" one $\mathcal{T}_l$, on the left. Let u be the length of the zero-prefix in both nodes. Let ℓ_r and ℓ_l be their respective sizes, let $k_r + k_l = k$ be their width.

First, notice that we have $1 - u - \ell_r \geq 0$, otherwise we could reduce the value of the parameter ℓ_r without increasing the time complexity.

We use the recurrence hypothesis on $\mathcal{T}_l$ and $\mathcal{T}_r$, relatively to the number of zeros u that they have (since they contain XORs on un bits instead of n). The right list, of size $u\frac{\ell_r}{u}$, is produced in time $\max\left(\alpha_{k_r}(1 + 2\frac{\ell_r}{u}), \frac{\ell_r}{u}\right) u = \max\left(\alpha_{k_r}(u + 2\ell_r), \ell_r\right)$.

Since we want to sample c times from the root node, we need to sample $c + \frac{1}{2}(1 - u - \ell_r)$ times from the left list, which costs:

$$\max\left(\alpha_{k_l}\left(u + 2\left(c + \frac{1}{2}(1 - u - \ell_r)\right)\right), \left(c + \frac{1}{2}(1 - u - \ell_r)\right)\right)$$

We obtain that the time complexity exponent t must be minimized under the constraints:

$$\begin{array}{ll} \text{(C1) } t \geq \alpha_{k_r}(u + 2\ell_r) & \text{(C2) } t \geq \ell_r \\ \text{(C3) } t \geq \alpha_{k_l}(2c + 1 - \ell_r) & \text{(C4) } t \geq c + \frac{1}{2} - \frac{u}{2} - \frac{\ell_r}{2} \end{array}$$

By combining these inequalities, we will obtain information about the shape of the optimal trees. We combine (C1), (C4) and (C3) to eliminate u and ℓ_r:

$$\text{(C1)} + 2\alpha_{k_r}\text{(C4)} + \frac{\alpha_{k_r}}{\alpha_{k_l}}\text{(C3)} \iff t\left(1 + 2\alpha_{k_r} + \frac{\alpha_{k_r}}{\alpha_{k_l}}\right) \geq 2\alpha_{k_r}(2c + 1) \enspace .$$

Then this inequality becomes:

$$t \geq \frac{2\alpha_{k_r}}{1 + 2\alpha_{k_r} + \frac{\alpha_{k_r}}{\alpha_{k_l}}}(2c + 1) = \frac{2\alpha_{k_r}\alpha_{k_l}}{\alpha_{k_l} + \alpha_{k_r} + 2\alpha_{k_r}\alpha_{k_l}}(2c + 1) \enspace .$$

We are interested in the quantity $\frac{2\alpha_{k_r}\alpha_{k_l}}{\alpha_{k_l} + \alpha_{k_r} + 2\alpha_{k_r}\alpha_{k_l}}$ when k_l and k_r vary. We would like to make it minimal, since this loosens the constraint on t. Thus, we want to maximize its inverse:

$$1 + \frac{1}{2\alpha_{k_l}} + \frac{1}{2\alpha_{k_r}} \enspace .$$

Since α_{k_l} is a decreasing function of k_l, this sum becomes maximal when k_l is close to $k_r = k - k_l$. More precisely: if k is even, then $k_r = k_l = \frac{k}{2}$ gives the smallest sum possible. If k is odd, then $k_l = \left\lfloor \frac{k}{2} \right\rfloor$ and $k_r = k - \left\lfloor \frac{k}{2} \right\rfloor$ *or the converse.*

In both cases, if we write $\kappa = \lfloor \log_2 k \rfloor$, then

$$\lfloor \log_2 \lfloor k/2 \rfloor \rfloor = \lfloor \log_2 (k - \lfloor k/2 \rfloor) \rfloor = \kappa - 1 \enspace .$$

Using the recurrence hypothesis, we obtain that:

$$1 + \frac{1}{2\alpha_{k_l}} + \frac{1}{2\alpha_{k_r}} = 1 + \frac{(1+\kappa-1)2^{\kappa-1} + k - \left\lfloor \frac{k}{2} \right\rfloor}{2^\kappa} + \frac{(1+\kappa-1)2^{\kappa-1} + \left\lfloor \frac{k}{2} \right\rfloor}{2^\kappa}$$
$$= \frac{2^\kappa(1+\kappa)+k}{2^\kappa} \enspace .$$

Thus, we can write: $t \geq (2c+1)\frac{2^\kappa}{2^\kappa(1+\kappa)+k}$, which gives the expected formula for α_k. The second inequality $t \geq c$ stems trivially from (C4).

We finish the proof of optimality by showing, also by induction on k, that the optimization of the balanced trees T_k indeed reaches this exponent.

Lemma 7. *Optimizing the balanced trees T_k yields the optimal exponents.*

First, we focus on the case $c \leq \frac{\alpha_k}{1-2\alpha_k}$, where the complexity exponent is expected to be $(2c+1)\alpha_k$, and we consider an even k. We choose $u = (1-3\alpha_k)(2c+1)$ and $\ell_r = \alpha_k(2c+1)$. This gives that $c + \frac{1}{2} - \frac{u}{2} - \frac{\ell_r}{2} = (2c+1)\alpha_k$, so (C4) is satisfied. Second, we have:

$$\alpha_{k/2}(u + 2\ell_r) = (2c+1)\alpha_{k/2}(1-\alpha_k) = (2c+1)\alpha_k$$

by definition of the α_k (their formula implies $\frac{\alpha_{k/2}}{1+\alpha_{k/2}} = \alpha_k$). Thus (C1) is satisfied. By a similar computation, (C3) is satisfied since $\alpha_{k/2}(2c+1-\ell_r) = \alpha_k(2c+1)$. Finally, (C2) is trivially satisfied by our choice of ℓ_r.

If k is odd, we choose

$$u = \left(1 - 3\alpha_k + \frac{1}{(1+\kappa)2^\kappa + k}\right)(2c+1) \text{ and } \ell_r = \left(\alpha_k - \frac{1}{(1+\kappa)2^\kappa + k}\right)(2c+1) \enspace .$$

Again, (C4) becomes an equality. (C1) is an equality as well, using the fact that $\alpha_{k_r} = \alpha_{\lfloor k/2 \rfloor} = \alpha_{(k-1)/2}$. Indeed, we have:

$$\alpha_{k_r}(u + 2\ell_r) = \alpha_{(k-1)/2}\left(1 - \alpha_k - \frac{1}{(1+\kappa)2^\kappa + k}\right)$$
$$= (2c+1)\frac{2^{\kappa-1}}{\kappa 2^{\kappa-1} + (k-1)/2}\left(\frac{2^\kappa(1+\kappa) + k - 2^\kappa - 1}{(1+\kappa)2^\kappa + k}\right) = (2c+1)\alpha_k \enspace .$$

The constraints (C2) and (C3) become strict inequalities, but they are also satisfied.

When $c \geq \frac{\alpha_k}{1-2\alpha_k}$, all the merges become classical. The only quantum operations remaining are the Grover searches in some newly inserted leaves.

□

D Proof of Optimality without QRACM

In the circuit model, we found that our new definition of merging trees allowed to reduce the exponents for $k = 5$ and 7 that were obtained in [27]. We obtain

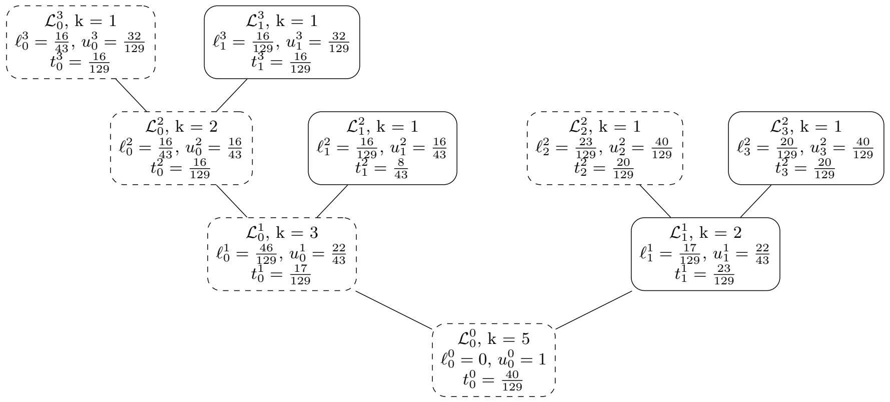

$\mathcal{L}_0^3$, k = 1
$\ell_0^3 = \frac{16}{43}$, $u_0^3 = \frac{32}{129}$
$t_0^3 = \frac{16}{129}$
$\mathcal{L}_1^3$, k = 1
$\ell_1^3 = \frac{16}{129}$, $u_1^3 = \frac{32}{129}$
$t_1^3 = \frac{16}{129}$
$\mathcal{L}_0^2$, k = 2
$\ell_0^2 = \frac{16}{43}$, $u_0^2 = \frac{16}{43}$
$t_0^2 = \frac{16}{129}$
$\mathcal{L}_1^2$, k = 1
$\ell_1^2 = \frac{16}{129}$, $u_1^2 = \frac{16}{43}$
$t_1^2 = \frac{8}{43}$
$\mathcal{L}_2^2$, k = 1
$\ell_2^2 = \frac{23}{129}$, $u_2^2 = \frac{40}{129}$
$t_2^2 = \frac{20}{129}$
$\mathcal{L}_3^2$, k = 1
$\ell_3^2 = \frac{20}{129}$, $u_3^2 = \frac{40}{129}$
$t_3^2 = \frac{20}{129}$
$\mathcal{L}_0^1$, k = 3
$\ell_0^1 = \frac{46}{129}$, $u_0^1 = \frac{22}{43}$
$t_0^1 = \frac{17}{129}$
$\mathcal{L}_1^1$, k = 2
$\ell_1^1 = \frac{17}{129}$, $u_1^1 = \frac{22}{43}$
$t_1^1 = \frac{23}{129}$
$\mathcal{L}_0^0$, k = 5
$\ell_0^0 = 0$, $u_0^0 = 1$
$t_0^0 = \frac{40}{129}$

Fig. 7. Optimal 5-XOR merging tree without QRACM.

$\beta_5 = \frac{40}{129}$ and $\beta_7 = \frac{15}{53}$ instead of $\frac{14}{45}$ and $\frac{2}{7}$ respectively. The details are given in Figure 7 and Figure 8. For other values of k, our results coincide with [27] and we prove:

For any integer $k \geq 8$ and $c > 0$, the best quantum merging procedure *without qRAM* that *samples* 2^{cn} times a k-XOR on n bits has a time complexity exponent $\max(\beta_k(1+c), c)$, where:

$$\beta_k = \begin{cases} \frac{1}{\kappa+1} \text{ if } k < 2^\kappa + 2^{\kappa-1} \\ \frac{2}{2\kappa+3} \text{ if } k \geq 2^\kappa + 2^{\kappa-1} \end{cases},$$

and $\kappa = \lfloor \log_2 k \rfloor$. This procedure samples *classically* all nodes, except some leaves.

Proof. We prove this by induction on k. For small values of k, the experimental results give us the optimal trees. We consider a merging tree $\mathcal{T}_k$ for $k \geq 8$, with a list size $c > 0$ at the root. We use the same notations as in Appendix C, and introduce $k_l, k_r, \beta_{k_l}, \beta_{k_r}$ and the variables u, ℓ_r.

Let us take some integer $k \geq 8$. Having $+c$ instead of $+2c$ in the formula comes from the use of a classical merging at the root. Let us remark the following: once we know that the root merge is classical, we can deduce easily that both subtrees must be of similar shapes, hence $k_\ell = \lfloor \frac{k}{2} \rfloor$ and $k_r = \lceil \frac{k}{2} \rceil$ or the converse. Then we can use the recurrence hypothesis easily: the two subtrees have the same complexity, which depends on the case for k. If $k < 2^\kappa + 2^{\kappa-1}$, then $\lfloor \frac{k}{2} \rfloor < 2^{\kappa-1} + 2^{\kappa-2}$; and conversely, if $k \geq 2^\kappa + 2^{\kappa-1}$, then $\lfloor \frac{k}{2} \rfloor \geq 2^{\kappa-1} + 2^{\kappa-2}$.

In order to prove that the root merge is classical, let us assume that it is quantum instead. We use the recurrence hypothesis for both subtrees. Although the actual optimal merging trees do not allow to sample quantumly, we suppose that they do. Thus, sampling $c + \frac{1}{2}(1 - u - \ell_r)$ times from the left child is done in time $\max(c + \frac{1}{2}(1 - u - \ell_r), \beta_{k_l}\left(u + c + \frac{1}{2}(1 - u - \ell_r)\right))$. Besides, for each time we sample the left child, we also need to do a memory query on the right, which

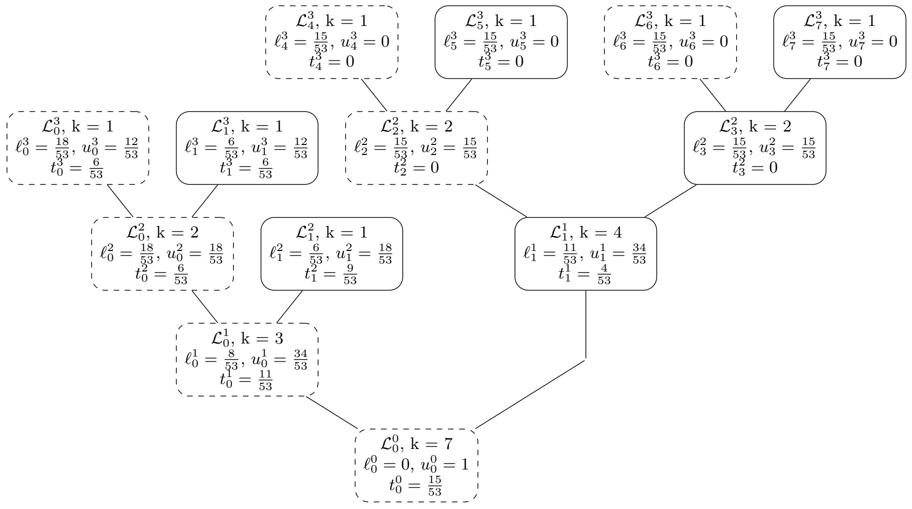

$\mathcal{L}_4^3$, k = 1 $\ell_4^3 = \frac{15}{53}$, $u_4^3 = 0$ $t_4^3 = 0$
$\mathcal{L}_5^3$, k = 1 $\ell_5^3 = \frac{15}{53}$, $u_5^3 = 0$ $t_5^3 = 0$
$\mathcal{L}_6^3$, k = 1 $\ell_6^3 = \frac{15}{53}$, $u_6^3 = 0$ $t_6^3 = 0$
$\mathcal{L}_7^3$, k = 1 $\ell_7^3 = \frac{15}{53}$, $u_7^3 = 0$ $t_7^3 = 0$
$\mathcal{L}_0^3$, k = 1 $\ell_0^3 = \frac{18}{53}$, $u_0^3 = \frac{12}{53}$ $t_0^3 = \frac{6}{53}$
$\mathcal{L}_1^3$, k = 1 $\ell_1^3 = \frac{6}{53}$, $u_1^3 = \frac{12}{53}$ $t_1^3 = \frac{6}{53}$
$\mathcal{L}_2^2$, k = 2 $\ell_2^2 = \frac{15}{53}$, $u_2^2 = \frac{15}{53}$ $t_2^2 = 0$
$\mathcal{L}_3^2$, k = 2 $\ell_3^2 = \frac{15}{53}$, $u_3^2 = \frac{15}{53}$ $t_3^2 = 0$
$\mathcal{L}_0^2$, k = 2 $\ell_0^2 = \frac{18}{53}$, $u_0^2 = \frac{18}{53}$ $t_0^2 = \frac{6}{53}$
$\mathcal{L}_1^2$, k = 1 $\ell_1^2 = \frac{6}{53}$, $u_1^2 = \frac{18}{53}$ $t_1^2 = \frac{9}{53}$
$\mathcal{L}_1^1$, k = 4 $\ell_1^1 = \frac{11}{53}$, $u_1^1 = \frac{34}{53}$ $t_1^1 = \frac{4}{53}$
$\mathcal{L}_0^1$, k = 3 $\ell_0^1 = \frac{8}{53}$, $u_0^1 = \frac{34}{53}$ $t_0^1 = \frac{11}{53}$
$\mathcal{L}_0^0$, k = 7 $\ell_0^0 = 0$, $u_0^0 = 1$ $t_0^0 = \frac{15}{53}$

Fig. 8. Optimal 7-XOR merging tree without QRACM.

costs ℓ_r because we don't have QRACM. Building the right child costs a time at least $\max(\beta_{k_r}(u + \ell_r), \ell_r)$ (notice that this is not tight for small k_r). Let t be the time exponent, then we have the constraints:

(C1) $t \geq \beta_{k_r}(u + \ell_r)$ (C2) $t \geq \frac{\beta_{k_l}}{2}(2c + 1 + u - \ell_r)$
(C3) $t \geq c + \frac{1}{2}(1 - u - \ell_r) + \ell_r$
$\implies 2t \geq (2c + 1 - u + \ell_r)$

By combining (C2) and (C3), we obtain:

$$\left(\frac{2}{\beta_{k_l}} + 2\right) t \geq 2(2c + 1) \implies t \geq \frac{\beta_{k_l}}{1 + \beta_{k_l}}(2c + 1) = \beta_{2k_l}(2c + 1)$$

where the last equality follows by definition of the β_i. But since $t \leq \beta_{k-1}(c+1)$, we obtain that $\beta_{k-1} \geq \beta_{2k_l} \implies 2k_l \geq k - 1 \implies k_l \geq \lfloor k/2 \rfloor$.

Furthermore, at the optimal point we expect:

$$\frac{\beta_{k_l}}{2}(2c + 1 + u - \ell_r) = \frac{1}{2}(2c + 1 - u + \ell_r) \implies u = \ell_r + \frac{\beta_{k_l} - 1}{\beta_{k_l} + 1}(2c + 1) \ .$$

Next, we remark that an algorithm without QRACM should cost at least as much as in the QRACM model, so we introduce:

$$\text{(C4)} \quad t \geq \alpha_{k_r}(u + 2\ell_r) \implies t \geq \alpha_{k_r}\left(3\ell_r + \frac{\beta_{k_l} - 1}{\beta_{k_l} + 1}(2c + 1)\right) \ .$$

Since $u \geq 0$, we should have $\ell_r \geq \frac{1 - \beta_{k_l}}{\beta_{k_l} + 1}(2c + 1)$. But then we find:

$$t \geq 2\alpha_{k_r}\frac{\beta_{k_l} - 1}{\beta_{k_l} + 1}(2c + 1) \ .$$

Since we have $k_l \geq \lfloor k/2 \rfloor$, at the same time, we should have $k_r \leq \lceil k/2 \rceil$ so $k_r \leq k_l + 1$ and $\alpha_{k_r} \geq \alpha_{k_l+1}$. This inequality becomes $t \geq 2\alpha_{k_l+1}\frac{1-\beta_{k_l}}{\beta_{k_l}+1}(2c+1)$. A quick computation of the first values of α_i and β_i shows that for $k_l \geq 15, 2\alpha_{k_l+1}\frac{1-\beta_{k_l}}{\beta_{k_l}+1} \geq \beta_{k_l+1}$. We would thus get $t \geq \beta_{k_l+1}(2c+1) \geq \beta_{k_l+1}(c+1)$, which contradicts the fact that the time should get smaller as k increases.

In conclusion, while small values of k may benefit from using a quantum search at the root of the tree (and this is indeed the case), for a general k, the root node is a classical merge between two classically stored lists.

□

E Proof of Optimality for Single-solution k-XOR

We now prove Theorem 3. More precisely, we will prove the following result. It implies the formula for the optimal complexity and the shape of the optimal trees that we gave.

Theorem 7. *For any k, the optimal time t for the Single-solution k-XOR problem, with our merging tree framework, is given by:*

$$t = \min_{k_1,k_2\in\mathbb{N}^2, k_1+k_2\leq k}\left(\max\left(\frac{k_2}{k}, \frac{1}{2}\left(1-\frac{k_1}{k}\right), \frac{1}{4}\left(1+\frac{k_1-k_2}{k}\right)\right)\right) \tag{12}$$

and can thus be obtained by solving a simple mixed integer linear program.

In Appendix E.1, we outline the definition of extended merging trees. In Appendix E.2, we show how this definition leads to optimal algorithms of a very simple shape, with a constant number of parameters to optimize. We reduce the constraints further in Appendix E.3 and finish the proof of the theorem.

E.1 Definition of Extended Merging Trees

Structurally, we still consider binary trees as in Definition 2. We adopt the same numbering of nodes and keep the variables k_i^j, u_i^j, ℓ_i^j that determine the shape of the list $\mathcal{L}_i^j$. Thus, the tree still represents an appropriate merging operation.

We introduce a new variable r for *repetitions*. We cannot expect the tree to *always* contain a k-XOR; instead, we will *repeat* the computation until we find one (with a quantum search). Constraint 3 and 2 remain unchanged, but we adapt the constraint for the root node:

Constraint 5 (Root node)**.** *At the root node: $u_0^0 + r = 1$ and $\ell_0^0 = 0$.*

Next, we add new *repetition* variables r^j. On most nodes we set $r = 0$, but we single out the *right subtrees on the main branch*, as depicted on Figure 9. Thus, there is only one non-zero repetition variable at each level, which is why we simply number them level by level.

For each subtree $\mathcal{T}^j$, r^j represents the number of times it must be recomputed. Each computation should produce a new, independent list of elements, possibly with a new prefix.

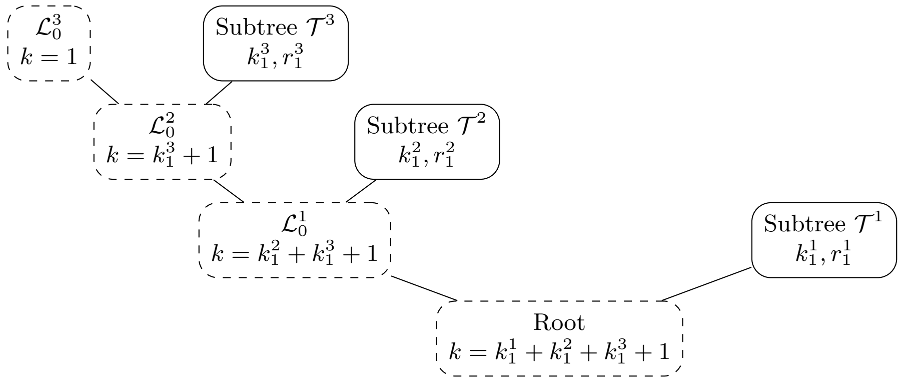


Fig. 9. *Main branch* of a merging tree and all the subtrees that are attached to it.

Constraint 6 (Repetitions)**.** *We have: $r = \sum_j r^j$, and for each subtree $\mathcal{T}^j$ of width k^j: $r \leq \frac{k_j}{k} - \ell^j$, where ℓ^j is the size of the list at the root of $\mathcal{T}^j$.*

We still denote by t_i^j the sampling time of a node. Constraint 4 still applies, in its simplest form, since we use the QRAQM model only.

Constraint 7 (Sampling)**.** *Let T_i^j be a node in the tree, either an S-node or an L-node. If T_i^j is a leaf, $t_i^j = \frac{u_i^j}{2}$. Otherwise, T_i^j has an S-child S_{2i}^{j+1} and an L-child S_{2i+1}^{j+1}, and:*

$$t_i^j = t_{2i}^{j+1} + \frac{1}{2}\max(u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0) \ . \tag{13}$$

However, the total time complexity will be computed differently. Focusing on the subtrees $\mathcal{T}^j$ of the main branch, we let t^j denote their respective *complete* time complexities, that is, the time to build the whole subtree with quantum merging.

Constraint 8 (Subtrees)**.** *Let $\mathcal{T}^j$ be the right subtree at level i. Then:*

$$t^j = \max\left(\max_{\textit{List nodes of } \mathcal{T}^j} (t_i^j + \ell_i^j) \right) \ , \tag{14}$$

where the sum is over all list nodes of $\mathcal{T}^j$, including its own root (since this is a list node itself).

Then, we can now define the formulas for the time and memory complexities.

Definition 5. *Let $\mathcal{T}$ be an extended k-merging tree. Let $\mathcal{T}^1, \dots, \mathcal{T}^p$ be the right subtrees of the main branch. We define $\mathsf{T_q}(\mathcal{T})$, $\mathsf{T_c}(\mathcal{T})$ and $\mathsf{M}(\mathcal{T})$ as:*

$$\mathsf{M}(\mathcal{T}) = \max_{\textit{List nodes}} (\ell_i^j)$$

$$\mathsf{T_q}(\mathcal{T}) = \max\left(\frac{r}{2} + t_0^0, \frac{r^1}{2} + t^1, \frac{r^1 + r^2}{2} + t^2, \dots, \frac{1}{2}\left(\sum_{j'=1}^{j} r^{j'}\right) + t^j, \dots, \frac{r}{2} + t^p\right)$$

The idea of this definition is that the algorithm performs p nested loops, one for each subtree of the main branch. We choose to nest from level 1 to p, with the idea that bigger and more costly subtrees may be attached to nodes at lower levels. This generic view is displayed in Algorithm 6. However in the optimal algorithm for Single-solution k-XOR, all these levels collapse into a single one.

Algorithm 6 Generic algorithm defined by an extended merging tree.

```
    Input: oracle access to $h : \{0,1\}^{n/k} \to \{0,1\}^n$
    Output: $k$-XOR solution tuple
 1: for all Choices of $\mathcal{T}^1$ do
                ▷ Either defined with a change of prefix, or a new choice of elements.
 2:     Build $\mathcal{T}^1$
 3:     for all Choices of $\mathcal{T}^2$ do
 4:         Build $\mathcal{T}^2$
    ...
 5:         for all Choices of $\mathcal{T}^p$ do
 6:             Build $\mathcal{T}^p$
 7:             Sample $x \in \mathcal{L}_0^p$
 8:                 Find a match in $\mathcal{T}^p$
 9:                 Find a match in $\mathcal{T}^{p-1}$
    ...
10:                 Find a match in $\mathcal{T}^1$
11:                 if this gives a complete $k$-XOR to zero then
12:                     return the solution
13:             EndSample
```

We can see that, with our definitions of t^j, t^0, r^j and r, the time complexity of Algorithm 6, up to a polynomial factor, is going to be:

$$2^{nr_1}\left(\underbrace{2^{nt_1}}_{\mathcal{T}_1} + 2^{nr_2}\left(\underbrace{2^{nt_2}}_{\mathcal{T}^2} + \dots + 2^{nr_p}\left(\underbrace{2^{nt_p}}_{\mathcal{T}_p} + \underbrace{2^{nt^0}}_{\text{Sample}}\right)\dots\right)\right) \tag{15}$$

where we recover the equation of Definition 5 (in the quantum setting, all these loops become nested quantum searches).

Correspondence between Trees and Algorithms. Similarly as in Section 4, to any extended merging tree corresponds a classical (respectively quantum) extended merging algorithm that has the wanted complexity.

Theorem 8 (Quantum extended merging strategies). *Let $\mathcal{T}_k$ be an extended k-merging tree and $\mathsf{T_q}(\mathcal{T}_k)$ computed as in Definition 5. Then there exists a* quantum extended merging algorithm *that, given access to a quantum oracle for h, finds a k-XOR.*

This algorithm succeeds with constant probability. It runs in time $\widetilde{\mathcal{O}}\left(2^{\mathsf{T_q}(\mathcal{T}_k)n}\right)$, counted in n-qubit register operations, makes the same number of queries to h. It uses a memory $\mathcal{O}\left(2^{\mathsf{M}(\mathcal{T}_k)n}\right)$, counted in n-qubit registers (QRAQM). The constants in the $\mathcal{O}$ depend on k.

Proof (sketched). We rely on Theorem 2 for the correctness of merging strategies. Each level of quantum search builds a new subtree. The search space itself is defined by an arbitrary prefix, either of the codomain (a merging constraint) or of the domain (an input sublist). A given bit-string of the search space at level j is good if, after building the corresponding subtree $\mathcal{T}^j$, and after running the search at level $j+1$, we find a solution k-XOR. Among all repetitions of the subtrees $\mathcal{T}^1, \ldots, \mathcal{T}^p$, only one choice shall lead to a solution. We miss it if the corresponding merging tree fails to find it, but Theorem 2 ensures a constant probability of success. □

E.2 Step 1: Reducing the Search Space

When merging, we drop many tuples. But since all possibilities must be studied in the end, this will only create more repetitions. For example, Schroeppel and Shamir's algorithm requires $2^{n/4}$ repetitions due to the intermediate prefix of $\frac{n}{4}$ bits. This simple fact simplifies considerably the shape of the trees.

Lemma 8. *For any k, the optimal time complexity is reached by a tree where all main subtrees are trivial merges: $\mathcal{T}^i$ has no guessed prefix, except at its root.*

Proof. Let $\mathcal{T}^i$ be one of the subtrees of the main branch. This is a list node of size ℓ and prefix u. Next, we assume that its children $\mathcal{L}_s$ (sampled) and $\mathcal{L}_l$ (built) have a non-empty prefix u'. We have $u' \leq u$ by the constraints of merging trees.

Let r_l be the number of times that $\mathcal{L}_l$ will be repeated, let r be the additional number of repetitions of $\mathcal{T}^i$. By dissociating the two, we are going to prove at the same time that more complex repetitions loops do not bring an advantage.

We have $r_l \geq u'$ and $r \geq u$, and the total number of repetitions of this subtree $\mathcal{T}^i$ is at least $r_l + r$. We will not write the total time complexity of the algorithm, because the other subtrees and loops intervene as well, but we focus on the terms that are related to $\mathcal{T}^i$. We use placeholders (*) for the terms that remain unchanged.

$$(*) \cdot \left(2^{\frac{nr_l}{2}} \left(\text{Build } \mathcal{L}_l + 2^{\frac{nr}{2}} \left(\text{Build } \mathcal{T}^i + (*) \right) \right) \right) .$$

Let t_l be the time to build $\mathcal{L}_l$, ℓ_l its size, t_s the time to sample $\mathcal{L}_s$. We rewrite this as:

$$(*) \cdot \left(2^{\frac{n(r_l - u') + nu'}{2}} \left(2^{nt_l} + 2^{\frac{nr}{2}} \left(2^{\frac{(u-u'-\ell_l)n}{2}} 2^{t_s n} \cdot 2^{n\ell} + (*) \right) \right) \right) .$$

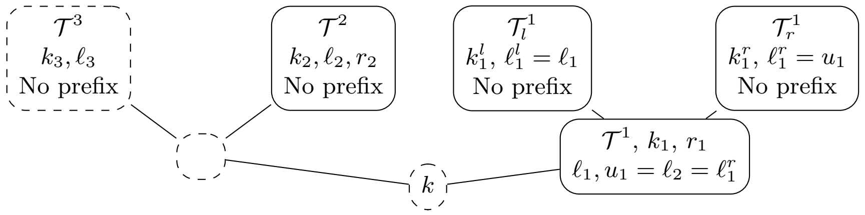


Fig. 10. Extended merging tree with three levels and a single non-empty prefix.

Now, let us simply remove the prefix u' from both $\mathcal{L}_s$ and $\mathcal{L}_l$. We want a list $\mathcal{L}_l$ of same size as before. This is possible by reducing the width of the subtree and / or taking a sublist of it (in which case it adds a small repetition loop). In return, we increase the size of $\mathcal{L}_s$ so that it forms a bigger search space.

We do not have to loop on u' anymore, although there are still $r_l - u' \geq 0$ repetitions to take into account. The terms t_s and t_l are replaced by $t'_s \leq t_s$ and $t'_l \leq t_l$. The complexity becomes:

$$(*) \cdot \left(2^{\frac{n(r_l - u')}{2}} \left(2^{nt'_l} + 2^{\frac{nr}{2}} \left(2^{\frac{(u-\ell_l)n}{2}} 2^{t'_s n} \cdot 2^{n\ell} + (*) \right) \right) \right) .$$

The only term that has increased here is the sampling of an element in the root of $\mathcal{T}^i$. Since the prefix condition on u' is removed, we have to iterate the quantum search $2^{\frac{(u-\ell_l)n}{2}}$ times instead of $2^{\frac{(u-u'-\ell_l)n}{2}}$ times. But this increase *is balanced with the removal of u'*.

Thus, any non-empty prefixes inside the main subtrees can be removed, by increasing the sampling time of these subtrees instead. If we focus on this particular example $\mathcal{T}^i$, then the list child $\mathcal{L}_l$ is simply a list of unconstrained sums of elements, and so is the sampled child $\mathcal{L}_s$.

We can also make another remark. If u is nonzero, then it must be equal to ℓ_l. Indeed, if u was smaller, then we might as well decrease the size of $\mathcal{L}_l$ and reduce the time complexity. But if it was bigger, we would pay a term $2^{(u-\ell_l)n/2}$ for each element of $\mathcal{T}^i$ produced, in addition to the repetition term $2^{u/2}$. Thus we might as well make the root list of $\mathcal{T}^i$ bigger to compensate. $\square$

Next, we can show that the main branch of this optimal tree has, actually, only three levels. This is represented on Figure 10.

Lemma 9. *For any k, the optimal time complexity is reached by a tree where only two nodes (children of the root) have a non-empty prefix.*

Proof. Let us consider a tree with four levels, with two non-trivial prefixes u_1 (at level 1) and u_2 (at level 2). As an illustration, we can picture a tree like in Figure 10 but with a non-empty prefix u_2 in $\mathcal{T}^2$.

We have at least two repetition loops: the outer one in which we choose u_1, then build $\mathcal{T}^1$ (of size ℓ_1), and the inner one in which we choose u_2, then build $\mathcal{T}^2$ (of size ℓ_2). Inside all these loops, there is a final term corresponding to the quantum search at the root. This term contains a factor $2^{(u_1-u_2-\ell_1)n/2}$ corresponding to the search of a matching element in $\mathcal{T}^1$, when we try to compute the root of the merging tree.

Similarly to the proof of Lemma 8, we will now remove the prefix u_2, but keep the size of $\mathcal{T}^2$ unchanged. As a result, we have to increase the search space for the last quantum search. The term $2^{(u_1-u_2-\ell_1)n/2}$ becomes $2^{(u_1-\ell_1)n/2}$, because we don't have the prefix u_2 anymore. However, this is balanced by the removal of $2^{u_2 n/2}$ quantum search iterates. □

E.3 Step 2: Solving the Constraints

We are now considering the simple tree shape of Figure 10. There are only a constant numbers of variables, some of which are integer: the shape of the tree is determined by k_3, k_2, k_1^l, k_1^r. The other parameters are ℓ_3, ℓ_2, ℓ_1 and $u_1 = \ell_2$ by optimization. Overall, the structure of the corresponding algorithm is similar to Algorithm 3. Let t be its time complexity.

Computing products of lists (i.e., nodes at level 2 in the tree) outside or inside the repetition loops makes a change in the constraints. We find that it is better to create them outside. There are possibly two repetition loops, with variables r_1 and r_2:

$$r_1 = u_1 + \frac{k_1^l}{k} - \ell_1 + \frac{k_1^r}{k} - \ell_2 = \frac{k_1^l}{k} - \ell_1 + \frac{k_1^r}{k} \quad \text{and} \quad r_2 = \frac{k_2}{k} - \ell_2 \ , \tag{16}$$

and t satisfies the constraints:

$t \geq \max\left(\frac{k_2}{k}, \frac{k_1^l}{k}\right)$	Computation of product lists
$\frac{k_1^r}{k} \geq \ell_2, \frac{k_1^l}{k} \geq \ell_1, \frac{k_2}{k} \geq \ell_2$	Limitations on the list sizes
$t \geq \frac{r_1}{2} + \ell_1$	Total workload of $\mathcal{T}^1$
$t \geq \frac{r_1}{2} + \frac{r_2}{2} + \frac{1}{2}\left(\frac{k_3}{k}\right)$	Final search in $\mathcal{T}^3$

which gives:

$$\text{(C1)} \quad t \geq \max\left(\frac{k_2}{k}, \frac{k_1 - k_1^r}{k}\right)$$

$$\text{(C2)} \quad \frac{k_1^r}{k} \geq \ell_2, \frac{k_1 - k_1^r}{k} \geq \ell_1, \frac{k_2}{k} \geq \ell_2$$

$$\text{(C3)} \quad t \geq \frac{1}{2}\left(\frac{k_1^l}{k} - \ell_1 + \frac{k_1^r}{k}\right) + \ell_1 \implies t \geq \frac{1}{2}\left(\frac{k_1}{k} + \ell_1\right)$$

$$\text{(C4)} \quad t \geq \frac{1}{2}(1 - \ell_1 - \ell_2)$$

We will now write a smaller set of constraints without the variables ℓ_1 and ℓ_2, and show that they are sufficient. From (C4), (C2) and (C3) we obtain: $4t \geq 1 + \frac{k_1 - k_2}{k}$. We also keep $t \geq \frac{k_2}{k}$. From (C4) and (C2) we obtain: $2t \geq 1 - \ell_1 - \ell_2 \geq 1 - \frac{k_1}{k}$.

To show that these constraints are sufficient, we need to exhibit a tree that reaches the prescribed complexity for any choice of k_1, k_2.

Lemma 10. *Let k_1, k_2 be such that $k_1 + k_2 \leq k$. Then there exists an extended merging tree algorithm solving Single-solution k-XOR in time (exponent):*

$$t = \max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right), \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right) \quad . \tag{17}$$

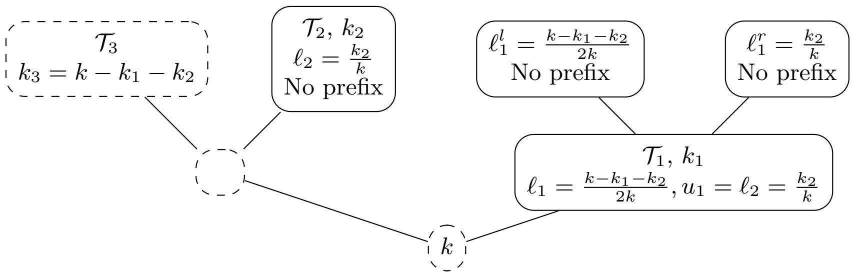


Fig. 11. Tree of Lemma 10.

Proof. First of all, consider the case $k_1 \leq k_2$, i.e., the subtree at level 2 is bigger than the subtree at level 1. This implies in particular $t \geq \frac{k_2}{k} \geq \frac{k_1}{k}$ and $t \geq \frac{1}{2}(1 - \frac{k_1}{k})$, thus $t \geq \frac{1}{3}$: this is unlikely to be a good parameter choice. In that case, we must find $t \leq \max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right)\right)$. This is easily obtained with a trivial tree, that has only two subtrees: one is obtained by the product of k_1 lists (time $\frac{k_1}{k} \leq \frac{k_2}{k}$), and the other is an exhaustive search over all $k - k_1$ remaining lists, in time $\frac{1}{2}\left(1 - \frac{k_1}{k}\right)$. There are no repetitions. Notice that this is actually the optimal strategy for $k = 3, 6$ with $k_1 = k_2 = \frac{k}{3}$.

So we can now suppose that $k_1 \geq k_2$. We notice that:

$$\frac{1}{2}\left(1 - \frac{k_1}{k}\right) \geq \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right) \iff 1 - \frac{2k_1}{k} \geq \frac{k_1 - k_2}{k} \iff k \geq 3k_1 - k_2 \quad .$$

So we next focus on the case $k \leq 3k_1 - k_2$ and we try to obtain a complexity $t \leq \max\left(\frac{k_2}{k}, \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right)$. The merging tree that we use is drawn on Figure 11. We attach two subtrees of width k_2 and k_1 to the main branch, there remains a subtree of width $k - k_1 - k_2$ to explore exhaustively. We build the subtree $\mathcal{T}_2$ in time $\frac{k_2}{k}$, externally, by constructing the product of k_2 lists. Then we repeat $\mathcal{T}_1$, which builds a list of size $\ell_1 = \frac{k - k_1 - k_2}{2k}$.

With our choice of parameters, we have to iterate:

$$\frac{1}{2}\left(\frac{k_1}{k} - \ell_1\right) = \frac{1}{4k}\left(2k_1 - k + k_1 + k_2\right) = \frac{1}{4k}\left(3k_1 + k_2 - k\right)$$

times, which is positive, since $k \leq 3k_1 - k_2 \leq 3k_1 + k_2$. In each iteration, we build the subtree $\mathcal{T}_1$ in time ℓ_1 and exhaust the subtree $\mathcal{T}_3$ with quantum search, with the same time. Thus, the total time complexity is given by:

$$t = \max\left(\frac{k_2}{k}, \frac{1}{2}\left(\frac{k_1}{k} + \ell_1\right)\right) = \max\left(\frac{k_2}{k}, \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right) \quad .$$

Finally, in the case $3k_1 + k_2 \leq k$, we notice that $3k_1 \leq k$ implies $\frac{1}{2}\left(1 - \frac{k_1}{k}\right) \geq \frac{1}{3}$, as it happened before for the case $k_1 \leq k_2$. Further, $\frac{1}{2}\left(1 - \frac{k_1}{k}\right) \geq \frac{k_1}{k}$. The same strategy works: we build an intermediate subtree with a product of k_1 lists, in time $\frac{k_1}{k}$, then look for a collision on it with a single quantum search in the product of the $k - k_1$ remaining lists.

Thus, regardless the choice of k_1 and k_2, we can meet the time complexity given by Equation (17). □

Finally, we observe that the minimization over k_1, k_2 of this quantity gives the formula of γ_k of Theorem 3, finishing the proof of the theorem:

$$\min_{\substack{k_1, k_2 \in \mathbb{N}^2 \\ k_1 + k_2 \leq k}} \max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right), \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right) = \frac{k + \left\lfloor \frac{k+6}{7} \right\rfloor + \left\lfloor \frac{k+1}{7} \right\rfloor - \left\lfloor \frac{k}{7} \right\rfloor}{4k}$$

The minimum can be reached with the parameters given in Equation 5.

F Limiting the List Size

In this section, we study the time-memory product of merging trees when the list size is fixed to $2^{n/k}$. This corresponds to a situation where we solve the k-XOR problem with lists: the memory is at least $2^{n/k}$ since this is the size of the initial lists given to the algorithm. In the "oracle" version, there are some corner cases in which the memory can even decrease below $2^{n/k}$. When optimizing for the time-memory product, we observe this memory limitation, which is the same as in [27]. It seems that in general, increasing the list sizes places a higher burden on the memory complexity that it improves the time.

First of all, we remark that the result of Lemma 8 (all subtrees of the main branch are trivial) remains valid: even if we add the memory to the objective value, removing the internal prefixes can only decrease this objective.

Under this new constraint, we can write down the shape of an optimal tree as follows:

- it has a main branch with p levels, subtrees $\mathcal{T}_i$, and $\mathcal{T}_p$ is a leaf node with $u_p = 0$. Each $\mathcal{T}_i$ is of width k_i, and the k_i decrease (otherwise, exchanging two of the subtrees would reduce the time complexity)
- at level p on the left, there is a subtree $\mathcal{T}'_p$ which is of width k'_p, with $\sum_{i=1}^{p} k_i + k'_p = k$.
- each $\mathcal{T}_i$ induces a repetition loop with $r_i = \frac{1}{2}\left(\frac{k_i}{k} - \ell_i\right)$. Since the ℓ_i are of size $\frac{1}{k}$ at most, and k_i decreases with i, the number of repetitions decreases with i

- $\forall i, \ell_i \leq \frac{1}{k}$ and $u_{i-1} = u_i + \ell_i$ (as before), so we still have $u_i = \sum_{j=i+1}^{p} \ell_j$. Furthermore $u_i + \ell_i \leq \frac{k_i}{k}$

Thus the time complexity t satisfies the constraints:

$$\begin{cases} \forall i, \frac{1}{k} \geq \ell_i \geq 0, \frac{k_i}{k} \geq \sum_{j=i}^{p} \ell_j \\ t \geq \sum_i r_i + \frac{1}{2}\left(1 - \frac{\sum_i k_i}{k}\right) = \frac{1}{2}\left(1 - \sum_i \ell_i\right) \\ \forall 1 \leq i \leq p-1, t \geq \left(\sum_{j=1}^{i} r_j\right) + \frac{1}{2}\max\left(\left(\sum_{j=i+1}^{p} \ell_j\right) - \frac{\lfloor \log_2 k_i \rfloor}{k}, 0\right) + \ell_i \end{cases} \tag{18}$$

Contrary to the previous minimization, we will find that the depth of the tree increases with k, and the merging is not as trivial as before. By bounding the ℓ_i, we obtain that $\frac{k_1}{k} \geq \sum_{j=1}^{p} \ell_j$ and $\frac{p}{k} \geq \sum_{j=1}^{p} \ell_j$. Thus, t is lower bounded by:

$$t \geq \frac{1}{2}\left(1 - \max\left(\frac{k_1}{k}, \frac{p}{k}\right)\right) \quad .$$

This is to say, the tree should have the bigger depth possible, while at the same time, having the widest possible subtree at level 1. In practice, we observe that the constraints (18) are sufficient to obtain the best trees. Both k_1 and the number of levels in the tree are of order $\mathcal{O}\left(\sqrt{k}\right)$, which is why we will observe a decrease of the value kt, then an increase. The minimum $k = 17$ is obtained by numerical experiments.