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. Applications include 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.
The generic study of quantum algorithms k-XOR (and variants) was started by Grassi *et al.* (ASIACRYPT 2018), in the case where many solutions exist. At EUROCRYPT 2020, Naya-Plasencia and Schrottenloher defined a class of *quantum merging algorithms* 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 new, simplified representation of *merging trees* that makes their analysis easier. As a consequence, we improve the quantum time complexity of 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 generic 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.

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

The numerous applications of collision algorithms may be explained by the fact that there exist many related problems, such as claw-finding (where we want to make the outputs of *two* different functions collide), which can be solved with the same techniques. But there are also many generalizations of collision search that arise in different contexts. A possible generalization would be to look for more than two elements having the same image: the problem (*multicollision* search) then becomes harder. Another would be to have *more than two elements collide* in the sense that their combination satisfies some condition. This leads to the *Generalized Birthday Problem*, or k-XOR for us, formulated by Wagner [30]:

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

The problem is open to many variants and generalizations. The $\oplus$ operation can be replaced by modular additions or even non-commutative group operations. Likewise, we may limit the sizes of the lists queried. Some of these cases have been considered by Wagner [30], and many more have occurred in the literature afterwards.

In [30], 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 attempt at a generic study. Later on, many works have either pursued the generic direction [27, 12], or the optimization of more specific algorithms falling in the same framework. For example, the best algorithms for randomized instances of subset-sum [17, 3, 6] actually solve k-list problems with additional constraints, and use *merging* as an algorithmic subroutine.

Quantum k-XOR Algorithms. In the context of post-quantum security, we need a similar level of understanding of *quantum* algorithms to solve the k-XOR problem. We already know that quantum k-list algorithms play a role in generic decoding [19] or in lattice sieving [20], which is why it is important to know how far the "generic" advantage can go.

Grassi *et al.* [15] tackled the Many-solutions case for a generic k, and obtained some quantum time speedups when the elements are produced by a random function with quantum oracle access. More complete results were obtained in [25]. 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, the authors studied the Single-solution case for a generic k. Previous results had shown that, contrary to what occurs in the classical case, the Single-solution k-XOR problem has a time complexity advantage when k increases. More precisely, the Single-solution 2-XOR problem has a quantum time complexity $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$ [1], and an algorithm of time complexity $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$ [5] for the 4-XOR problem was known. In [25], many more algorithms were given and a closed formula for the time complexity exponent, depending on k, showed a convergence towards 0.3, which could then be conjectured to be a lower bound.

Contributions. We improve the study of [25] in different ways. First of all, we give a new definition of *merging trees*. In the case where many solutions exist, we recover the results of [25] and give different, more succinct proofs of their optimality in the class of merging trees. Going to the single-solution case, we modify one of the constraints enforced in [25]. This allows us to obtain a new closed formula, which converges towards 2/7 instead of 0.3 (and reaches it for 7-XOR). Finally, we remark that quantum walks can be used 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 $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$. All these algorithms can be easily described.

Organization of the Paper. We open the paper with a formal definition of the k-XOR problem and a presentation of *classical* merging algorithms in Section 2. In Section 3, we introduce some preliminaries of quantum computing, quantum memory models and quantum search. In Section 4, we give our new definition of *merging trees* and prove the correspondence between these abstract objects and quantum algorithms for Many-solutions k-XOR. In Section 5, we briefly explain how the trees are extended to the Single-solution case, and we give some of our new results. Next, in Section 6, we introduce quantum walk algorithms for Claw-finding as black-box tools and use them to get our best exponents. In Section 7, we show that these algorithms can be applied to all *bicomposite* problems, in particular multiple-encryption.

2 Classical Merging Algorithms

In this section, we define the k-XOR problem and review Wagner's algorithm [30].

2.1 k-XOR Problem and Extensions

We focus on a simple variant of the Generalized Birthday Problem, where the data is generated by a single random function h.

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 $h(x_1) \oplus \ldots \oplus h(x_k) = 0$.

Throughout this paper, we will assume that *quantum* access to h is given. We will also consider the problem with a single solution on average, by restricting the codomain of h.

Problem 2 (Single-solution k-XOR). Given oracle access to a random function $h : \{0,1\}^{n/k} \to \{0,1\}^n$, find distinct inputs $(x_1, \ldots, x_k)$ such that $h(x_1) \oplus \ldots \oplus h(x_k) = 0$.

Note that in this case, having quantum access to h is not a strong restriction, since the whole function can be queried at the beginning of the algorithm.

We will use the term "k-XOR" to refer either to Problem 1 or Problem 2, as opposed to the original "Generalized Birthday" formulation recalled in the introduction. We name "k-SUM" the problem where $\oplus$ is replaced by addition modulo 2^n. Since we take k as a constant, and consider asymptotic complexities in n, we will assume that n is divisible by k (or by any constant depending on k) and write n/k instead of $\lceil n/k \rceil$. Finally, since we formulate the problem with a random function, a solution might not exist. Since our algorithms are probabilistic, we include this as a case of failure.

Extension to other operations. Since all the algorithms that we will study are based on the *merging* subroutine (and its quantum version), we can modify the problem as long as *merging* has an appropriate definition. This was already remarked by Wagner [30]. We choose to restrict our generic study to the XOR for simplicity, but we will consider other operations in our applications.

Query Complexity. The classical query complexity of the k-XOR problem, Single- or Many-solutions, is $\Omega(2^{n/k})$. When h is a random function, 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 [31] 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 $\mathcal{O}(2^{\alpha_k n})$ where the exponent α_k depends only on k. The quantum algorithms that we will present are based on Grover's quantum search algorithm [16] and on quantum walks [23], which achieve at most a quadratic speedup. So this is the best that we can expect and, in practice, it will be less. This is already well known, for example in the case of collision search ($k = 2$, Many- or Single-solution case) where the complexity goes from $\mathcal{O}\big(2^{n/2}\big)$ classically to $\widetilde{\mathcal{O}}\big(2^{n/3}\big)$ quantumly [7, 1].

Conventions. Since all the complexities studied in this paper are exponential in n, we focus on the exponents. We use the $\widetilde{\mathcal{O}}$ notation to hide polynomial factors, which will usually be logarithmic. 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 ℓ_i are constants.

2.2 Classical Merging

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 $\mathcal{L}_1 \bowtie_u \mathcal{L}_2$ 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.

Remark 1. In this paper, t will either have an arbitrary value, or take all values in $\{0,1\}^{un}$, which is why the notation focuses on the parameter u.

Remark 2. Computationally, the join list will contain rather three-tuples $(x_1, x_2, x_1 \oplus x_2)$, so that we keep the knowledge of x_1 and x_2.

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. The algorithm for merging lists pairwise is given in Algorithm 1.1.

Algorithm 1.1 Classical merging.

```
    Input: $\mathcal{L}_1$, $\mathcal{L}_2$ sorted lists of $n$-bit strings of sizes $L_1, L_2$, prefix $t$ of $un$
    bits. Elements of these lists are denoted $x_{1,i_1}$ and $x_{2,i_2}$ respectively. We use
    lexicographic ordering.
    Output: Sorted join list $\mathcal{L}_u = \mathcal{L}_1 \bowtie_u \mathcal{L}_2$
 1: function MERGE($\mathcal{L}_1$, $\mathcal{L}_2$, $t$)
 2:     Set $b, s = 1, 2$ if $L_1 > L_2$ and $2, 1$ otherwise
 3:     If $t$ is not provided, set $t = 0_{un}$
 4:     $\mathcal{L}_s \leftarrow \{x \oplus (t \| 0), x \in \mathcal{L}_s\}$
 5:     $i_1 \leftarrow 0, i_2 \leftarrow 0$
 6:     $\mathcal{L}_u \leftarrow \varnothing$
 7:     while $i_1 \leqslant L_1$ and $i_2 \leqslant L_2$ do
 8:         if $x_{1,i_1}|_{un} > x_{2,i_2}|_{un}$ then
 9:             Increment $i_2$ until $x_{1,i_1}|_{un} \leqslant x_{2,i_2}|_{un}$
10:         else if $x_{1,i_1}|_{un} < x_{2,i_2}|_{un}$ then
11:             Increment $i_1$ until $x_{1,i_1}|_{un} \geqslant x_{2,i_2}|_{un}$
12:         else
13:             $\mathcal{L}_u \leftarrow \mathcal{L}_u \cup \{(x_{b,i_b}, x_{s,i_s} \oplus t)\}$
    return $\mathcal{L}_u$
```

The result of the merging operation 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 [24, Section 4].

In this paper, we consider the following heuristic, which is enough to ensure the correctness of our algorithms. We show in Appendix A that our quantum algorithms work as well without.

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). *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 such* *that* $\mathbb{E}(L_u) = \frac{L_1 L_2}{2^{un}}$. *Under Heuristic 1, the deviation from its expectation is exponentially small.*

2.3 Wagner's Algorithm

Judging by the time complexity exponent only, the best classical procedure to solve Problem 1 remains, to date, Wagner's algorithm [30]. It uses a recursive *merging strategy* which can be 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. An example for $k = 4$ is given in Figure 1.

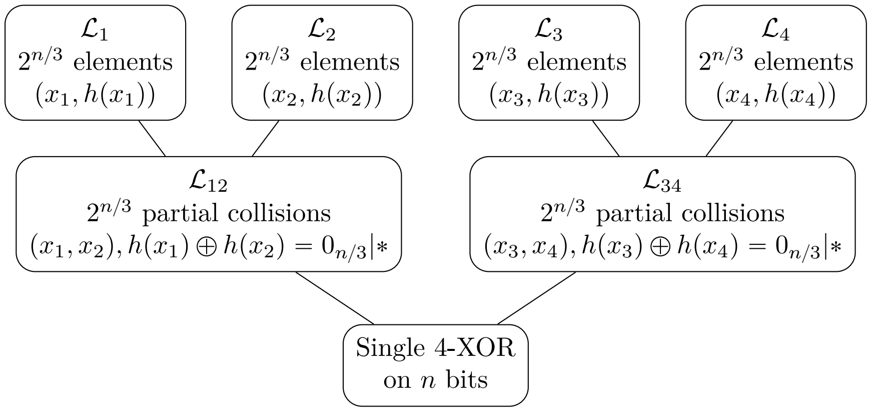

$\mathcal{L}_1$ $2^{n/3}$ elements $(x_1, h(x_1))$
$\mathcal{L}_2$ $2^{n/3}$ elements $(x_2, h(x_2))$
$\mathcal{L}_3$ $2^{n/3}$ elements $(x_3, h(x_3))$
$\mathcal{L}_4$ $2^{n/3}$ elements $(x_4, h(x_4))$
$\mathcal{L}_{12}$ $2^{n/3}$ partial collisions $(x_1, x_2), h(x_1) \oplus h(x_2) = 0_{n/3}|*$
$\mathcal{L}_{34}$ $2^{n/3}$ partial collisions $(x_3, x_4), h(x_3) \oplus h(x_4) = 0_{n/3}|*$
Single 4-XOR on n bits

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

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. Using the MERGE function recursively, for any such merging tree, there exists a k-XOR algorithm 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 $\frac{1}{\lfloor \log_2(k) \rfloor + 1}$.

2.4 Depth-first Computation of Lists

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 (Figure 2).

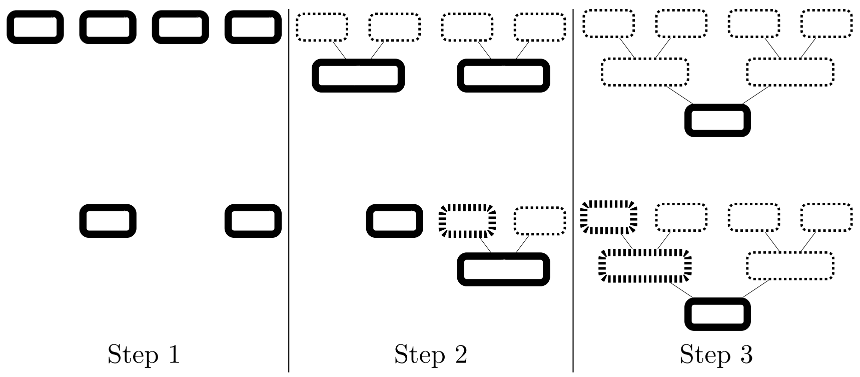


Fig. 2. Building the 4-XOR tree of Figure 1 in a breadth-first (above) or depth-first manner (below) (the figure is taken from [25]). The new nodes are put in bold. Between two steps, only the lists in bold are stored. Dotted lists are either discarded at this step, or do not need to be stored at all.

This depth-first traversal has also the effect of rewriting a sequence of Merge procedures as a sequence of Sample procedures, where Sample procedures are defined for each list in the tree as follows:

- If the list $\mathcal{L}$ is a leaf node, then Sample($\mathcal{L}$) consists in making an arbitrary query to h and returning $(x, h(x))$
- Otherwise, for $\mathcal{L}_u = \mathcal{L}_1 \bowtie_u \mathcal{L}_2$: we assume that the *intermediate list* $\mathcal{L}_2$ has been built, and that a Sample for $\mathcal{L}_1$ has been defined. Then Sample($\mathcal{L}_u$) simply consists in sampling $x_1 \in \mathcal{L}_1$, and looking for an element in $x_2 \in \mathcal{L}_2$ 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, replacing the Sample function by a more efficient quantum procedure is much more easy than the Merge function. This was remarked in [25] and leads to the definition of *quantum merging algorithms.*

3 Quantum Preliminaries

In this section, we regroup some technical preliminaries necessary for the quantum algorithms studied in this paper. We stress that no technical knowledge of quantum computing is required, as we will only use well-known black-boxes. In particular, we will focus here on *quantum search*, and defer the definition of quantum walks to Section 6.

3.1 Quantum Algorithms

Our algorithms are ultimately written in the quantum circuit model (see [26] for an introduction), but we stand on a higher level of abstraction. A quantum circuit is defined as a sequence of *quantum gates* acting on a fixed set of *qubits*. The state of an individual qubit is represented as a normalized vector in a two-dimensional Hilbert space, and we denote by $|0\rangle, |1\rangle$ an arbitrary basis. The state of a system of n qubits lies in $\mathbb{C}^{2^n}$, with a basis $(|i\rangle, 0 \leqslant i \leqslant n-1)$ representing all possibilities for the joint state of the n qubits. The system evolves through unitary operators of $\mathbb{C}^{2^n}$.

Quantum gates are elementary operations, that is, fixed unitary operators applying to one or two qubits at a time. We only assume that we rely on a standard universal gate set, as all are equivalent up to a polynomial factor.

For an algorithm $\mathcal{A}$, we let $\mathsf{T_q}(\mathcal{A})$ denote its quantum time complexity, the number of gates in the circuit, and $\mathsf{M}(\mathcal{A})$ the number of qubits used.

Sometimes, some of the qubits are actually *bits*, e.g., when we use a classical value to control a quantum operation applied. This prompts us to make a difference between the *quantum* and *classical* memories used in a quantum algorithm. In this paper, we will use three types of memory:

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

$$|i\rangle |y\rangle \xmapsto{\mathsf{qRAM}} |i\rangle |y \oplus M_i\rangle$$

  where M_i is the data at index i. That is, all memory cells can be accessed simultaneously in superposition.
- Quantum memory with quantum random-access (QRAQM)[2]: it also allows superposition access, but the data can be a quantum state:

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

- Classical memory: it contains classical data with classical random access, and we can use this data to control quantum gates classically. Since no qRAM gate is available, we say that we use the *(plain) quantum circuit model.*

QRACM and QRAQM are ubiquitous in the quantum computing literature, although often implicit. They are required by many algorithms, including most k-XOR algorithms of [25]. The QRACM/QRAQM terminology is borrowed from [21], and the qRAM gate is defined in [1, Section 6.1].

Although the "full' QRAQM and QRACM models may seem very powerful at first sight, they are simply conservative in terms of security, as the implementations of random access for quantum architectures are still blurry. Besides, there are cases in which QRAQM can be replaced by QRACM, and QRACM by QRAQM:

[1]QACM in [25].
[2]QAQM in [25].

- In [25], the quantum Many-solutions k-XOR algorithms require the QRACM model only. In the previous work [15], different algorithms of similar complexities were obtained *with QRAQM*;
- An algorithm that requires QRACM, but makes only a few queries to it, can be placed in the standard circuit model by replacing all QRACM queries with *sequential lookup circuits*. A qRAM gate spanning M data registers can be replaced by $\widetilde{\mathcal{O}}(M)$ standard quantum gates.

3.2 Quantum Search

Grover's quantum search [16] 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. [8]. It speeds up the search for a "good" output of *any* probabilistic algorithm.

Theorem 1 ([8], Theorem 2). *Let $\mathcal{A}$ be a quantum algorithm that uses no 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:*

$$\left\lfloor \frac{\pi}{4\theta_a} \right\rfloor (2|\mathcal{A}| + |O_f|)$$

that obtains a good result with success probability greater than $\max(1-a, a)$.

We define a *quantum sampling* black-box, analogous to a classical algorithm **Sample** whose task would be to sample uniformly at random from some well-defined set.

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).*

In our case, the sets will be lists in a merging trees, and quantum sampling algorithms will be obtained using Amplitude Amplification, with other quantum samplings as subroutines. With Heuristic 1, we do not only ensure that the solution exists with high probability, but also, that the list sizes do not deviate much from their expectation. This ensures that all our quantum searches can run with the *expected* number of iterations, and still succeed significantly. If the deviation is higher (say, a constant), a way to fix this is to make several copies of each $\mathsf{qSample}()$ in order to dismiss their errors, but this incurs polynomial time factors. We elaborate on the time complexities *without the heuristic* in Appendix A.

4 Quantum Merging Algorithms

As we recalled in Section 2.4, writing a classical merging procedure in a depth-first manner allows to replace the Merging subroutine by the Sample one, which is essentially an exhaustive search. Following [25], we then replace each Sample by a quantum algorithm $\mathsf{qSample}$, using quantum search. This is what we do in Section 4.1 below, where we define the building block of quantum merging algorithms.

In [25], a framework of *merging trees* is presented. This essentially consists in taking trees that represent merging algorithms, and computing their complexity depending on their parameters. The set of constraints is adapted to take quantum search into account. We provide in Section 4.2 a description of merging trees that simplifies the one from [25].

Possible extensions of this framework are discussed in [25]. None of the classical techniques of [12, 2, 13, 27, 24] seem to bring further improvement to the k-XOR problem in the quantum setting.

4.1 Merging in the Quantum Setting

Quantum merging algorithms are based on a result analogous to Lemma 1: if the list $\mathcal{L}_2$ is given, then from a procedure that samples from the list $\mathcal{L}_1$, we can create a procedure that samples from $\mathcal{L}_u$ by simply trying to *match* the elements of $\mathcal{L}_1$ against $\mathcal{L}_2$.

Lemma 2 (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:

$$\mathsf{T_q}\left(\mathsf{qSample}(\mathcal{L}_u)\right) = \begin{cases} \left(\mathsf{T_q}\left(\mathsf{qSample}(\mathcal{L}_1)\right) + \mathcal{O}(n)\right) \cdot \max(2^{\frac{(u-\ell_2)}{2}n}, 1) \text{ with QRACM} \\ \left(\mathsf{T_q}\left(\mathsf{qSample}(\mathcal{L}_1)\right) + 2^{\ell_1 n}\right) \cdot \max(2^{\frac{(u-\ell_2)}{2}n}, 1) \text{ without} \end{cases} \tag{1}$$

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 $2^{(u-\ell_2)n}$ iterations up to a constant. Each iteration calls $\mathsf{qSample}(\mathcal{L}_1)$ (setup) and queries the memory.

Without the QRACM, we use a circuit that performs a sequence of $2^{\ell_2 n}$ classically controlled comparisons.

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 ordering the QRACM in a radix tree. □

As an illustration, let us recall the 4-XOR example of [25]. Translating the depth-first classical 4-XOR algorithm (Figure 1) to a quantum one gives the following heuristic complexity:

$$\underbrace{2^{n/3}}_{\mathcal{L}_2} + \underbrace{2^{n/3}}_{\mathcal{L}_4} + \underbrace{2^{n/3}}_{\mathcal{L}_{34}} + \underbrace{\mathcal{O}\left(\sqrt{2^{n/3}}\right)}_{\text{Final sample}} = \mathcal{O}\left(2^{n/3}\right) \tag{2}$$

where the final qSample procedure requires QRACM access to $\mathcal{L}_2$ and $\mathcal{L}_{34}$, and superposition queries to h. Writing down the lists $\mathcal{L}_2, \mathcal{L}_4, \mathcal{L}_{34}$ is a step that we do not expect to accelerate, and this is the bottleneck of the algorithm.

Though it is not necessary to change the structure of the tree, this situation calls for an *inherently quantum* re-optimization of the parameters. This is displayed in Figure 3. We will decrease the size of the lists $\mathcal{L}_2, \mathcal{L}_4, \mathcal{L}_{34}$ and increase the size of $\mathcal{L}_1$ and $\mathcal{L}_{12}$. The complexity becomes:

$$\underbrace{2^{n/4}}_{\mathcal{L}_2} + \underbrace{2^{n/4}}_{\mathcal{L}_4} + \underbrace{2^{n/4}}_{\mathcal{L}_{34}} + \underbrace{\mathcal{O}\left(\sqrt{2^{n/2}}\right)}_{\text{Final sample}} = \mathcal{O}\left(2^{n/4}\right) . \tag{3}$$

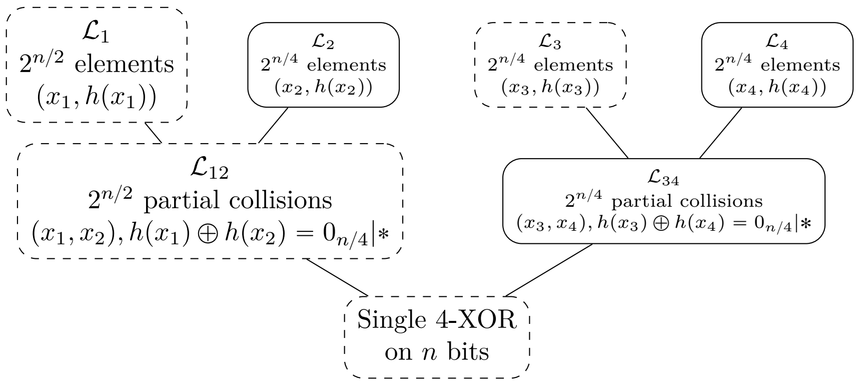


Fig. 3. Re-optimization of 4-XOR merging. Dashed lists correspond to nested QSamples.

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. We emphasize that from now on, our representation will differ from [25].

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 on Figure 3. 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 representing the attributes of $\mathcal{L}_i^j$:

- 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 $(x_1, \dots, x_{k_i^j})$ such that $x_1 \oplus \dots \oplus x_{k_i^j} = 0_{u_i^j n}|*$, 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 constraint 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 proportion to the minimum between them, obtaining an easier strategy.

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

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). $\forall i, j \geqslant 1, \ell_i^{j-1} = \ell_{2i}^j + \ell_{2i+1}^j - (u_i^{j-1} - u_{2i+1}^j)$.

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:*

$$t_i^j = \begin{cases} t_{2i}^{j+1} + \frac{1}{2}\max\left(u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0\right) & \textit{with QRACM} \\ \max(t_{2i}^{j+1}, \ell_{2i+1}^{j+1}) + \frac{1}{2}\max\left((u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0\right) & \textit{without} \\ t_{2i}^{j+1} + \max\left(u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0\right) & \textit{classically} \end{cases} \tag{4}$$

If the node is a leaf, then we simply run Grover's algorithm multiple times. Equation (4) is simply a translation of (1) in the case of a specific node. The third option needs to be added when qRAM 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 its parent Sample is also a classical one. Next, from the individual sampling times of each node, we can compute what should be 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:*

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

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.

With these definitions, and with the help of Lemma 2, we can make merging trees correspond to merging algorithms.

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 O_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 O_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).

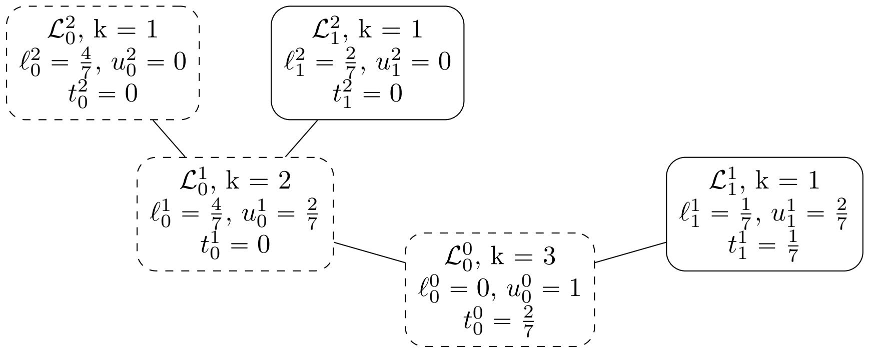
Proof. We define recursively the correspondence $\mathcal{T}_k \overset{\mathcal{A}}{\longmapsto} \mathcal{A}(\mathcal{T}_k)$ from a merging tree $\mathcal{T}_k$ to an algorithm $\mathcal{A}(\mathcal{T}_k)$ solving the k-XOR problem, with the wanted time and memory complexities.

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:
  1. we use $\mathcal{A}(L)$ to sample repeatedly from the child L: we build the list. Each call costs time $\mathcal{O}\left(2^{\mathsf{T}_\mathsf{q}(L)n}\right)$ and we need to make $\mathcal{O}(2^{\ell'' n})$ of them to obtain a list of the wanted size, with high probability.
  2. then we apply Lemma 2, since we have a sample for the child S: $\mathcal{A}(S)$
- If it is a List, the situation is the same, except that we repeat the operation exponentially many times.

When taking sums of complexities exponential in n, we write $\mathcal{O}(2^{\alpha n})+\mathcal{O}\left(2^{\beta n}\right) = \mathcal{O}\left(2^{\max(\alpha,\beta)n}\right)$, which remains sound since we do that only a constant number of times. A global factor $\mathcal{O}(n)$ comes from the memory operations. $\square$

Thus, merging trees offer a compact and sound way to represent quantum merging algorithms for the k-XOR problem. As an example, we represent the optimal merging tree for 3-XOR of [25] on Figure 4.



$\mathcal{L}_0^2$, k = 1
$\ell_0^2 = \frac{4}{7}$, $u_0^2 = 0$
$t_0^2 = 0$
$\mathcal{L}_1^2$, k = 1
$\ell_1^2 = \frac{2}{7}$, $u_1^2 = 0$
$t_1^2 = 0$
$\mathcal{L}_0^1$, k = 2
$\ell_0^1 = \frac{4}{7}$, $u_0^1 = \frac{2}{7}$
$t_0^1 = 0$
$\mathcal{L}_1^1$, k = 1
$\ell_1^1 = \frac{1}{7}$, $u_1^1 = \frac{2}{7}$
$t_1^1 = \frac{1}{7}$
$\mathcal{L}_0^0$, k = 3
$\ell_0^0 = 0$, $u_0^0 = 1$
$t_0^0 = \frac{2}{7}$

Fig. 4. 3-XOR (optimal) merging tree with QRACM.

4.4 Optimal Trees for 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 $\mathsf{T}_\mathsf{q}\left(\mathcal{T}_k\right)$.

Given a tree $\mathcal{T}_k$, its time and memory complexity exponents $\mathsf{T}_\mathsf{q}\left(\mathcal{T}_k\right)$ and $\mathsf{M}\left(\mathcal{T}_k\right)$ 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 $\mathsf{T}_\mathsf{q}\left(\mathcal{T}_k\right)$, under Constraints 1, 2, 3 and 4. The authors of [25] remarked that this was a linear problem, solvable with Mixed Integer Linear Programming (MILP). Given k, we try all possible tree structures (all binary trees with k leaves) and find the optimal one. As an example, the optimization problem for Figure 4 would be:

$$\begin{aligned}
&\text{Minimize: } \max(t_0^0, \ell_1^1 + t_1^1, \ell_1^2 + t_1^2) \\
&\text{Under the constraints: } t_1^1 = \frac{u_1^1}{2}, t_1^2 = \frac{u_1^2}{2}, t_0^2 = \frac{u_0^2}{2}, \\
&t_0^1 = t_0^2 + \frac{1}{2}\max\left(u_0^1 - u_0^2 - \ell_1^2, 0\right), \\
&t_0^0 = t_0^1 + \frac{1}{2}\max\left(1 - u_0^1 - \ell_1^1, 0\right), u_0^2 = u_1^2, u_0^1 = u_1^1
\end{aligned}$$

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.

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 $T_{k'}$.*

In particular, when $k = 2^\kappa$ is a power of 2, T_k is Wagner's balanced binary tree.

Theorem 3 (Quantum k-XOR with QRACM, from [25]). *Let $k \geqslant 2$ be an integer and $\kappa = \lfloor \log_2(k) \rfloor$. The best quantum merging tree finds a k-XOR on n bits in quantum time (and memory) $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ where $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$. To find 2^{cn} k-XORs for $c > 0$, the complexity exponent goes to $\max(\alpha_k(1+2c), c)$. Furthermore, for every k, the optimum is realized by T_k: $\alpha_k = \mathsf{T_q}(T_k)$.*

Theorem 4 (Quantum k-XOR in the circuit model, from [25]). *Let $k > 2, k \neq 3, 5, 7$ be an integer and $\kappa = \lfloor \log_2(k) \rfloor$. The best quantum merging tree finds a k-XOR on n bits in quantum time and classical memory $\mathcal{O}(2^{\beta_k n})$ where:*

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

To find 2^{cn} k-XORs, the complexity exponent goes to $\max(\beta_k(1+c), c)$. Furthermore, for every $k \neq 3, 5, 7$, the optimum is realized by T_k.

In the circuit model, the strategies for 2, 3, 5, 7 reach respectively $\beta_2 = \frac{2}{5}$ ([11]), $\beta_3 = \frac{5}{14}$ ([15]), $\beta_5 = \frac{40}{129}$ and $\beta_7 = \frac{15}{53}$. Incidentally, the latter two improve over [25] thanks to our rewriting of the constraints (see Appendix C). We prove Theorem 3 in Appendix B and Theorem 4 in Appendix C. These proofs are more constructive, in the sense that the optimal parameters for the whole trees are easier to derive.

5 Extending the Problem

The algorithms of Section 4 solve the Many-solutions case (Problem 1). Following again the study in [25], we extend the merging trees to target the Single-solution case (Problem 2). In this section, we assume QRAQM.

5.1 Repeating the Trees

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. This is the core idea of Schroeppel and Shamir's algorithm [29] and more generally, the *Dissection* algorithms of [13, Section 3]. In the quantum setting, it encompasses some proposed algorithms such as the element distinctness algorithm of [9] (which corresponds to a Single-solution 2-XOR).

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 will allow to *decrease* the time complexity with respect to the quantum birthday bound $\mathcal{O}(2^{n/3})$, as shown in [5, 25].

Algorithm 1.2 Schroeppel and Shamir's algorithm, based on a repetition loop.

```
    Input: oracle access to $h : \{0,1\}^{n/4} \rightarrow \{0,1\}^n$
    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 $\mathcal{L}_i, 0 \leqslant i \leqslant 3$, 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
```

5.2 Extended Merging Trees

The *extended merging trees* that we use in this paper subsume those given in [25], with a technical trick that will allow smaller complexities. It turns out that, although we can make a very generic definition of these trees and the algorithms that they represent, the optimal strategies are very simple. Thus, we defer the complete definition in Appendix D.1, alongside the proof that our strategies are optimal in this definition.

As in Schroeppel and Shamir's algorithm, the merging tree is now extended with *repetition loops*. We make the selection of some arbitrary prefixes, or more generally, sublists of list nodes; that is, of a subtree of the merging tree. We complete the merging process. If a solution is obtained, then the choice of prefixes (resp. sublists) was good. These repetition loops are performed with multiple levels of quantum search (but in practice, one).

Remark 3 (QRAQM requirement). In Section 4, we only needed QRACM, as all intermediate lists could be written down classically, and quantum access was necessary to sample new elements. However, here, the merging operations (writing down the lists) are performed *under a quantum search*, which is why QRAQM is necessary.

Remark 4 (Amending the constraints). Our improved complexities with respect to [25] rely on the following idea. A subtree $\mathcal{T}^j$ of width k^j can cost 0 inside the repetitions if a global cost $\frac{k^j}{k}$ (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, with their sum, in order to retrieve quickly those having a wanted prefix.

Correspondence with Dissection algorithms. The idea of guessing intermediate values is the core of the Dissection framework of [13]. This is exactly what we do here, since under the exhaustive search loops, we merge using arbitrary prefixes: these intermediate values are the prefixes of the subtrees $\mathcal{T}^j$.

Proposition 1. *Any Dissection algorithm for k-XOR (by their definition in [13, Section 3]) can be reframed as a classical extended merging tree.*

5.3 New Results for Single-solution k-XOR

Remark 4 allows us to reach better exponents than [25], and to break the previous lower bound of 0.3 for k-list merging.

Theorem 5 (New trees for unique 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:*

$$\gamma_k = \frac{k + \left\lfloor \frac{k+6}{7} \right\rfloor + \left\lfloor \frac{k+1}{7} \right\rfloor - \left\lfloor \frac{k}{7} \right\rfloor}{4k} \quad . \tag{5}$$

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

The proof of this optimality is given in Appendix D. The formula of Theorem 5 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 $k \leqslant 5$, the results of Theorem 5 and [25] coincide and we can refer to [25]. The novelty of Theorem 5 appears with Algorithm 1.3 (Figure 5), whose total time complexity is:

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

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 $\mathcal{L}_{67}$.

Algorithm 1.3 New Single-solution 7-XOR algorithm.

```
    Input: 7 lists $\mathcal{L}_i$
    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 $s \in \{0,1\}^{2n/7}$ such that          ▷ $2^{n/7}$ 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 $x \in \mathcal{L}_1 \times \mathcal{L}_2$ such that      ▷ $2^{n/7}$ 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:         if $x \oplus y \oplus z = 0$ then
 9:             Exit and return the result
10:     EndSample
11: EndSample
```

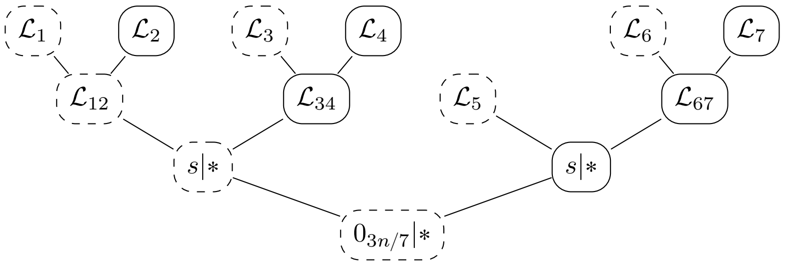


Fig. 5. Single-solution 7-XOR merging tree of Algorithm 1.3.

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:

$$\begin{cases} k_1 = \left\lfloor \frac{3k}{7} \right\rceil \\ k_2 = \left\lfloor \frac{2k}{7} \right\rceil - \left\lfloor \frac{k-1}{7} \right\rfloor + \left\lfloor \frac{k-2}{7} \right\rfloor \ , \end{cases} , \tag{6}$$

where $\left\lfloor \frac{3k}{7} \right\rceil$ is the integer closest to $\frac{3k}{7}$, and we perform Algorithm 1.4. The tree structure (Figure 6) 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. The fact that this choice of structure matches the complexity given by Theorem 5 is not obvious, but it will follow naturally from the proof in Appendix D.

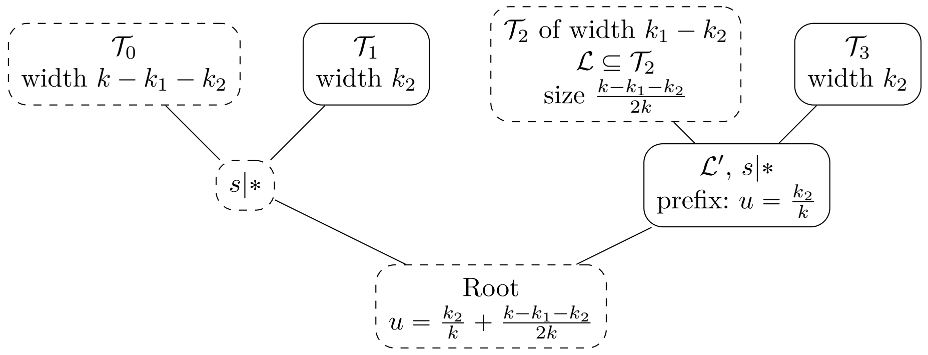

$\mathcal{T}_0$
width $k - k_1 - k_2$
$\mathcal{T}_1$
width k_2
$\mathcal{T}_2$ of width $k_1 - k_2$
$\mathcal{L} \subseteq \mathcal{T}_2$
size $\frac{k-k_1-k_2}{2k}$
$\mathcal{T}_3$
width k_2
$s|*$
$\mathcal{L}'$, $s|*$
prefix: $u = \frac{k_2}{k}$
Root
$u = \frac{k_2}{k} + \frac{k-k_1-k_2}{2k}$

Fig. 6. Generic merging tree that reaches the optimal complexity for Single-solution k-XOR (see Algorithm 1.4).

Algorithm 1.4 Generic Single-solution k-XOR algorithm.

```
    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 6
 2: Build $\mathcal{T}_1$ and $\mathcal{T}_3$, each with the product of $k_2$ lists
 3: Sample $s \in \{0,1\}^{\frac{k_2}{k}n}$ such that
 4:     Sample Sublists $\mathcal{L}$ of $\mathcal{T}_2$ of size $\frac{k-k_1-k_2}{2k}$ such that
 5:         Merge $\mathcal{L}$ with $\mathcal{T}_3$ to obtain an intermediate list $\mathcal{L}'$ with prefix of $\frac{k_2}{k}n$ bits and size $\frac{k-k_1-k_2}{2k}$
 6:         Sample $x \in \mathcal{T}_0$ such that        ▷ $2^{\frac{k-k_1-k_2}{2k}n}$ quantum search iterates
 7:             Find $y \in \mathcal{T}_1$ such that $x \oplus y = s|*$
 8:             if There is a collision on $\mathcal{L}'$ then
 9:                 Exit and return the result
10:         EndSample
11:     EndSample
12: EndSample
```

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 obtain the same results as [25]. However, we could compute trees for higher values of k. We find that the best time-memory product decreases for small k, reaches a minimum, and then increases, as the exponent behaves like $(k - \mathcal{O}(\sqrt{k}))/2$.

Proposition 2. *The time-memory product for Single-solution k-XOR merging trees is lower bounded by: $\widetilde{\mathcal{O}}\left(2^{\frac{7}{17}n}\right)$ which is reached by $k = 17$.*

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 get completely rid of QRAQM, and use only classical memory with quantum access. 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 powers 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 $\mathcal{O}\big(2^{3n/7}\big)$ using $\mathcal{O}\big(2^{n/7}\big)$ qubits [18]. 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 Merge 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.

6 Extension with Quantum Walks

The algorithms presented so far are the best ones achievable, but in the restricted model of quantum merging trees. One of the open questions left in [25] 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 5 that we will now explicit. Our strategy is very simple. Instead of going back to the roots of the framework, we start from our generic result (Algorithm 1.4) and modify it as to allow quantum walks. In particular, we obtain the first 4-SUM algorithm with complexity below $\mathcal{O}\big(2^{0.3n}\big)$ (obtained in [5] with a quantum walk).

Theorem 6 (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:*

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

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

6.1 Preliminaries

In this paper, we only use quantum walks that solve the following problem.

Problem 3 (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 $\mathcal{O}\big(2^{2\ell n/3}\big)$ when $\ell_1 = \ell_2 = \ell$. We will give some high-level ideas and refer to [1], but also to [5, 19] 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 choice 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, depending on r:

$$2^{rn} + \frac{2^{2\ell n}}{2^{2rn}} \left(2^{rn}\right) \quad ,$$

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 [23], achieves:

$$2^{rn} + \sqrt{\frac{2^{2\ell n}}{2^{2rn}}} \left(\sqrt{2^{rn}}\right) \quad ,$$

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 [19]. 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 $r_1 \leqslant \ell_1$ and $r_2 \leqslant \ell_2$. 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 7. *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:*

$$\tau = \max\left(r, \frac{\ell_1 + \ell_2 - r}{2}\right) \quad ,$$

for any r such that $r \leqslant \ell_1, r \leqslant \ell_2, r \geqslant 0$.

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 shall remain 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 7 as a building block.

We will reuse the tree structure of Figure 6. We name $\mathcal{L}_0$ to $\mathcal{L}_3$ the nodes of level 2 (actually products of base lists). Thus, we reuse most of the structure of Algorithm 1.4, except that the parameters will be re-optimized and that the two innermost **Sample** loops are replaced by a single call to **Claw-finding**. In fact, this change is the same that occurs between Buhrman et al.'s element distinctness [9] and the improvement by Ambainis [1]. This is why we reach an improved time complexity. The new choice of k_1, k_2 is the following:

$$\begin{cases} k_1 = \begin{cases} \left\lfloor \frac{k+1}{7} \right\rfloor + \left\lfloor \frac{k+4}{7} \right\rfloor + \left\lfloor \frac{k+6}{7} \right\rfloor \text{ for } k \geqslant 4 \\ 1, 1, 2 \text{ for } k = 2, 3, 4 \text{ respectively} \end{cases} \\ k_2 = \left\lfloor \frac{k}{7} \right\rfloor + \left\lfloor \frac{k+4}{7} \right\rfloor \end{cases} . \tag{8}$$

The key idea of Algorithm 1.5 is that the knowledge of $\mathcal{L}_1$ and $\mathcal{L}_3$, and the constraints of merging, make sure that we can run the quantum walk as expected (that is, we can query as efficiently the lists at level 1 as the lists at level 2).

Algorithm 1.5 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 (8)
2: Build $\mathcal{T}_1$ and $\mathcal{T}_3$, each with the product of $k_2$ lists
3: Sample $s \in \{0,1\}^{\frac{k_2}{k} n}$ such that
4:     Apply Claw-finding between the lists $\mathcal{L}_0 \bowtie_s \mathcal{L}_1$ and $\mathcal{L}_2 \bowtie_s \mathcal{L}_3$
5:     if A claw is found then
6:         Exit and return the result
7: EndSample
```

By definition of k_1 and k_2, the base lists $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$ 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 7, we compute the following time complexity for Algorithm 1.5:

$$2^{\frac{k_2}{2k}n}\left(2^{rn} + 2^{\left(\frac{k-k_1-k_2}{2k}+\frac{k_1-k_2}{2k}-r\right)n} \times 2^{rn/2}\right) + 2^{\frac{k_2}{k}n} \enspace , \tag{9}$$

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 1.5 is solution to the following optimization problem:

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

Here constraints (C1) and (C3) correspond to the walks, (C2) corresponds 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 (8), and the time complexity exponent of Theorem 7.

6.3 Results

In Figure 7, we compare Algorithm 1.5 with the previous work of [25] (where the formula was $\gamma_k = \frac{1}{k}\frac{k+\lceil k/5 \rceil}{4}$). We remark that our curve now includes Ambainis' algorithm for $k=2$ as a special case, which was not the case before, and we actually improve over the complexity $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$ for 4-SUM obtained in [5]. Precise numbers are given in Table 1.

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: $\sqrt{2^{\frac{n}{4}}} \times 2^{\frac{n}{4}\times\frac{2}{3}} = 2^{7n/24}$.

7 Applications

We now show that, similarly as those of [25], the algorithms of Section 6 apply to the class of *bicomposite* problems studied by Dinur et al. [13].

7.1 Bicomposite Problems

The classical Dissection algorithms of [13] are not formulated as Single-solution k-XOR algorithms, although they can be used in this context. They solve a more general problem that we will now define. In this definition, the notation r assumes the same role as k.

Table 1. Quantum time and memory complexity exponents for Single-solution k-XOR obtained with Algorithm 1.5. 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
25	0.2867	43/150	0.28	7/25	0.1467	11/75	11	7
26	0.2885	15/52	0.2692	7/26	0.1538	2/13	11	7
27	0.2901	47/162	0.2593	7/27	0.1605	13/81	12	7
28	0.2857	2/7	0.2857	2/7	0.1429	1/7	12	8
29	0.2874	25/87	0.2759	8/29	0.1494	13/87	13	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

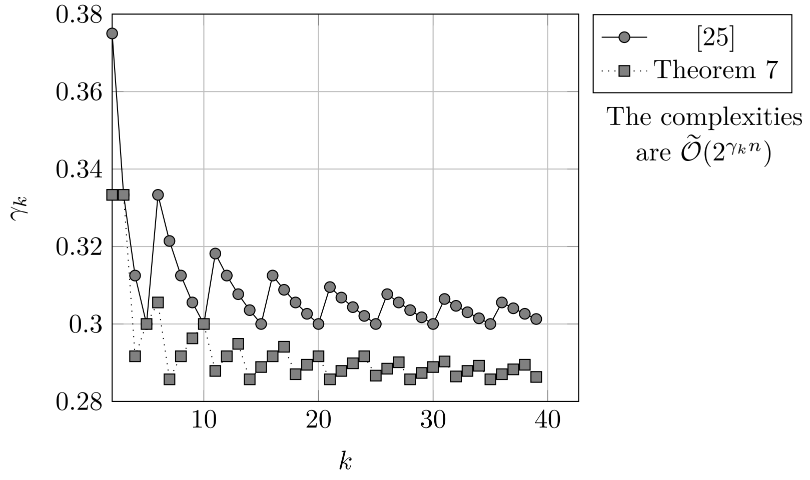


Fig. 7. Single-solution k-XOR time complexity and comparison with [25].

Problem 4 (r-composite search). Let $(X_i, 1 \leqslant i \leqslant r+1)$ be a family of $r+1$ finite sets of same cardinality (say, 2^n for some parameter n). Let $\mathcal{F}_1, \ldots, \mathcal{F}_r$ be r families of functions: $f_i \in \mathcal{F}_i \ : X_i \rightarrow X_{i+1}$, e.g., permutations or random functions, with the same cardinality 2^n.

Let $(x_1^1, \ldots, x_r^1) \in X_1$ and $(x_1^{r+1}, \ldots, x_r^{r+1}) \in X_{r+1}$. Find $f_1 \in \mathcal{F}_1, \ldots, f_r \in \mathcal{F}_r$ such that:

$$\forall i, (f_r \circ \ldots \circ f_1)(x_i^1) = x_i^{r+1} \ .$$

In other words, we are given lists of transitions (the families $\mathcal{F}_i$) that act on r inputs (the x_i^1), and we want to find a sequence of such transitions that brings these inputs to r given targets. The *bicomposite* nature of this problem comes from the orthogonality between the choices of the transitions and the targets. For example, assume that $x_1^{i+1}, \ldots, x_i^{i+1}$ are given, then we can find immediately the sequence of i transitions $f_i \circ \ldots \circ f_1$ that lead to these values: we do not need to guess the full tuple $x_1^{i+1}, \ldots, x_r^{i+1}$.

A prominent example of a bicomposite problem is the case where all the f_i are permutations: this is the *multiple-encryption* problem.

Problem 5 (r-encryption). Let $E^1, \ldots, E^r$ be r 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 $k_1, \ldots, k_r$:

$$\forall i, c_i = \left(E_{k_r}^r \circ \ldots \circ E_{k_1}^1\right)(p_i),$$

then retrieve $k_1, \ldots, k_r$.

The number of given plaintext-ciphertext pairs is enough to discriminate the good sequence of keys with constant probability, as each key is of n bits and each plaintext is of n bits.

7.2 Relation with k-XOR

The Single-solution k-XOR and k-SUM problems are naturally k-composite. The intermediate states x^i are n-bit strings, that we can divide into a product of k strings of n/k bits. A k-composite algorithm makes some guesses for partial intermediate states, which correspond to merging the lists $\mathcal{L}_i$ with arbitrary prefixes.

This relation goes the other way. Instead of trying to prove a full-fledged correspondence between Single-solution r-XOR algorithms of our framework and r-composite problems, we can focus on the algorithms presented in Section 6. For simplicity, we consider a single family of permutations $\mathcal{F}$, of size 2^n, although this works as well with r families and for one-way functions (since we can tabulate them individually). We write down Algorithm 1.6. Its time complexity is computed with the same formula as the k-XOR variant.

Theorem 8. *For any $r \geqslant 2$, let γ_r be the optimal time complexity exponent for unique r-XOR with merging trees, given by Theorem 6. Then there exists a quantum algorithm for r-composite search, with domain size 2^n, of time complexity $\mathcal{O}(2^{\gamma_r r n})$.*

8 Conclusion

In this paper, we have improved the known quantum algorithms for the k-XOR and k-SUM problems, also leading to the best quantum algorithms for *bicomposite search*, and in particular, multiple-encryption.

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 $\frac{2}{7}$. In particular, any problem that can be reduced to k-SUM, *for any k*, does not see any improvement from using walks, for now.

Of course, this might be because the involvement of quantum walks has remained rather superficial, since most of our analysis has been done with quantum search only. 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.

There are also more specific questions that remain open. For example, can we obtain a better time complexity than $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$ for Single-solution 3-XOR? Can we obtain a better time complexity than $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$ for Single-solution k-XOR in the QRACM model?

Acknowledgements. The author would like to thank André Chailloux and María Naya-Plasencia for many discussions and comments on intermediate versions of this work, and Pierre Briaud and Johanna Loyer 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).

Algorithm 1.6 Quantum r-composite search.

```
    Input: two $r$-tuples $x^1 = (x_1^1, \dots, x_r^1)$ and $x^{r+1} = (x_1^{r+1}, \dots, x_r^{r+1})$, a family of permutations $\mathcal{F}$ indexed by a "key" $k \in K$, with quantum oracle access to:
        $$x, k \mapsto f_k(x)$$
    Output: a sequence of "keys" $k_1, \dots, k_r$ such that $x^1$ is mapped to $x^{r+1}$ by $f_{k_r} \circ \dots \circ f_{k_1}$
 1: Select $r_1, r_2$ with Equation 6
 2: Build a list $\mathcal{L}_1$:
        $$\mathcal{L}_1 = \{(f_{k_{r_2}} \circ \dots \circ f_{k_1})(x^1), k_{r_2}, \dots, k_1 \in K\}$$
                              ▻ Thus, the list contains all possible choices for the $r_2$ first steps
 3: Build a list $\mathcal{L}_3$:
        $$\mathcal{L}_3 = \{(f^{-1}_{k_{r-r_2}} \circ \dots \circ f^{-1}_{k_r})(x^{r+1}), k_{r-r_2+1}, \dots, k_r \in K\}$$
                              ▻ Thus, the list contains all possible choices for the $r_2$ last steps
 4: Sample $y_1, \dots, y_{r_2}$ such that
                              ▻ These guesses are from the intermediate state $x_{r-r_1}$
 5:     Define: $\mathcal{L}_0$ the list of all key sequences for steps $r_2 + 1, \dots, r - r_1$
 6:     Define: $\mathcal{L}_2$ the list of all key sequences for steps $r - r_1 + 1, \dots, r - r_2 + 1$

 7:     Define: $\mathcal{L}_{01}$ the list of all key-sequences from $\mathcal{L}_1 \times \mathcal{L}_0$ such that $x_1^1, \dots, x_{r_2}^1$ is mapped to $y_1, \dots, y_{r_2}$
 8:     Define: $\mathcal{L}_{23}$ the list of all key-sequences from $\mathcal{L}_2 \times \mathcal{L}_3$ such that $y_1, \dots, y_{r_2}$ is mapped to $x_1^{r+1}, \dots, x_{r_2}^{r+1}$
 9:     Find a claw between $\mathcal{L}_{01}$ and $\mathcal{L}_{23}$
10: EndSample
```

It remains to explain how we sample from $\mathcal{L}_{01}$ (resp. $\mathcal{L}_{23}$) in time 1, given the knowledge of $\mathcal{L}_1$ (resp. $\mathcal{L}_3$). This is made possible by the fact that both $\mathcal{L}_1$ and $\mathcal{L}_3$ contain all their key-sequences.

- For $\mathcal{L}_{01}$: when we sample an element from $\mathcal{L}_0$, we have a key-sequence for steps $r_2 + 1, \dots, r - r_1$, and we also have an endpoint $y_1, \dots, y_{r_2}$. Thus we invert the steps from $y_1, \dots, y_{r_2}$, we obtain an intermediate $z_1, \dots, z_{r_2}$ and we find in $\mathcal{L}_1$ a key-sequence that sends $x_1^1, \dots, x_{r_2}^1$ to this intermediate.
- For $\mathcal{L}_{23}$: when we sample an element from $\mathcal{L}_2$, we have a key-sequence for steps $r - r_1 + 1, \dots, r - r_2 + 1$, and we also have a starting point $y_1, \dots, y_{r_2}$. Thus we obtain an intermediate $z_1, \dots, z_{r_2}$ and we find in $\mathcal{L}_3$ the key-sequence that sends this intermediate to $x_1^{r+1}, \dots, x_{r_2}^{r+1}$.

References

[1] Andris Ambainis. “Quantum Walk Algorithm for Element Distinctness”. In: *SIAM J. Comput.* 37.1 (2007), pp. 210–239.

[2] Shi Bai et al. “Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem”. In: *J. Cryptology* 32.1 (2019), pp. 35–83.

[3] Anja Becker, Jean-Sébastien Coron, and Antoine Joux. “Improved Generic Algorithms for Hard Knapsacks”. In: *EUROCRYPT*. Vol. 6632. LNCS. Springer, 2011, pp. 364–385.

[4] Aleksandrs Belovs and Robert Spalek. “Adversary lower bound for the k-sum problem”. In: *ITCS*. ACM, 2013, pp. 323–328.

[5] Daniel J. Bernstein et al. “Quantum Algorithms for the Subset-Sum Problem”. In: *PQCrypto*. LNCS Vol. 7932. Springer, 2013, pp. 16–33.

[6] Xavier Bonnetain et al. “Improved Classical and Quantum Algorithms for Subset-Sum”. In: *ASIACRYPT*. LNCS. Springer, 2020. URL: `https://eprint.iacr.org/2020/168`.

[7] Gilles Brassard, Peter Høyer, and Alain Tapp. “Quantum Cryptanalysis of Hash and Claw-Free Functions”. In: *LATIN*. Vol. 1380. LNCS. Springer, 1998, pp. 163–169.

[8] Gilles Brassard et al. “Quantum amplitude amplification and estimation”. In: *Contemporary Mathematics* 305 (2002), pp. 53–74.

[9] Harry Buhrman et al. “Quantum Algorithms for Element Distinctness”. In: *SIAM J. Comput.* 34.6 (2005), pp. 1324–1330.

[10] Paul Camion and Jacques Patarin. “The Knapsack Hash Function proposed at Crypto’89 can be broken”. In: *EUROCRYPT*. Vol. 547. LNCS. Springer, 1991, pp. 39–53.

[11] André Chailloux, María Naya-Plasencia, and André Schrottenloher. “An Efficient Quantum Collision Search Algorithm and Implications on Symmetric Cryptography”. In: *ASIACRYPT (2)*. Vol. 10625. LNCS. Springer, 2017, pp. 211–240.

[12] Itai Dinur. “An algorithmic framework for the generalized birthday problem”. In: *Des. Codes Cryptogr.* 87.8 (2019), pp. 1897–1926.

[13] Itai Dinur et al. “Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems”. In: *CRYPTO*. Vol. 7417. LNCS. Springer, 2012, pp. 719–740.

[14] Philippe Flajolet and Andrew M. Odlyzko. “Random Mapping Statistics”. In: *EUROCRYPT*. Vol. 434. LNCS. Springer, 1989, pp. 329–354.

[15] Lorenzo Grassi, María Naya-Plasencia, and André Schrottenloher. “Quantum Algorithms for the k-xor Problem”. In: *ASIACRYPT 2018*. Vol. 11272. LNCS. Springer, 2018, pp. 527–559.

[16] Lov K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search”. In: *STOC*. ACM, 1996, pp. 212–219.

[17] Nick Howgrave-Graham and Antoine Joux. “New Generic Algorithms for Hard Knapsacks”. In: *EUROCRYPT*. Vol. 6110. LNCS. Springer, 2010, pp. 235–256.

[18] Samuel Jaques and André Schrottenloher. "Low-gate Quantum Golden Collision Finding". In: *SAC*. LNCS. Springer, 2020. URL: `https://eprint.iacr.org/2020/424`.

[19] Ghazal Kachigar and Jean-Pierre Tillich. "Quantum Information Set Decoding Algorithms". In: *PQCrypto*. Vol. 10346. LNCS. Springer, 2017, pp. 69–89.

[20] Elena Kirshanova et al. "Quantum Algorithms for the Approximate k-List Problem and Their Application to Lattice Sieving". In: *ASIACRYPT (1)*. Vol. 11921. Lecture Notes in Computer Science. Springer, 2019, pp. 521–551.

[21] Greg Kuperberg. "Another Subexponential-time Quantum Algorithm for the Dihedral Hidden Subgroup Problem". In: *TQC*. Vol. 22. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 20–34.

[22] Vadim Lyubashevsky. "The Parity Problem in the Presence of Noise, Decoding Random Linear Codes, and the Subset Sum Problem". In: *APPROX-RANDOM*. Vol. 3624. LNCS. Springer, 2005, pp. 378–389.

[23] Frédéric Magniez et al. "Search via Quantum Walk". In: *SIAM J. Comput.* 40.1 (2011), pp. 142–164.

[24] Lorenz Minder and Alistair Sinclair. "The Extended k-tree Algorithm". In: *J. Cryptology* 25.2 (2012), pp. 349–382.

[25] María Naya-Plasencia and André Schrottenloher. "Optimal Merging in Quantum k-XOR and k-SUM Algorithms". In: *EUROCRYPT (2)*. LNCS Vol. 12106. Springer, 2020, pp. 311–340.

[26] Michael A. Nielsen and Isaac L. Chuang. "Quantum information and quantum computation". In: *Cambridge: Cambridge University Press* 2.8 (2000), p. 23.

[27] Ivica Nikolic and Yu Sasaki. "Refinements of the k-tree Algorithm for the Generalized Birthday Problem". In: *ASIACRYPT (2)*. Vol. 9453. LNCS. Springer, 2015, pp. 683–703.

[28] Alessandro Panconesi and Aravind Srinivasan. "Randomized Distributed Edge Coloring via an Extension of the Chernoff-Hoeffding Bounds". In: *SIAM J. Comput.* 26.2 (1997), pp. 350–368.

[29] Richard Schroeppel and Adi Shamir. "A $T = \mathcal{O}(2^{n/2}), S = \mathcal{O}(2^{n/4})$ Algorithm for Certain NP-Complete Problems". In: *SIAM J. Comput.* 10.3 (1981), pp. 456–464.

[30] David A. Wagner. "A Generalized Birthday Problem". In: *CRYPTO*. Vol. 2442. LNCS. Springer, 2002, pp. 288–303.

[31] Mark Zhandry. "How to Record Quantum Queries, and Applications to Quantum Indifferentiability". In: *CRYPTO (2)*. Vol. 11693. LNCS. Springer, 2019, pp. 239–268.

A 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 $\mathcal{L}_u$, and start with the case where $\mathcal{L}_u$ is smaller.

Lemma 3. *If $\ell_u \leqslant \max(\ell_1, \ell_2)$, 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 $\ell_u \leqslant \max(\ell_1, \ell_2)$, we will use an argument similar to [22]. We first need a technical result, which is a property of random mappings.

Lemma 4. *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:*

$$\Pr(Y(h) > 0.4 \cdot 2^n) \leqslant 0.9987^{2^n} \ . \tag{10}$$

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 [28], and for any $\delta > 0$:

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

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 $\ell_2 \leqslant \ell_1$. The idea is that since $\ell_u \leqslant \ell_1$, 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:

$$\forall x, \forall \delta \leqslant 1, \Pr(|L_2^x - \mathbb{E}\left(L_2^x\right)| \geqslant \delta\, \mathbb{E}\left(L_2^x\right)) \leqslant 2e^{-\frac{\delta^2\, \mathbb{E}\left(L_2^x\right)}{3}}, \tag{11}$$

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

$$\Pr\Big(\exists x, |L_2^x - \mathbb{E}(L_2^x)| \geqslant \mathbb{E}(L_2^x)^{2/3}\Big) \leqslant 2^{un+1} e^{-\frac{\mathbb{E}\left(L_2^x\right)^{1/3}}{3}} \quad . \tag{12}$$

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 4. 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 (12), 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 3 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 [30] 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 [24]. 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 5. *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 4, $\mathcal{L}_1$ 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 4 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. □

Lemma 5 contains all what 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 1.4):

- 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 5, 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 2) 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.

B Proof of Optimality in the QRACM Setting

We prove Theorem 3:

> 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 $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$.

Proof. We use a recurrence on k. 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 [7]. This works as long as $c \leqslant 1$, i.e., $2c + 1 \leqslant 3$. 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 $1 - u - \ell_r \geqslant 0$, 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 $\max\left(\alpha_{k_r}(1 + 2\frac{\ell_r}{u}), \frac{\ell_r}{u}\right) u = \max(\alpha_{k_r}(u + 2\ell_r), \ell_r)$.

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:

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

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:

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

Then this inequality becomes:

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

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:

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

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 $\lfloor \log_2 \lfloor k/2 \rfloor \rfloor = \lfloor \log_2 (k - \lfloor k/2 \rfloor) \rfloor = \kappa - 1$. Using the recurrence hypothesis, we obtain that:

$$\begin{aligned} 1 + \frac{1}{2\alpha_{k_l}} + \frac{1}{2\alpha_{k_r}} &= 1 + \frac{(1+\kappa-1)2^{\kappa-1} + k - \left\lfloor \frac{k}{2} \right\rfloor}{2^\kappa} + \frac{(1+\kappa-1)2^{\kappa-1} + \left\lfloor \frac{k}{2} \right\rfloor}{2^\kappa} \\ &= \frac{2^\kappa(1+\kappa)+k}{2^\kappa} \quad . \end{aligned}$$

Thus, we can write: $t \geqslant (2c+1)\frac{2^\kappa}{2^\kappa(1+\kappa)+k}$, which gives the expected formula for α_k. The second inequality $t \geqslant c$ 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 6. *Optimizing the balanced trees T_k yields the optimal exponents.*

First, we focus on the case $c \leqslant \frac{\alpha_k}{1-2\alpha_k}$, 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

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

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:

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

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

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

C Proof of Optimality in the Circuit Model

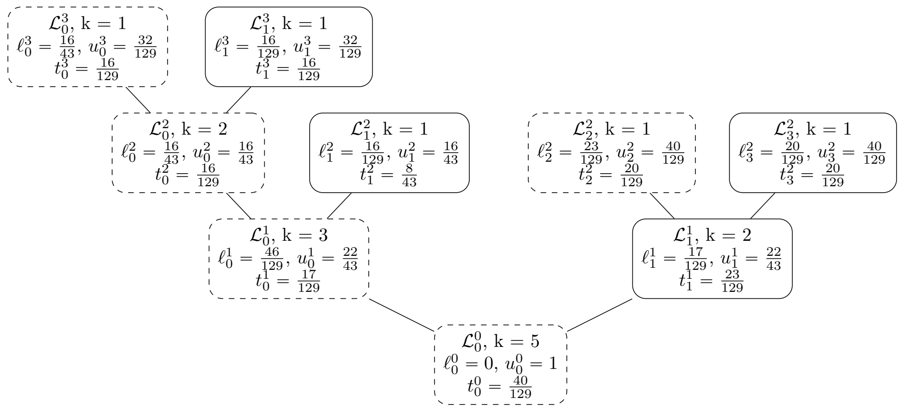

$\mathcal{L}_0^3$, k = 1
$\ell_0^3 = \frac{16}{43}$, $u_0^3 = \frac{32}{129}$
$t_0^3 = \frac{16}{129}$
$\mathcal{L}_1^3$, k = 1
$\ell_1^3 = \frac{16}{129}$, $u_1^3 = \frac{32}{129}$
$t_1^3 = \frac{16}{129}$
$\mathcal{L}_0^2$, k = 2
$\ell_0^2 = \frac{16}{43}$, $u_0^2 = \frac{16}{43}$
$t_0^2 = \frac{16}{129}$
$\mathcal{L}_1^2$, 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. 8. Optimal 5-XOR merging tree in the circuit model.

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 [25]. We obtain $\beta_5 = \frac{40}{129}$ and $\beta_7 = \frac{15}{53}$ instead of $\frac{14}{45}$ and $\frac{2}{7}$ respectively. The details are given in Figure 8 and Figure 9. For other values of k, our results coincide with [25] and we prove Theorem 4:

> For any integer $k \geqslant 8$ 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:
>
> $$\beta_k = \begin{cases} \frac{1}{\kappa+1} & \text{if } k < 2^\kappa + 2^{\kappa-1} \\ \frac{2}{2\kappa+3} & \text{if } k \geqslant 2^\kappa + 2^{\kappa-1} \end{cases}$$
>
> And when $k \geqslant 8$, this procedure samples *classically.*

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 $k \geqslant 8$, with a list size $c > 0$ at the root. We use the same notations as in Section B, and introduce $k_l, k_r, \beta_{k_l}, \beta_{k_r}$ and the variables u, ℓ_r.

For $k \leqslant 7$, we notice that the exponent is always *at least* $\max(\beta_k(1+c), c)$ (it will lie somewhere between $\beta_k(1+c)$ and $\beta_k(1+2c)$). Having $+c$ instead

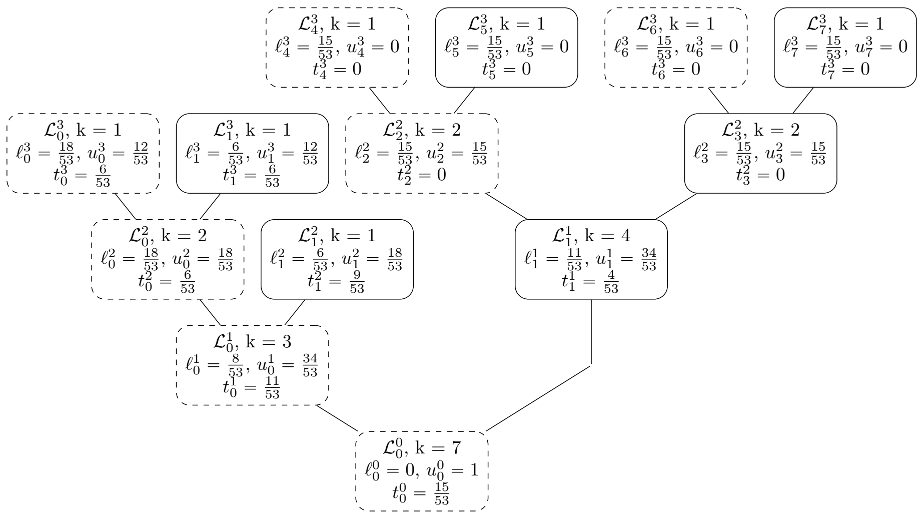

$\mathcal{L}_4^3$, k = 1 $\ell_4^3 = \frac{15}{53}$, $u_4^3 = 0$ $t_4^3 = 0$
$\mathcal{L}_5^3$, k = 1 $\ell_5^3 = \frac{15}{53}$, $u_5^3 = 0$ $t_5^3 = 0$
$\mathcal{L}_6^3$, k = 1 $\ell_6^3 = \frac{15}{53}$, $u_6^3 = 0$ $t_6^3 = 0$
$\mathcal{L}_7^3$, k = 1 $\ell_7^3 = \frac{15}{53}$, $u_7^3 = 0$ $t_7^3 = 0$
$\mathcal{L}_0^3$, 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. 9. Optimal 7-XOR merging tree in the circuit model.

of $+2c$ comes from the use of a classical merging at the root. Once we know that the merge is classical, we can deduce easily that both subtrees must be of similar shapes, hence $k_\ell = \left\lfloor \frac{k}{2} \right\rfloor$ and $k_r = \left\lceil \frac{k}{2} \right\rceil$ 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 $\left\lfloor \frac{k}{2} \right\rfloor < 2^{\kappa-1} + 2^{\kappa-2}$; and conversely, if $k \geqslant 2^\kappa + 2^{\kappa-1}$, then $\left\lfloor \frac{k}{2} \right\rfloor \geqslant 2^{\kappa-1} + 2^{\kappa-2}$.

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 $\max(c + \frac{1}{2}(1 - u - \ell_r), \beta_{k_l}\left(u + c + \frac{1}{2}(1 - u - \ell_r)\right))$. We also do the same number of QRACM emulations, in time ℓ_r each. With the right child, we have 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) $t \geqslant \beta_{k_r}(u + \ell_r)$ (C2) $t \geqslant \frac{\beta_{k_l}}{2}(2c + 1 + u - \ell_r)$
(C3) $t \geqslant \frac{1}{2}(2c + 1 - u + \ell_r)$

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

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

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

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 in the circuit model should cost at least as much as in the QRACM model, so we introduce:

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

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

Since we have $k_l \geqslant \lfloor k/2 \rfloor$, at the same time, we should have $k_r \leqslant \lceil k/2 \rceil$ so $k_r \leqslant k_l + 1$ and $\alpha_{k_r} \geqslant \alpha_{k_l+1}$. This inequality becomes $t \geqslant 2\alpha_{k_l+1}\frac{1-\beta_{k_l}}{\beta_{k_l}+1}(2c+1)$. A quick computation of the first values of α_i and β_i shows that for $k_l \geqslant 15, 2\alpha_{k_l+1}\frac{1-\beta_{k_l}}{\beta_{k_l}+1} \geqslant \beta_{k_l+1}$. In other words, 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.

Again, we can verify that the balanced trees T_k give the optimal results for $k \geqslant 8$. □

D Proof of Optimality for Single-solution k-XOR

We now prove Theorem 5. 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 9. *For any k, the optimal time t for the Single-solution k-XOR problem, with our merging tree framework, is given by:*

$$t = \min_{k_1,k_2 \in \mathbb{N}^2, k_1+k_2 \leqslant k}\left(\max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right), \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right)\right) \qquad (13)$$

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

In Section D.1, we give the definition of extended merging trees. In Section D.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 Section D.3 and finish the proof of the theorem.

D.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. That is, do another quantum search. Constraints 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 the 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 10. Thus, there is only one non-zero repetition variable at each level, which is why we simply number them level by level.

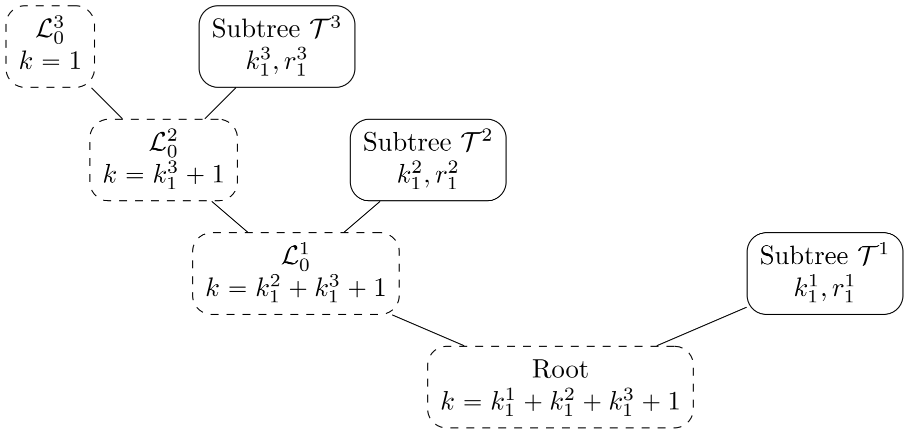


Fig. 10. *Main branch* of a merging tree and all the subtrees that are attached to it.

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.

Constraint 6 (Repetitions)**.** *We have:* $r = \sum_j r^j$*, and for each subtree* $\mathcal{T}^j$ *of width* k^j*:* $r \leqslant \frac{k_j}{k} - \ell^j$ *, 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:*

$$t_i^j = t_{2i}^{j+1} + \frac{1}{2}\max(u_i^j - u_{2i}^{j+1} - \ell_{2i+1}^{j+1}, 0) \ . \tag{14}$$

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:*

$$t^j = \max\left(\max_{\text{List nodes of } \mathcal{T}^j}(t_i^j + \ell_i^j)\right) , \tag{15}$$

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 $\mathcal{T}^1, \ldots, \mathcal{T}^p$ 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:*

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

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

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

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, as in Algorithm 1.7, with the idea that bigger and more costly subtrees may be attached to nodes at lower levels. We will see however that in the optimal algorithm for Single-solution k-XOR, all these levels collapse into a single one.

We can see that, with our definitions of t^j, t^0, r^j and r, the time complexity of Algorithm 1.7, up to a polynomial factor, is going to be:

$$2^{nr_1}\left(\underbrace{2^{nt_1}}_{\mathcal{T}^1} + 2^{nr_2}\left(\underbrace{2^{nt_2}}_{\mathcal{T}^2} + \ldots + 2^{nr_p}\left(\underbrace{2^{nt_p}}_{\mathcal{T}^p} + \underbrace{2^{nt^0}}_{\text{Sample}}\right)\ldots\right)\right) \tag{16}$$

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

Remark 5 (Further generalizations). In full generality, one could add repetition loops to *any* list node in the tree, and put them in any order that remains coherent with the tree. Except for the specific example of Remark 4, it does not seem to bring any improvement at all.

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 10 (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*

Algorithm 1.7 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 $x \in \mathcal{L}_0^p$ such that
 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
```

a quantum extended merging algorithm *that, given access to a quantum oracle O_h, finds a k-XOR.*

This algorithm succeeds with constant probability. It runs in time $\mathcal{O}\left(2^{\mathsf{T_q}(\mathcal{T}_k)n}\right)$, counted in n-qubit register operations, makes the same number of queries to O_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. We rely on Theorem 2 for the correctness of merging strategies. Each level of quantum search builds a new subtree, as in Algorithm 1.7. The search space itself is defined by an arbitrary prefix, either of the codomain (a merging constraint, as in Schroeppel and Shamir's algorithm) 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 (otherwise there would be too many repetitions). We miss it if the corresponding merging tree fails to find it; but Theorem 2 ensures a constant probability of success. □

D.2 Step 1: Reducing the Search Space

When merging, we drop many tuples. 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 7. *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 $u' \leqslant u$ 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 $r_l \geqslant u'$ and $r \geqslant u$, 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 $\mathcal{T}^i$:

$$\text{(Some factor)}\left(2^{\frac{nr_l}{2}}\left(\text{Build } \mathcal{L}_l + 2^{\frac{nr}{2}}\left(\text{Build } \mathcal{T}^i + \text{(Other terms)}\right)\right)\right) \enspace .$$

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 with placeholders for the terms that will remain unchanged:

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

Now, let us simply remove the prefix condition on both $\mathcal{L}_s$ and $\mathcal{L}_l$. We want a list $\mathcal{L}_l$ of same size as before, so in practice, we may reduce the width of its subtree. 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 $r_l - u' \geqslant 0$ repetitions to take into account. The terms t_s and t_l are replaced by $t'_s \leqslant t_s$ and $t'_l \leqslant t_l$. The complexity becomes:

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

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, the subtrees of the main branch are very simple. Inside of them, there are only empty prefixes, except the root. 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. □

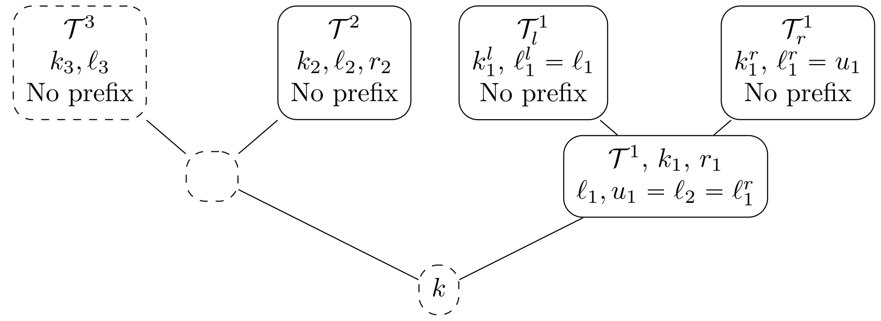


Fig. 11. Extended merging tree with three levels and a single non-empty prefix.

Next, we show that the main branch of this optimal tree has, actually, only three levels. This is represented on Figure 11.

Lemma 8. *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 11 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 7, 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. □

D.3 Step 2: Solving the Constraints

We are now considering the simple tree shape of Figure 11. 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 1.4. 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 rewriting of the constraints. We find that it is better to create them outside, possibly taking sublists of them afterwards if necessary.

There are possibly two repetition loops, with variables r_1 and r_2:

$$r_1 = u_1 + \frac{k_1^l}{k} - \ell_1 + \frac{k_1^r}{k} - \ell_2 = \frac{k_1^l}{k} - \ell_1 + \frac{k_1^r}{k} \quad \text{and} \quad r_2 = \frac{k_2}{k} - \ell_2 \ , \tag{17}$$

and t satisfies the constraints:

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

which gives:

$$\begin{array}{ll} \text{(C1)} & t \geqslant \max\left(\frac{k_2}{k}, \frac{k_1 - k_1^r}{k}\right) \\ \text{(C2)} & \frac{k_1^r}{k} \geqslant \ell_2, \frac{k_1 - k_1^r}{k} \geqslant \ell_1, \frac{k_2}{k} \geqslant \ell_2 \\ \text{(C3)} & t \geqslant \frac{1}{2}\left(\frac{k_1^l}{k} - \ell_1 + \frac{k_1^r}{k}\right) + \ell_1 \implies t \geqslant \frac{1}{2}\left(\frac{k_1}{k} + \ell_1\right) \\ \text{(C4)} & t \geqslant \frac{1}{2}\left(1 - \ell_1 - \ell_2\right) \end{array}$$

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: $4t \geqslant 1 + \frac{k_1 - k_2}{k}$. We also keep $t \geqslant \frac{k_2}{k}$. From (C4) and (C2) we obtain: $2t \geqslant 1 - \ell_1 - \ell_2 \geqslant 1 - \frac{k_1}{k}$.

We now show that these constraints are necessary and sufficient: given a choice of k_1, k_2, we exhibit merging trees that reach the prescribed complexity. These trees are those given in Theorem 5.

Lemma 9. *Let k_1, k_2 be such that $k_1 + k_2 \leqslant k$. Then there exists an extended merging tree algorithm solving Single-solution k-XOR in time (exponent):*

$$t = \max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right), \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right) \ . \tag{18}$$

Proof. First of all, consider the case $k_1 \leqslant k_2$, i.e., the subtree at level 2 is bigger than the subtree at level 1. This implies in particular $t \geqslant \frac{k_2}{k} \geqslant \frac{k_1}{k}$ and $t \geqslant \frac{1}{2}(1 - \frac{k_1}{k})$, thus $t \geqslant \frac{1}{3}$: this is unlikely to be a good parameter choice. In that case, we must find $t \leqslant \max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right)\right)$. This is easily obtained with a trivial tree, that has only two subtrees: one is obtained by the product of k_1 lists (time $\frac{k_1}{k} \leqslant \frac{k_2}{k}$), 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 $k_1 \geqslant k_2$. We notice that:

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

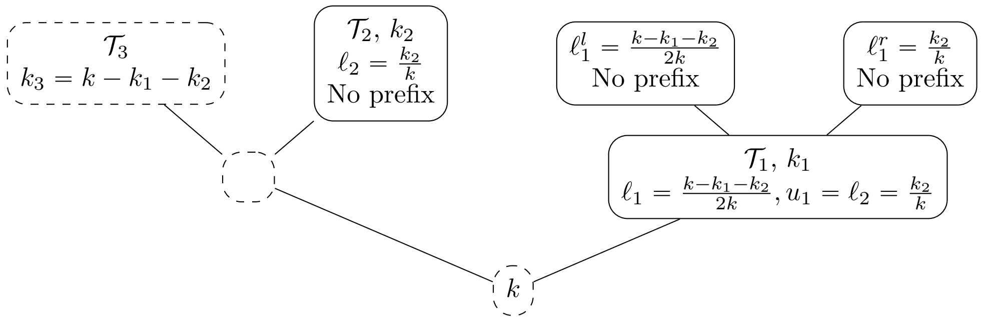


Fig. 12. Tree of Lemma 9.

So we next focus on the case $k \leqslant 3k_1 - k_2$ and we try to obtain a complexity $t \leqslant \max\left(\frac{k_2}{k}, \frac{1}{4}\left(1 + \frac{k_1-k_2}{k}\right)\right)$. The merging tree that we use is drawn on Figure 12. 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 $k \leqslant 3k_1 - k_2 \leqslant 3k_1 + k_2$. 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 $3k_1 + k_2 \leqslant k$, we notice that $3k_1 \leqslant k$ implies $\frac{1}{2}\left(1 - \frac{k_1}{k}\right) \geqslant \frac{1}{3}$, as it happened before for the case $k_1 \leqslant k_2$. Further, $\frac{1}{2}\left(1 - \frac{k_1}{k}\right) \geqslant \frac{k_1}{k}$. 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 (18). $\square$

Finally, we observe that the minimization over k_1, k_2 of this quantity gives the formula of γ_k of Theorem 5, finishing the proof of the theorem:

$$\min_{\substack{k_1,k_2 \in \mathbb{N}^2 \\ k_1+k_2 \leqslant k}} \max\left(\frac{k_2}{k}, \frac{1}{2}\left(1 - \frac{k_1}{k}\right), \frac{1}{4}\left(1 + \frac{k_1 - k_2}{k}\right)\right) = \frac{k + \left\lfloor\frac{k+6}{7}\right\rfloor + \left\lfloor\frac{k+1}{7}\right\rfloor - \left\lfloor\frac{k}{7}\right\rfloor}{4k} \quad . \tag{19}$$

And the minimum can be reached by choosing:

$$\begin{cases} k_1 = \left\lfloor \frac{3k}{7} \right\rceil \\ k_2 = \left\lfloor \frac{2k}{7} \right\rfloor - \left\lfloor \frac{k-1}{7} \right\rfloor + \left\lfloor \frac{k-2}{7} \right\rfloor \end{cases}$$

where $\left\lfloor \frac{3k}{7} \right\rceil$ is the integer that is closest to $\frac{3k}{7}$. We have obtained the tree structure given in Figure 6 and Algorithm 1.4 and proven its optimality among merging trees.