2012 IEEE 53rd Annual Symposium on Foundations of Computer Science

Learning-Graph-Based Quantum Algorithm for k-distinctness

Aleksandrs Belovs
Faculty of Computing, University of Latvia
Riga, Latvia
stiboh@gmail.com

***Abstract*—We present a quantum algorithm solving the k-distinctness problem in a less number of queries than the previous algorithm by Ambainis. The construction uses a modified learning graph approach. Compared to the recent paper by Belovs and Lee, the algorithm doesn't require any prior information on the input, and the complexity analysis is much simpler.**

***Keywords*-quantum computing, query complexity**

I. Introduction

The *element distinctness* problem consists of computing function $f\colon [m]^n \to \{0,1\}$ that evaluates to 1 iff there is a pair of equal elements in the input, i.e., $f(x_1,\ldots,x_n)=1$ iff $\exists i \neq j : x_i = x_j$. (Here we use notation $[n] = \{1,2,\ldots,n\}$.) The quantum query complexity of the element distinctness problem is well understood. It is known to be $\Theta(n^{2/3})$, with the algorithm given by Ambainis [1], and the lower bound shown by Aaronson and Shi [2] and Kutin [3] for the case of large alphabet size $\Omega(n^2)$, and by Ambainis [4] in the general case.

Ambainis' algorithm for the element distinctness problem was the first application of the quantum random walk framework to a "natural" problem (i.e., one seemingly having little relation to random walks), and it had significantly changed the way quantum algorithms have been developed since then. The core of the algorithm is quantum walk on the Johnson graph. This primitive has been reused in many other algorithms: triangle detection in a graph given by its adjacency matrix [5], matrix product verification [6], restricted range associativity [7], and others. Given that the behavior of quantum walk is well-understood for arbitrary graphs [8], [9], it is even surprising that the applications have been mostly limited to the Johnson graph.

The *k-distinctness problem* is a direct generalization of the element distinctness problem. Given the same input, the function evaluates to 1 iff there is a set of k input elements that are all equal, i.e., a set of indices $a_1,\ldots,a_k \in [n]$ with $a_i \neq a_j$ and $x_{a_i} = x_{a_j}$ for all $i \neq j$.

The situation with the quantum query complexity of the k-distinctness problem is not so clear. (In this paper we assume $k = O(1)$, and consider the complexity of k-distinctness as $n \to \infty$.) As element distinctness reduces to k-distinctness by repeating each element $k-1$ times, the lower bound of $\Omega(n^{2/3})$ carries over to the k-distinctness problem (this argument is attributed to Aaronson in Ref. [1]). This simple lower bound is the best known so far.

In the same paper [1] with the element distinctness algorithm, Ambainis applied quantum walk on the Johnson graph in order to solve the k-distinctness problem. This resulted in a quantum algorithm with query complexity $O(n^{k/(k+1)})$. This was the best known algorithm for this problem prior to this paper.

The aforementioned algorithms work by searching for a small subset of input variables such that the value of the function is completely determined by the values within the subset. For instance, the values of two input variables are sufficient to claim the value of the element distinctness function is 1, provided their values are equal. This is formalized by the notion of certificate complexity as follows.

An *assignment* for a function $f\colon \mathcal{D} \to \{0,1\}$ with $\mathcal{D} \subseteq [m]^n$ is a function $\alpha\colon S \to [m]$ with $S \subseteq [n]$. The *size* of α is $|S|$. An input $x = (x_i) \in [m]^n$ *satisfies* assignment α if $\alpha(i) = x_i$ for all $i \in S$. An assignment α is called a *b-certificate* for f, with $b \in \{0,1\}$, if $f(x) = b$ for any $x \in \mathcal{D}$ satisfying α. The *certificate complexity* $C_x(f)$ of f on x is defined as the minimal size of a certificate for f that x satisfies. The b-certificate complexity $C^{(b)}(f)$ is defined as $\max_{x \in f^{-1}(b)} C_x(f)$. Thus, for instance, 1-certificate complexity of element distinctness is 2, and 1-certificate complexity of triangle detection is 3.

Soon after the Ambainis' paper, it was realized [10] that the algorithm developed for k-distinctness can be used to evaluate, in the same number of queries, any function with 1-certificate complexity equal to k. Now we know that for some functions this algorithm is tight, due to the lower bound for the k-sum problem [11]. The goal of the k-sum problem is to detect, given n elements of an Abelian group as input, whether there are k of them that sum up to a prescribed element of the group. The k-sum problem is noticeable in the sense that, given any $(k-1)$-tuple of input elements, one has absolutely no information on whether they form a part of an (inclusion-wise minimal) 1-certificate, or not.

The aforementioned applications of the quantum walk on the Johnson graph (triangle finding, etc.) went beyond $O(n^{k/(k+1)})$ upper bound by utilizing additional relations between the input variables: the adjacency relation of the edges for the triangle problem, row-column relations for

0272-5428/12 $26.00 © 2012 IEEE
DOI 10.1109/FOCS.2012.18

the matrix products, and so on. For instance, two edges in a graph can't be a part of a 1-certificate for the triangle problem, if they are not adjacent.

The k-distinctness problem is different in the sense that it doesn't possess any structure of the variables. But it does possess a relation between the *values* of the variables: two elements can't be a part of a 1-certificate if their values are different. However, it seems that quantum walk on the Johnson graph fails to utilize this structure efficiently.

In this paper, we use the learning graph approach to construct a quantum algorithm that solves the k-distinctness problem in $O\left(n^{1-2^{k-2}/(2^k-1)}\right)$ queries. The learning graph is a novel way of construction quantum query algorithms. Somehow, it may be thought as a way of designing a more flexible quantum walk than just on the Johnson graph. And compared to the quantum walk design paradigms from Ref. [8], [9], it is easier to deal with. In particular, it doesn't require any spectral analysis of the underlying graph.

Up to date, the applications of learning graphs are as follows. Belovs [12] introduced the framework and used it to improve the query complexity of triangle detection. Zhu [13] and Lee, Magniez and Santha [14] extended this algorithm to the containment of arbitrary subgraphs. Belovs and Lee [15] developed an algorithm for the k-distinctness problem that beats the $O(n^{k/(k+1)})$-query algorithm given some prior information about the input. Belovs and Reichardt [16] use a construction resembling learning graph to obtain an optimal algorithm for finding paths and claws of arbitrary length in the input graph. Also, they deal with time-efficient implementation of learning graphs.

The paper is organized as follows. In Section II we define the (dual of the) adversary bound. It is the main technical tool underlying our algorithm. Also, we describe learning graphs and the previous algorithm for the k-distinctness problem. In Section III, we describe the intuition behind our algorithm, and describe the changes we have made to the model of the learning graph. In Sections IV and V we describe the k-distinctness algorithm.

II. Preliminaries

In this paper, we are mainly concerned with query complexity of quantum algorithms, i.e., we measure the complexity by the number of queries to the input the algorithm makes in the worst case. For the definition of query complexity and its basic properties, a good reference is [17].

In Section II-A we describe a tight characterization of the query complexity by a relatively simple semi-definite program (SDP): the adversary bound, Eq. (1). This is the main technical tool underlying our algorithm.

Although Eq. (1) is an SDP, and thus can be solved in polynomial time in the size of the program, the latter is exponential in the number of variables, and becomes very hard to solve exactly as its size grows. The *learning graph* [12] is a tool for designing feasible solutions to Eq. (1), whose complexity is easier to analyze. We define it in Sections II-B and II-C. In the first one, we describe the model following Ref. [12], [15]. In the second one, we describe a common way of constructing learning graphs for specific problems, and give an example of a learning graph for the k-distinctness problem corresponding to the Ambainis' algorithm.

A. Dual Adversary Bound

The adversary bound, originally introduced by Ambainis [18], is one of the most important lower bound techniques for quantum query complexity. A strengthening of the adversary bound, known as the general adversary bound [19], has recently been shown to characterize quantum query complexity, up to constant factors [20], [21].

The (general) adversary bound is a semi-definite program, and admits two equivalent formulations: the primal, used to prove lower bounds; and the dual, used in algorithm construction. We use the latter.

Definition 1. Let $f\colon \mathcal{D} \to \{0,1\}$ with $\mathcal{D} \subseteq [m]^n$ be a function. The adversary bound $\mathrm{Adv}^{\pm}(f)$ is defined as the optimal value of the following optimization problem:

$$\text{minimize} \quad \max_{x\in\mathcal{D}} \sum\nolimits_{j\in[n]} X_j[\![x,x]\!] \tag{1a}$$

$$\text{subject to} \quad \sum\nolimits_{x_j\neq y_j} X_j[\![x,y]\!] = 1 \quad \text{if } f(x)\neq f(y) \tag{1b}$$

$$X_j \succeq 0 \qquad \text{for all } j\in[n] \tag{1c}$$

where the optimization is over positive semi-definite matrices X_j with rows and columns labeled by the elements of $\mathcal{D}$, and $X[\![x,y]\!]$ is used to denote the element of matrix X on the intersection of the row and column labeled by x and y, respectively.

The general adversary bound characterizes quantum query complexity. Let $Q(f)$ denote the query complexity of the best quantum algorithm evaluating f with a bounded error.

Theorem 2 ([19]–[21]). *Let f be as above. Then, $Q(f) = \Theta(\mathrm{Adv}^{\pm}(f))$.*

B. Learning graphs: Model-driven description

In this section we briefly introduce the simplest model of learning graph following Ref. [12], [15].

Definition 3. A *learning graph* $\mathcal{G}$ on n input variables is a directed acyclic connected graph with vertices labeled by subsets of $[n]$, the input indices. It has arcs connecting vertices labeled by S and $S\cup\{j\}$ only, where $S\subseteq[n]$ and $j\in[n]\setminus S$. The root of $\mathcal{G}$ is the vertex labeled by $\emptyset$. Each arc e is assigned positive real *weight* w_e.

Note that it is allowed to have several (or none) vertices labeled by the same subset $S\subseteq[n]$. If there is unique vertex of $\mathcal{G}$ labeled by S, we usually use S to denote it. Otherwise, we denote the vertex by (S,a) where a is some additional

parameter used to distinguish vertices labeled by the same subset S.

A learning graph can be thought of as a way of modeling the development of one's knowledge about the input during a query algorithm. Initially, nothing is known, and this is represented by the root labeled by $\emptyset$. At a vertex labeled by $S \subseteq [n]$, the values of the variables in S have been learned. Following an arc e connecting vertices labeled by S to $S \cup \{j\}$ can be interpreted as querying the value of variable x_j. We say the arc *loads* element j. When talking about a vertex labeled by S, we call S the set of *loaded elements*.

The graph $\mathcal{G}$ itself has a very loose connection to the function being calculated. The following notion is the essence of the construction.

Definition 4. Let $\mathcal{G}$ be a learning graph on n input variables, and $f : \mathcal{D} \to \{0,1\}$ be a function with domain $\mathcal{D} \subseteq [m]^n$. A *flow* on $\mathcal{G}$ is a real-valued function $p_e(x)$ where e is an arc of $\mathcal{G}$ and $x \in f^{-1}(1)$. For a fixed input x, the flow $p_e = p_e(x)$ has to satisfy the following properties:

- vertex $\emptyset$ is the only source of the flow, and it has value 1. In other words, the sum of p_e over all e leaving $\emptyset$ is 1;
- a vertex labeled by S is a sink iff it contains a 1-certificate for f on input x. Such vertices are called *accepting*. Thus, if $S \neq \emptyset$ and S is not accepting then, for a vertex labeled by S, the sum of p_e over all incoming arcs equals the sum of p_e over all out-going arcs.

We always assume a learning graph $\mathcal{G}$ is equipped with a function f and a flow p that satisfy the constraints of Definition 4. Define the *negative complexity* of $\mathcal{G}$ and the *positive complexity for input* $x \in f^{-1}(1)$ as

$$\mathcal{C}^0(\mathcal{G}) = \sum_{e \in E} w_e \qquad \text{and} \qquad \mathcal{C}^1(\mathcal{G}, x) = \sum_{e \in E} \frac{p_e(x)^2}{w_e}, \quad (2)$$

respectively, where E is the set of arcs of $\mathcal{G}$. The *positive complexity* and the *(total) complexity* of $\mathcal{G}$ are defined as

$$\mathcal{C}^1(\mathcal{G}) = \max_{x \in f^{-1}(1)} \mathcal{C}^1(\mathcal{G}, x), \quad \mathcal{C}(\mathcal{G}) = \max\{\mathcal{C}^0(\mathcal{G}), \mathcal{C}^1(\mathcal{G})\}, \quad (3)$$

respectively. The following theorem links learning graphs and quantum query algorithms:

Theorem 5 ([15]). *Assume $\mathcal{G}$ is a learning graph for a function $f : \mathcal{D} \to \{0,1\}$ with $\mathcal{D} \subseteq [m]^n$. Then there exists a bounded-error quantum query algorithm for the same function with complexity $O(\mathcal{C}(\mathcal{G}))$.*

Proof sketch.: We reduce to Theorem 2. For each arc e from S to $S \cup \{j\}$, we define a block-diagonal matrix $X_j^e = \sum_\alpha Y_\alpha$, where the sum is over all assignments α on S. Each Y_α is defined as $\psi\psi^*$ where, for each $z \in \mathcal{D}$:

$$\psi[\![z]\!] = \begin{cases} p_e(z)/\sqrt{w_e}, & f(z) = 1, \text{ and } z \text{ satisfies } \alpha; \\ \sqrt{w_e}, & f(z) = 0, \text{ and } z \text{ satisfies } \alpha; \\ 0, & \text{otherwise.} \end{cases}$$

Finally, we define X_j in (1) as $\sum_e X_j^e$ where the sum is over all arcs e loading j.

Condition (1c) is trivial, and the expression for the objective value (1a) is straightforward to check. The feasibility (1b) is as follows. Fix any $x \in f^{-1}(1)$ and $y \in f^{-1}(0)$. By construction, $X_j^e[\![x, y]\!] = p_e(x)$, if $x_S = y_S$ where S is the origin of e; otherwise, it is zero. Thus, only arcs e from S to $S \cup \{j\}$, such that $x_S = y_S$ and $x_j \neq y_j$, contribute to the sum in (1b). These arcs define a cut between the source $\emptyset$ and all the sinks of the flow $p_e = p_e(x)$, hence, the total value of the flow on these arcs is 1, as required. ∎

C. Learning graphs: Procedure-driven description

In this section, we describe a way of designing learning graphs that was used in Ref. [12] and other papers. The learning graph, introduced in Section II-B, may be considered as a randomized procedure for loading values of the variables with the goal of convincing someone the value of the function is 1. For each input $x \in f^{-1}(1)$, the designer of the learning graph builds its own procedure. The goal is to load a 1-certificate for x. Usually, for each positive input, one specific 1-certificate is chosen. The elements inside the certificate are called *marked*. The procedure is not allowed to err, i.e., it always has to load all the marked elements in the end. The value of the complexity of the learning graph arises from the interplay between the procedures for different inputs.

We illustrate this concepts with an example of a learning graph corresponding to the k-distinctness algorithm by Ambainis [1]. Fix a positive input x, i.e., one evaluating to 1. Let $M = \{a_1, a_2 \ldots, a_k\}$ be such that $x_{a_1} = x_{a_2} = \cdots = x_{a_k}$. It is a 1-certificate for x. The elements inside M are marked. One possible way of loading the marked elements consists of $k+1$ stage and is given in Table I. The internal randomness of the procedure is concealed in the choice of the r elements on stage I. (Here $r = o(n)$ is some parameter to be specified later.) Each choice has probability $q = \binom{n-k}{r}^{-1}$.

I.	Load r elements different from $a_1, \ldots, a_k$.
II.1	Load a_1.
II.2	Load a_2.
	$\vdots$
II.k	Load a_k.

Table I
LEARNING GRAPH FOR THE k-DISTINCTNESS PROBLEM CORRESPONDING TO THE ALGORITHM FROM REF. [1].

Let us describe how a graph $\mathcal{G}$ and flow p is constructed from the description in Table I. At first, we define the *key*

vertices of $\mathcal{G}$. If d is the number of stages, the key vertices are $V_0 \cup \cdots \cup V_d$, where $V_0 = \{\emptyset\}$ and V_i consists of all possible sets of variables loaded after i stages.

For a fixed input x and fixed internal randomness, the sets $S_{i-1} \in V_{i-1}$ and $S_i \in V_i$ of variables loaded before and after stage i, respectively, are uniquely defined. In this case, we connect S_{i-1} and S_i by a *transition* e. For that, we choose an arbitrary order $t_1, \ldots, t_\ell$ of elements in $S_i \setminus S_{i-1}$, and connect S_{i-1} and S_i by a path:

$$S_{i-1}, (S_{i-1} \cup \{t_1\}, e), (S_{i-1} \cup \{t_1, t_2\}, e), \ldots, (S_i \setminus \{t_\ell\}, e), S_i$$

in $\mathcal{G}$. Here, additional labels e in the internal vertices assure that the paths corresponding to the transitions do not intersect, except at the ends. We say transition e and all arcs therein belong to stage i.

In the case like in the previous paragraph, we say the transition e is *taken* for this choice of x and the randomness. We say a transition is *used* for input x, if it is taken for some choice of the internal randomness. The set of transitions of $\mathcal{G}$ is the union of all transitions used for all inputs in $f^{-1}(1)$. For instance, stage II.2 of the learning graph from Table I consists of all transitions from S to $S \cup \{j\}$ where $|S| = r+1$ and $j \notin S$.

The flow $p_e(x)$ is defined as the probability, over the internal randomness, that transition e is taken for input x. All arcs forming the transition are assigned the same flow. Thus, the transition e is used by x iff $p_e(x) > 0$. In the learning graph from Table I, $p_e(x)$ attains two values only: 0 and q.

So far, we have constructed the graph $\mathcal{G}$ and the flow p. It remains to define the weights w_e. This is done using Theorem 6 below. But, for that, we need some additional notions.

The *length* of stage i is the number of variables loaded on this stage, i.e., $|S_i \setminus S_{i-1}|$ for a transition e from S_{i-1} to S_i of stage i. In our applications in this paper this number is independent on the choice of e. We say the flow is *symmetric* on stage i if the non-zero value of $p_e(x)$ is the same for all e on stage i and all x. The flow in the learning graph from Table I is symmetric.

If the flow is symmetric on stage i, we define the *speciality* T_i of stage i as the ratio of the total number of transitions on stage i, to the number of ones used by x. In a symmetric flow, this quantity doesn't depend on x.

Finally, we define the *(total) complexity of stage* i, $\mathcal{C}_i(\mathcal{G})$, similarly as $\mathcal{C}(\mathcal{G})$ is defined in (2) and (3) with the summation over E_i, the set of all arcs on stage i, instead of E. It is easy to see that $\mathcal{C}(\mathcal{G})$ is at most $\sum_i \mathcal{C}_i(\mathcal{G})$.

Theorem 6 ([12]). *If the flow is symmetric on stage i, the arcs on stage i can be weighted so that the complexity of the stage becomes $L_i\sqrt{T_i}$.*

Proof sketch: Let q be the non-zero value of the flow on stage i. Assign weight $q/\sqrt{T_i}$ to all arcs on stage i. ■

Now we are able to calculate the complexity of the learning graph in Table I. The length of stage I is r, and the length of stage II.i is 1 for all i. It is also not hard to see that the corresponding specialities are $O(1)$ and $O(n^i/r^{i-1})$. For example, a transition from S to $S \cup \{j\}$ on stage II.k is used by input x iff $a_1, \ldots, a_{k-1} \in S$ and $j = a_k$. For a random choice of S and $j \notin S$, the probability of $j = a_k$ is $1/n$, and the probability of $a_1, \ldots, a_{k-1} \in S$, given $j = a_k$, is $\Omega(r^{k-1}/n^{k-1})$. Thus, the total probability is $\Omega(r^{k-1}/n^k)$ and the speciality is the inverse of that.

Thus, the complexity of the algorithm, by Theorems 6 and 5, is $O(r + \sqrt{n^k/r^{k-1}})$. It is optimized when $r = n^{k/(k+1)}$, and the complexity is $O(n^{k/(k+1)})$.

III. Outline of the Algorithm

In this section we describe how the learning graph from Table I is transformed into a new learning graph with a better complexity. Many times when learning graphs were applied to new problems, they were modified accordingly [12], [15], [16]. This paper is not an exception, thus, we also describe the modifications we make to the model of a learning graph.

The main point of the learning graph in Table I and similar ones is to reduce the speciality of the last step, loading a_k. In the learning graph from Table I, it is achieved by loading r non-marked elements before loading the certificate. This way, the speciality of the last step gets reduced from $O(n^k)$ to $O(n^k/r^{k-1})$. We say that $a_1, \ldots, a_{k-1}$ are *hidden* among the r elements loaded on stage I. The larger the set we hide the elements into, the better.

Unfortunately, we can't make r as large as we like, because loading the non-marked elements also counts towards the complexity. At the equilibrium point $r = n^{k/(k+1)}$, we attain the optimal complexity of the learning graph.

In Ref. [15] a learning graph was constructed with better complexity. It uses a more general version of the learning graph than in Section II-B, with weights of the arcs dependent on the values of the element loaded so far. Its main idea is to hide $a_1, \ldots, a_{k-1}$ as one entity, not $k-1$ independent elements. By gradually distilling vertices of the learning graph having large number of $(k-1)$-tuples of equal elements, the learning graph manages to reduce the speciality of the last step without increasing the number of elements loaded, because $\{a_1, \ldots, a_{k-1}\}$ gets hidden among a relatively large number of $(k-1)$-tuples of equal elements.

But this learning graph has serious drawbacks. Due to dealing with the values of the variables in the distilling phase, the flow through the learning graph ceases to be symmetric and depends heavily on the input. This makes the analysis of the learning graph quite complicated. What is even worse, the learning graph requires strong prior knowledge on the structure of the input to attain reasonable complexity.

In this paper we construct a learning graph that combines the best features of both learning graphs. Its complexity is

the same as in Ref. [15]. Also, it has the flow symmetric and almost independent on the input, like the one in Table I. This has three advantages compared to the learning graph in Ref. [15]: its complexity is easier to analyze, it doesn't require any prior information on the input, and it is more suitable for a time-efficient implementation along the lines of Ref. [16]. This is achieved at the cost of a more involved construction.

Let us outline the modifications the learning graph from Table I undergoes in order to reduce the complexity. Again, we assume x is a positive input, and $M = \{a_1, \ldots, a_k\}$ is such that $x_{a_1} = \cdots = x_{a_k}$.

1) We achieve a symmetric flow with smaller speciality of the last step by finding a way to load more non-marked elements in the first stages of the learning graph. There is an indication that it is possible in some cases: the values of r Boolean variables can be learned in less than r queries, if there is a bias between the number of ones and zeros [22]. More precisely, if the number of ones is ℓ, the values can be loaded in $O(\sqrt{r\ell})$ queries.
2) We start with dividing the set S of loaded elements into k subsets: $S = S_1 \sqcup \cdots \sqcup S_{k-1}$, where $\sqcup$ denotes disjoint union. Set S_i has size $r_i = o(n)$. We use S_i to hide a_i when loading a_k. This step doesn't reduce the speciality, but this division will be necessary further.
3) Consider the situation before loading a_k. If an element $j \in S_2$ is such that $x_j \neq x_t$ for all $t \in S_1$, this element cannot be a part of the certificate (i.e., it can't be a_2), and its precise value is irrelevant. (This is the place where we utilize the relations between the values of the variables as mentioned in the introduction.) In this case, we say j doesn't have a *match* in S_1, and represent it by a special symbol $\star$. Otherwise, we *uncover* the element, i.e., load its precise value. Similarly, when loading S_i with $i > 2$, we uncover those elements only that have a match among the uncovered elements of S_{i-1}.
4) Usually, the number of elements in S_i having a match in S_{i-1} is much smaller than the total number of elements in S_i. Similarly to Point 1, we can reduce the complexity of loading elements in S_i because of this bias. Thus, we have $r_i = \omega(r_1)$, while the complexity of loading remains $O(r_1)$. Now we have more elements to hide a_i in between, hence, the speciality of loading a_k gets reduced.
5) When loading a_k, we do want a_i to be in S_i for $i \in [k-1]$, because that is where we hide them. On the other hand, in order to keep the speciality of loading non-marked elements in $S_1, \ldots, S_{k-1}$ equal to $O(1)$, we would like to add a_1 to S_1 only after all elements in S_{k-1} have been already loaded. Thus, we load $a_1, \ldots, a_{k-1}$ between these two stages and put them in $S_1, \ldots, S_{k-1}$. This is summarized in Table II.
6) Since the uncovering of elements in S_i, for $i > 1$, depends on the values contained in S_j with $j < i$, adding a_i to S_i afterwards is a bit of cheating. This does cause some problems we describe in more detail in Section IV-C. We describe a solution in Section V.

I.1	Load a set S_1 of r_1 elements not from M.
I.2	Load a set S_2 of r_2 elements not from M, uncovering those elements only that have a match in S_1.
	$\vdots$
I.$(k-1)$	Load a set S_{k-1} of r_{k-1} elements not from M, uncovering those elements only that have a match among uncovered elements of S_{k-2}.
II.1	Load a_1 and add it to S_1.
	$\vdots$
II.$(k-1)$	Load a_{k-1} and add it to S_{k-1}.
II.k	Load a_k.

Table II
AN ILLUSTRATIVE (NOT CORRECT) VERSION OF THE LEARNING GRAPH FOR k-DISTINCTNESS

In order to account for these changes, we use the following modifications to the learning graph model.

A) In Section V, we are forced to drop the flow notion from Definition 4. We use Theorem 2 directly, borrowing some concepts from the proof of Theorem 5. Namely, the notion of a vertex and an arc leaving it. Also, we keep the internal randomness intuition from Section II-C. The loading procedure still doesn't err in some sense formalized in (11).
B) We change the way the vertices of the learning graph are represented. Firstly, we keep track to which S_i each loaded element belongs, like said in Point 2. Also, we assume the condition on uncovering of elements, and use the special symbol $\star$ as a notation for a covered element, as described in Point 3. Technically, this corresponds to modification of the definition of an assignment α in Y_α in the proof of Theorem 5.
C) Instead of having a rank-1 matrix Y_α as in the proof of Theorem 5, we define it as a rank-2 matrix. The weight of the arc depends now on the value of the variable being loaded as well, although in a rather restricted form. Thus, we are able to make use of the bias as described in Point 4, and to account for the introduction of $\star$ in Point 3.

In Sections IV and V we describe the algorithm for k-distinctness. In order to simplify the exposition, we first give a version of the learning graph from Table II that illustrates the main idea of the algorithm, but has a flaw. We identify it in Section IV-C and then describe a workaround in Section V. The complexity analysis of the second algorithm is analogous to the first one, so we do it for the first algorithm.

IV. Algorithm for k-distinctness: First attempt

The aim of this and the next sections is to prove the following theorem:

Theorem 7. *For arbitrary but fixed integer $k \geq 2$, the k-distinctness problem can be solved by a quantum computer in $O\left(n^{1-2^{k-2}/(2^k-1)}\right)$ queries with a bounded error.*

As mentioned in Section III, we do not rely on previous results like Theorem 6 in the proof, and use Theorem 2 directly. The construction of the algorithm deviates from the graph representation: a bit in Section IV, and quite strongly in Section V. However, we keep the term "vertex" for an entity describing some knowledge of the values of the input variables, and the term "arc" for a process of loading a value of a variable (possibly, only partially). Each arc originates in a vertex, but we do not specify where it goes. Inspired by Section II-C, the vertices are divided into *key* ones denoted by the *set of loaded variables* S with additional structure. The non-key vertices are denoted by (S, R) where S is the set of loaded variables, and R is an additional label used to distinguish vertices with the same S, as described in Section II-B. Also, we use the "internal randomness" term from Section II-C.

Throughout Sections IV and V, let $f : [m]^n \to \{0,1\}$ be the k-distinctness function. The section is organized as follows. In Section IV-A, we rigorously define the learning graph from Table II; in Section IV-B, analyze its complexity; and, finally, describe the flaw mentioned in Point 6 of Section III in Section IV-C.

Similarly to the analysis in Ref. [1], we may assume there is unique k-tuple of equal elements in any positive input. One of the simplest reductions to this special case is to take a sequence T_i of uniformly random subsets of $[n]$ of sizes $(2k/(2k+1))^i n$, and to run the algorithm, for each i, with the input variables outside T_i removed. One can prove that if there are k equal elements in the input then there exists i such that, with probability at least $1/2$, T_i will contain unique k-tuple of equal elements. The complexities of the executions of the algorithm for various i form a geometric series, and their sum is equal to the complexity of the algorithm for $i = 0$ up to a constant factor. Refer to Ref. [1] for more detail and alternative reductions.

A. Construction

Let x be a positive input, and let $M = \{a_1, a_2, \ldots, a_k\}$ denote the unique k-tuple of equal elements in x. The key vertices of the learning graph are $V_1 \cup \cdots \cup V_k$, where V_s, for $s \in [k]$, consists of all $(k-1)$-tuples $S = (S_1, \ldots, S_{k-1})$ of pairwise disjoint subsets of $[n]$ of the following sizes. For V_s, we require that $|S_i| = r_i + 1$ for $i < s$, and $|S_i| = r_i$ for $i \geq s$.

Again, a vertex $R = (R_1, \ldots, R_{k-1}) \in V_1$ completely specifies the internal randomness. We assume that, for any $R \in V_1$, an arbitrary order $t_1, \ldots, t_r$ of the elements in $\bigcup R = R_1 \cup \cdots \cup R_{k-1}$ is fixed so that all elements of R_i precede all elements of R_{i+1} for all $i \leq k-2$. (Here $r = \sum_i r_i$.) We say $R \in V_1$ is *consistent* with x if $\{a_1, \ldots, a_k\} \cap (\bigcup R) = \emptyset$.

For each $x \in f^{-1}(1)$, there are exactly $\binom{n-k}{r_1,\ldots,r_{k-1}}$ choices of $R \in V_1$ consistent with x. We take each of them, in the sense of Section II-C, with probability $q = \binom{n-k}{r_1,\ldots,r_{k-1}}^{-1}$.

For a fixed input x and fixed randomness $R \in V_1$ consistent with x, the elements are loaded in the following order:

$$t_1, t_2, \ldots, t_r, t_{r+1} = a_1, t_{r+2} = a_2, \ldots, t_{r+k} = a_k. \quad (4)$$

The non-key vertices of $\mathcal{G}$ are of the form $v = (R \cap \{t_1, \ldots, t_\ell\}, R)$, where $R \in V_1$, $0 \leq \ell < r$, and $\{t_i\}$ are from (4). Here we use notation $R \cap T = (R_1 \cap T, \ldots, R_{k-1} \cap T)$. The first element of the pair describes the set of loaded elements.

Let us describe the arcs A_j^v of $\mathcal{G}$, where, again, j is the variable the arc loads, and v is the vertex of $\mathcal{G}$ it originates in. The arcs of the stages I.s have $v = (R \cap \{t_1, \ldots, t_\ell\}, R)$ and $j = t_{\ell+1}$ with $0 \leq \ell < r$. The arc belongs to stage I.s iff $t_{\ell+1} \in R_s$. The arcs of stage II.s have $v = S$, with $S \in V_s$, and $j \notin \bigcup S$.

For a fixed $x \in f^{-1}(1)$ and fixed internal randomness $R \in V_1$ consistent with x, the following arcs are *taken*:

$$A_{t_{\ell+1}}^{(R \cap \{t_1,\ldots,t_\ell\},R)} \quad \text{and} \quad A_{a_{\ell+1}}^{R[a_1,\ldots,a_\ell]}. \quad (5)$$

Here $R[a_1, a_2, \ldots, a_\ell] = (R_1 \cup \{a_1\}, R_2 \cup \{a_2\} \ldots, R_\ell \cup \{a_\ell\}, R_{\ell+1}, \ldots, R_{k-1})$. We say x *satisfies* all these arcs. Note that, for a fixed x, no arc is taken for two different choices of R.

Again, for each arc A_j^v, we assign a matrix $X_j^v \succeq 0$, so that X_j in (1) are given by $X_j = \sum_v X_j^v$. Assume A_j^v is fixed. Let $S = (S_1 \ldots, S_{k-1})$ be the set of loaded elements. Define an assignment on S as a function $\alpha \colon \bigcup S \to [m] \cup \{\star\}$, where $\star$ represents the *covered* elements of stages I.s for $s > 1$. Thus, α must satisfy $\star \notin \alpha(S_1)$ and $\alpha(S_{i+1}) \subseteq \alpha(S_i) \cup \{\star\}$ for $1 \leq i \leq k-2$. An input $z \in [m]^n$ *satisfies* assignment α iff, for each $t \in \bigcup S$,

$$\alpha(t) = \begin{cases} z_t, & t \in S_1; \\ z_t, & t \in S_i \text{ for } i > 1 \text{ and } z_t \in \alpha(S_{i-1}); \\ \star, & \text{otherwise.} \end{cases}$$

Each input z satisfies unique assignment on S. Again, we say inputs x and y *agree* on S, if they satisfy the same assignment on S.

We define X_j^v as $\sum_\alpha Y_\alpha$ where the sum is over all assignments α on S. The definition of Y_α depends on whether A_j^v is on stage I.s with $s > 1$, or not. If A_j^v is

not on one of these stages then $Y_\alpha = q\psi\psi^*$ where, for each $z \in [m]^n$,

$$\psi[\![z]\!] = \begin{cases} 1/\sqrt{w}, & f(z) = 1, \text{ and } z \text{ satisfies } \alpha \text{ and } A_j^v; \\ \sqrt{w}, & f(z) = 0, \text{ and } z \text{ satisfies } \alpha; \\ 0, & \text{otherwise.} \end{cases}$$

Here w is a positive real number: the weight of the arc. It only depends on the stage of the arc, and will be specified later. Thus, X_j^v consists of the blocks of the following form:

	x	y
x	q/w	q
y	q	qw

(6)

Here x and y represent inputs mapping to 1 and 0, respectively, all satisfying some assignment α. The inputs represented by x have to satisfy the arc A_j^v as well.

If A_j^v is on stage I.s with $s > 1$, the elements having a match in S_{s-1} and the ones that don't must be treated differently. In this case, $Y_\alpha = q(\psi\psi^* + \phi\phi^*)$, where

$$\psi[\![z]\!] = \begin{cases} 1/\sqrt{w_1}, & f(z) = 1,\ z_j \in \alpha(S_{s-1}), \\ & \text{and } z \text{ satisfies } \alpha \text{ and } A_j^v; \\ \sqrt{w_1}, & f(z) = 0, \text{ and } z \text{ satisfies } \alpha; \\ 0, & \text{otherwise;} \end{cases}$$

and

$$\phi[\![z]\!] = \begin{cases} 1/\sqrt{w_0}, & f(z) = 1,\ z_j \notin \alpha(S_{s-1}), \\ & \text{and } z \text{ satisfies } \alpha \text{ and } A_j^v; \\ \sqrt{w_0}, & f(z) = 0,\ z_j \in \alpha(S_{s-1}), \\ & \text{and } z \text{ satisfies } \alpha; \\ 0, & \text{otherwise.} \end{cases}$$

Here w_0 and w_1 are again parameters to be specified later. In other words, X_j^v consists of the blocks of the following form:

$x_j \in \alpha(S_{s-1})$	q/w_1	0	q	q
$x_j \notin \alpha(S_{s-1})$	0	q/w_0	q	0
$y_j \in \alpha(S_{s-1})$	q	q	$q(w_0+w_1)$	qw_1
$y_j \notin \alpha(S_{s-1})$	q	0	qw_1	qw_1

(7)

Here x and y are like in (6). Note that if x_j and y_j are both represented by $\star$ in the assignments on $(S_1, \dots, S_{s-1}, S_s \cup \{j\}, S_{s+1}, \dots, S_{k-1})$ they satisfy then $X_j^v[\![x, y]\!] = 0$.

B. Complexity

Similarly to Section II-C, let us define the complexity of stage i on input $z \in [m]^n$ as $\sum_{j\in[n]} X'_j[\![z, z]\!]$, where $X'_j = \sum_v X_j^v$ with the sum over v such that A_j^v belongs to stage i. Also, define the complexity of stage i as the maximum complexity over all inputs $z \in \{0,1\}^n$. Clearly, the objective value (1a) of the whole program is at most the sum of the complexities of all stages.

Let us start with stage I.1. We set $w = 1$ for all arcs on this stage. There are $r_1\binom{n}{r_1,\dots,r_{k-1}}$ arcs on this stage, and, by (6), each of them contributes at most q to the complexity of each $z \in \{0,1\}^n$. Hence, the complexity of stage I.1 is $O\left(qr_1\binom{n}{r_1,\dots,r_{k-1}}\right) = O(r_1)$.

Now consider stage II.s for $s \in [k]$. The total number of arcs on the stage is $(n-r-s+1)\binom{n}{r_1+1,\dots,r_{s-1}+1,r_s,\dots,r_{k-1}}$. By (6), each of them contribute qw to the complexity of each $y \in f^{-1}(0)$. Out of these arcs, for any $x \in f^{-1}(1)$, exactly $\binom{n-k}{r_1,\dots,r_{k-1}}$ satisfy x. And each of them contribute q/w to the complexity of x. Thus, the complexities of stage II.s for any input in $f^{-1}(0)$ and $f^{-1}(1)$ are $O\left(n^s w/(r_1\cdots r_{s-1})\right)$, and $1/w$, respectively. By setting $w = \left(n^s/(r_1\cdots r_{s-1})\right)^{-1/2}$, we get complexity $O\left(\sqrt{n^s/(r_1\cdots r_{s-1})}\right)$ of stage II.s. The maximal complexity is attained for stage II.k.

Now let us calculate the complexity of stage I.s for $s > 1$. The total number of arcs on this stage is $r_s\binom{n}{r_1,\dots,r_{k-1}}$. Consider an input $z \in [m]^n$, and a choice of the internal randomness $R = (R_1, \dots, R_{k-1}) \in V_1$. An element j is uncovered on stage I.s for this choice of R if and only if there is an s-tuple $(b_1, \dots, b_s)$ of elements with $j = b_s$ such that $b_i \in R_i$ and $z_{b_i} = z_{b_j}$ for all $i, j \in [s]$. By our assumption on the uniqueness of a k-tuple of equal elements in a positive input, the total number of such s-tuples is $O(n)$. And, for each of them, there are $\binom{n-s}{r_1-1,\dots,r_s-1,r_{s+1},\dots,r_{k-1}}$ choices of $R \in V_1$ such that $b_i \in R_i$ for all $i \in [s]$. By (7), the complexities of this stage for an input in $f^{-1}(0)$ and in $f^{-1}(1)$ are, respectively, at most $O\left(\frac{r_1\cdots r_s}{n^{s-1}}w_0 + r_s w_1\right)$ and $O\left(\frac{r_1\cdots r_s}{n^{s-1}w_1} + \frac{r_s}{w_0}\right)$. By assigning $w_0 = \sqrt{n^{s-1}/(r_1\cdots r_{s-1})}$ and $w_1 = \sqrt{r_1\cdots r_{s-1}/n^{s-1}}$, both these quantities become $O\left(r_s\sqrt{r_1\cdots r_{s-1}/n^{s-1}}\right)$.

With this choice of the weights, the value of the objective function in (1a) is, up to a constant factor,

$$r_1 + r_2\sqrt{\frac{r_1}{n}} + \dots + r_{k-1}\sqrt{\frac{r_1\cdots r_{k-2}}{n^{k-2}}} + \sqrt{\frac{n^k}{r_1\cdots r_{k-1}}} \quad (8)$$

Assuming all terms in (8) except the last one are equal, and denoting $\rho_i = \log_n r_i$, we get that

$$\rho_i + \frac{1}{2}(\rho_1 + \dots + \rho_{i-1}) - \frac{i-1}{2} = \rho_{i+1} + \frac{1}{2}(\rho_1 + \dots + \rho_i) - \frac{i}{2}$$

or, equivalently,

$$\rho_{i+1} = \frac{1+\rho_i}{2}, \qquad \text{for } i = 1, \dots, k-2.$$

Assuming the first term, r_1, equals the last one, we get $\rho_1 = 1 - 2^{k-2}/(2^k - 1)$, hence, the complexity of the algorithm is $O\left(n^{1-2^{k-2}/(2^k-1)}\right)$.

C. (In)feasibility

Assume x and y are inputs such that $f(x) = 1$ and $f(y) = 0$. Let $R = (R_1, \dots, R_{k-1}) \in V_1$ be a choice of the internal randomness consistent with x. Let Z_j be the

matrix corresponding to the arc loading j that is taken for input x and randomness R (i.e., the one from (5) with sub-index j, or the zero matrix, if there are none).

We would like to prove that

$$\sum_{j:x_j\neq y_j} Z_j[\![x,y]\!] = q. \tag{9}$$

(This is what we meant by saying in Point A of Section III that the learning graph doesn't err for all choices of the internal randomness.) Unfortunately, it doesn't always hold. Assume x, y and $R \in V_1$ are such that x and y agree on R. Thus, the contribution to (9) is 0 from all arcs of stages I.s. Now assume that $x_{a_1} = y_{a_1}$ and there exists $b \in R_2$ such that $y_b = x_{a_1}$. This doesn't contradict that x and y agree on R, because y_b is represented by $\star$ in the assignment it satisfies on R.

But x and y disagree on $R[a_1]$, because y_b gets uncovered there. Thus, the contribution to (9) is 0 from all arcs of stages II.s as well. Thus, equation (9) doesn't hold. We deal with this problem in the next section.

V. Final version

In Section IV-C, we saw that the learning graph in Table II is incorrect. This is due to *faults*. A fault is an element b of R_i with $i > 1$ such that $y_b = x_{a_1}$. This is the only element that can suddenly uncover itself when adding a_{i-1} to R_{i-1} on stage II.$(i-1)$, because we have assumed x contains a unique k-tuple of equal elements, hence, if $R \in V_1$ is consistent with x, no b in $\bigcup R$ satisfies $x_b = x_{a_1}$.

But since y is a negative input, there are at most $k - 1 = O(1)$ faults for every choice of x. Thus, all we need is to develop a fault-tolerant version of the learning graph from Table II that is capable of dealing with this number of faults.

As an introductory example, consider case $k = 3$. In this case, a fault may only occur in R_2. A fault may come in action only if $y_{a_1} = x_{a_1}$, hence, we may assume there are at most $k-2$ faults in any y. Split R_2 into $k-1$ subsets $\{R_2(d)\}_{d\in[k-1]}$. We know that at least one of them is not faulty, but it is not enough: we have to assure the contribution from these arcs is q exactly, no matter how many of $R_2(d)$ are faulty, i.e., a variant of (9). We achieve this by splitting R_1 into $2^{k-1}-1$ parts $\{R_1(D)\}$ labeled by non-empty subsets D of $[k-1]$. We uncover an element in $R_2(d)$ if and only if it has a match in $R_1(D)$ for some $D \ni d$. By adding a_1 to $R_1(D)$, we can test whether $\bigcup_{d\in D} R_2(d)$ contains a fault. This is enough to guarantee (9) by an application of the inclusion-exclusion principle. The construction in Section V-A is a generalization of this idea for arbitrary k.

A. Construction

The key vertices of the learning graph are $V_1 \cup \cdots \cup V_k$, where V_s consists of all collections of pairwise disjoint subsets $S = \big(S_i(d_1, d_2, \ldots, d_{i-1}, D)\big)$ labeled by $i \in [k-1]$, $d_j \in [k-j]$, and $\emptyset \subset D \subseteq [k-i]$. There are additional requirements on the sizes of these subsets.

For a non-empty subset $D \subset \mathbb{N}$, let $\mu(D)$ denote the minimal element of D. (Actually, any fixed element of D works as well.) For each sequence $(D_1, \ldots, D_{s-1})$, where D_i is a non-empty subset of $[k-i]$, let $V_s(D_1, \ldots, D_{s-1})$ consist of all collections $\big(S_i(d_1, d_2, \ldots, d_{i-1}, D)\big)$ such that

$$|S_i(d_1, \ldots, d_{i-1}, D)| = \begin{cases} r_i + 1, & i < s,\ d_j = \mu(D_j), \text{ and } D = D_i; \\ r_i, & \text{otherwise.} \end{cases}$$

Finally, let V_s be the union of $V_s(D_1, \ldots, D_{s-1})$ over all choices of $(D_1, \ldots, D_{s-1})$.

Again, a vertex in $R = \big(R_i(d_1, d_2, \ldots, d_{i-1}, D)\big) \in V_1$ completely specifies the internal randomness. For each of them, we fix an arbitrary order $t_1, \ldots, t_r$ of elements in $\bigcup R$ so that all elements of R_i precede all elements of R_{i+1} for all $i \leq k-2$. We say R is consistent with x, if $\{a_1, \ldots, a_k\}$ is disjoint from $\bigcup R$. Let q be the inverse of the number of $R \in V_1$ consistent with x. (Clearly, this number is the same for all choices of x.)

The elements still are loaded in the order from (4). We use a similar convention to name the arcs of the learning graph as in Section IV. Arcs of stages I.s are of the form $A_{t_{\ell+1}}^{(R\cap\{t_1,\ldots,t_\ell\},R)}$ for $R \in V_1$ and $0 \leq \ell < r$. Here, $R \cap T = \big(S_i(d_1, d_2, \ldots, d_{i-1}, D)\big)$ is defined by $S_i(d_1, d_2, \ldots, d_{i-1}, D) = R_i(d_1, d_2, \ldots, d_{i-1}, D) \cap T$. Arcs of stage II.$s$ are of the form A_j^R with $R \in V_s$ and $j \notin \bigcup R$.

For any $x \in f^{-1}(1)$ and $R \in V_1$ consistent with x, the following arcs are taken. On stage I.s, for $s \in [k-1]$, these are arcs $A_{t_{\ell+1}}^{(R\cap\{t_1,\ldots,t_\ell\},R)}$, where $t_{\ell+1}$ belongs to one of R_s. On stage II.s, for $s \in [k]$, we have many arcs loading a_s. For each choice of $(D_i)_{i\in[s-1]}$ where D_i is a non-empty subset of $[k-i]$, the arc $A_{a_s}^{R[D_1\leftarrow a_1,\ldots,D_{s-1}\leftarrow a_{s-1}]}$ is taken where $R[D_1 \leftarrow a_1, \ldots, D_{s-1} \leftarrow a_{s-1}] = \big(S_i(d_1, d_2, \ldots, d_{i-1}, D)\big)$ is defined by $S_i(d_1, \ldots, d_{i-1}, D) = R_i(d_1, \ldots, d_{i-1}, D) \cup \{a_i\}$, if $i < s$, $d_j = \mu(D_j)$, and $D = D_i$, and $S_i(d_1, \ldots, d_{i-1}, D) = R_i(d_1, \ldots, d_{i-1}, D)$, otherwise.

For each arc A_j^v, we define a positive semi-definite matrix X_j^v so that X_j in (1) are given by $\sum_v X_j^v$. Fix an arc A_j^v and let $S = \big(S_i(d_1, d_2, \ldots, d_{i-1}, D)\big)$ be the set of loaded elements. This time, we define an assignment on S as a function $\alpha\colon \bigcup S \to [m] \cup \{\star\}$ such that $\star \notin \bigcup_D \alpha(S_1(D))$, and, for all $i > 1$ and all possible choices of $d_1, \ldots, d_{i-1}$ and D, $\alpha(S_i(d_1, d_2, \ldots, d_{i-1}, D)) \subseteq \{\star\} \cup \bigcup_{K\ni d_{i-1}} \alpha(S_{i-1}(d_1, \ldots, d_{i-2}, K))$.

An input $z \in [m]^n$ satisfies assignment α iff, for each $t \in \bigcup S$, $\alpha(t) = z_t$, if $t \in S_1(D)$ or $t \in S_i(d_1, \ldots, d_{i-1}, D)$ and $z_t \in \bigcup_{K\ni d_{i-1}} \alpha(S_{i-1}(d_1, \ldots, d_{i-2}, K))$, and $\alpha(t) = \star$, otherwise. We say inputs x and y agree on S, if they satisfy the same assignment α.

Like before, we define X_j^v as $\sum_\alpha Y_\alpha$ where the sum is over all assignments α on S. For the arcs on stage I.1, Y_α are defined as in (6), and the arcs on stage I.s, for $s > 1$, are defined as in (7) with $\alpha(S_{s-1})$ replaced by $\bigcup_{K \ni d_{s-1}} \alpha(S_{s-1}(d_1, \dots, d_{s-2}, K))$.

Now consider stage II.s. Let A_j^S be an arc with $S \in V_s(D_1, \dots, D_{s-1})$. In this case, $Y_\alpha = q\psi\psi^*$ where

$$\psi[\![z]\!] = \begin{cases} 1/\sqrt{w}, & f(z) = 1, \text{ and } z \text{ satisfies } \alpha \text{ and } A_j^S; \\ \pm\sqrt{w}, & f(z) = 0, \ z \text{ satisfies } \alpha; \\ 0, & \text{otherwise;} \end{cases}$$

where there is $+$ in the second case, if and only if $s + |D_1| + \dots + |D_{s-1}|$ is odd. Thus, depending on the parity of $s + |D_1| + \dots + |D_{s-1}|$, X_j^S consists of the blocks of one of the following two types:

	x	y
x	q/w	q
y	q	qw

or

	x	y
x	q/w	$-q$
y	$-q$	qw

(10)

Complexity: Before we go on proving the correctness of this modified learning graph, let us consider the complexity issue. The complexity analysis follows the same lines as in Section IV-B. The complexity of stages I.s is proved similarly, by taking $R_i = \bigcup_{d_1,\dots,d_{i-1},D} R_i(d_1, \dots, d_{i-1}, D)$, and noting that $|R_i| = O(k!)r_i = O(r_i)$. Of course, having a match in R_{i-1} is not sufficient for an element in R_i to be uncovered, but this only reduces the complexity. The analysis of stage II.s is also similar, but this time instead of one arc loading element a_s for a fixed choice of x and $R \in V_1$, there are $2^{O(k^2)} = O(1)$ of them.

B. Feasibility

Fix inputs $x \in f^{-1}(1)$ and $y \in f^{-1}(0)$, and let $R \in V_1$ be a choice of the internal randomness consistent with x. Compared to the learning graph in Section IV, for a fixed $j \in [n]$, many arcs of the form A_j^v may be taken, thus, we have to modify the Z_j notation. Let $\mathcal{Z}$ be the set of arcs taken for this choice of x and R. The complete list is in Section V-A. We prove that

$$\sum_{A_j^v \in \mathcal{Z} \colon x_j \neq y_j} X_j^v[\![x, y]\!] = q. \tag{11}$$

Since, again, no arc is taken for two different choices of $R \in V_1$, this proves feasibility (1b).

If x and y disagree on R then (11) holds. It is not hard to check that there exists $i \in [r]$ such that x and y disagree on $R \cap \{t_1, \dots, t_{i'}\}$ if and only if $i' \geq i$. Let $j = t_i$, $T = \{t_1, \dots, t_{i-1}\}$, $S = R \cap T$ and $S' = R \cap (T \cup \{j\})$. We claim that $X_j^{(S,R)}[\![x, y]\!] = q$ and $x_j \neq y_j$.

Indeed, let α be the assignment x and y both satisfy on S, and let α_x and α_y be the assignments x and y, respectively, satisfy on S'. By the order imposed on the elements in (4), we get that $\alpha(t) = \alpha_x(t) = \alpha_y(t)$ for all $t \in T$. Since x and y disagree on S', it must hold that $\alpha_x(j) \neq \alpha_y(j)$. Hence, $x_j \neq y_j$, and at least one of the is not represented by $\star$ in the assignment on S'. Thus, $X_j^{(S,R)}[\![x, y]\!] = q$ by (6) or (7), in dependence on whether $A_j^{(S,R)}$ belongs to stage I.1 or not.

We claim the contribution to the sum in (11) from the arcs in $\mathcal{Z}$ loading $t_{i'}$ for $i' \in [r + k] \setminus \{i\}$ is zero. For $i' > i$, this follows from that x and y disagree before loading $t_{i'}$. Now consider $i' < i$. Inputs x and y agree on $S = R \cap \{t_1, \dots, t_{i'}\}$. Let $j' = t_{i'}$ and α be the assignment x and y both satisfy on S. We have either $x_{j'} = y_{j'}$, or they both are represented by $\star$ in α. In both cases, the contribution is zero (in the second case, by (7)).

Now assume x and y agree on R. The contribution to (11) from the arcs of stages I.s is 0 by the same argument as in the previous paragraph. Let s be the first element such that $x_{a_s} \neq y_{a_s}$. We claim that if $s' \neq s$, the contribution to (11) from the arcs $A_{a_{s'}}^S \in \mathcal{Z}$ with $S \in V_{s'}$ is 0.

Indeed, if $s' < s$ then $x_{a_{s'}} = y_{a_{s'}}$. If $s' > s$, for each choice of $(D_i)_{i \in [s'-1]}$, x and y disagree on $R[D_1 \leftarrow a_1, \dots, D_{s'-1} \leftarrow a_{s'-1}]$, because, by construction, all a_i with $i < s'$ are uncovered in the assignment of x.

The total contribution from the arcs $A_{a_s}^S \in \mathcal{Z}$ with $S \in V_s$ is q. This is a special case of Lemma 8 below. Before stating the lemma we have to introduce additional notations. For a vertex $S = R[D_1 \leftarrow a_1, \dots, D_\ell \leftarrow a_\ell]$ of the learning graph with $\ell < s$, let the *block* $\mathcal{B}(S)$ on this vertex be defined as the set of vertices $R[D_1 \leftarrow a_1, \dots, D_{s-1} \leftarrow a_{s-1}]$, where $\emptyset \subset D_i \subseteq [k - i]$ for $i = \ell + 1, \dots, s - 1$. Also, define the *contribution* of the block on this vertex as $\mathcal{C}(S) = \sum_{S' \in \mathcal{B}(S)} X_{a_s}^{S'}[\![x, y]\!]$. We prove the following lemma by induction on $s - \ell$:

Lemma 8. *Let R and s be as above. If x and y agree on $S = R[D_1 \leftarrow a_1, \dots, D_\ell \leftarrow a_\ell]$ then the contribution from the block on S is $(-1)^{\ell + |D_1| + \dots + |D_\ell|} q$. Otherwise, it is 0.*

Note that if $\ell = 0$, the lemma states that the contribution of the block on R is q. But this block consists of all arcs of the form $A_{a_s}^S$ from $\mathcal{Z}$. Thus, this proves (11).

Proof of Lemma 8: If x and y disagree on S, they disagree on any vertex from the block, hence, the contribution is 0.

Now assume x and y agree on S. If $\ell = s - 1$, there is only S in the block. Hence, the contribution is $(-1)^{\ell + |D_1| + \dots + |D_\ell|} q$ by (10), because x and y agree on S and $x_{a_s} \neq y_{a_s}$. Now assume $\ell < s - 1$, and the lemma holds for ℓ replaced by $\ell + 1$. The block $\mathcal{B}(S)$ can be expressed as the following disjoint union:

$$\bigsqcup_{\emptyset \subset D_{\ell+1} \subseteq [k-\ell-1]} \mathcal{B}(R[D_1 \leftarrow a_1, \dots, D_{\ell+1} \leftarrow a_{\ell+1}]).$$

Let I be the set of $i \in [k - \ell - 1]$ such that $\bigcup_D R_{\ell+2}(\mu(D_1), \dots, \mu(D_\ell), i, D)$ does not contain a fault.

It is not hard to see that x and y agree on $R[D_1 \leftarrow a_1, \ldots, D_{\ell+1} \leftarrow a_{\ell+1}]$ if and only if $D_{\ell+1} \subseteq I$. Since $y_{a_1} = \cdots = y_{a_{s-1}} = x_{a_1}$ and there is at most $k-1$ element in y equal to x_{a_1}, there are at most $k-1-(s-1) < k-\ell-1$ faults. Hence, I is non-empty. Using the inductive assumption,

$$\begin{aligned}\mathcal{C}(S) &= \sum_{\emptyset \subset D_{\ell+1} \subseteq [k-\ell-1]} \mathcal{C}(R[D_1 \leftarrow a_1, \ldots, D_{\ell+1} \leftarrow a_{\ell+1}]) \\ &= \sum_{\emptyset \subset D_{\ell+1} \subseteq I} (-1)^{\ell+1+|D_1|+\cdots+|D_{\ell+1}|} q \\ &= (-1)^{\ell+|D_1|+\cdots+|D_\ell|} q,\end{aligned}$$

by inclusion-exclusion. ∎

VI. Conclusion

A quantum query algorithm for k-distinctness is presented in the paper. The algorithm uses the learning graph framework. The improvement in complexity is due to a sequence of new ideas enhancing the framework: partial assignments in the vertices of the learning graph, arcs with the weight dependent on the variable being loaded, fault-tolerant learning graphs, and others.

The future research may concentrate on the following problems. Is it possible to use some of these ideas to improve the quantum query complexity of other problems? The complexity of the algorithm in the paper has rather bad dependence on k. Is it possible to improve the dependence using a more advanced fault-tolerance technique? Finally, we know that the Ambainis' algorithm can be implemented time-efficiently. Is this true for the algorithm in this paper?

Acknowledgments

I am grateful to Robin Kothari for sharing his construction of learning graphs with different arc weights for different values of the variable being loaded and to Andris Ambainis for sharing his algorithm for the graph collision problem that has mostly triggered this research. Also, I would like to thank Andris Ambainis and the anonymous referees for the significant help in improving the presentation of the paper.

This work has been supported by the European Social Fund within the project "Support for Doctoral Studies at University of Latvia" and by FET-Open project QCS.

References

[1] A. Ambainis, "Quantum walk algorithm for element distinctness," *SIAM Journal on Computing*, vol. 37, pp. 210–239, 2007.

[2] S. Aaronson and Y. Shi, "Quantum lower bounds for the collision and the element distinctness problems," *Journal of the ACM*, vol. 51, no. 4, pp. 595–605, 2004.

[3] S. Kutin, "Quantum lower bound for the collision problem with small range," *Theory of Computing*, vol. 1, no. 1, pp. 29–36, 2005.

[4] A. Ambainis, "Polynomial degree and lower bounds in quantum complexity: Collision and element distinctness with small range," *Theory of Computing*, vol. 1, pp. 37–46, 2005.

[5] F. Magniez, M. Santha, and M. Szegedy, "Quantum algorithms for the triangle problem," *SIAM Journal on Computing*, vol. 37, no. 2, pp. 413–424, 2007.

[6] H. Buhrman and R. Špalek, "Quantum verification of matrix products," in *Proc. of 17th ACM-SIAM SODA*, 2006, pp. 880–889.

[7] S. Dörn and T. Thierauf, "The quantum query complexity of algebraic properties," in *Proc. of 16th FCT*, vol. 4639. Springer-Verlag, 2007, pp. 250–260.

[8] M. Szegedy, "Quantum speed-up of markov chain based algorithms," in *Proc. of 45th IEEE FOCS*, 2004, pp. 32–41.

[9] F. Magniez, A. Nayak, J. Roland, and M. Santha, "Search via quantum walk," in *Proc. of 39th ACM STOC*, 2007, pp. 575–584.

[10] A. Childs and J. Eisenberg, "Quantum algorithms for subset finding," *Quantum Information & Computation*, vol. 5, no. 7, pp. 593–604, 2005.

[11] A. Belovs and R. Špalek, "Adversary lower bound for the k-sum problem," 2012, arXiv:1206.6528.

[12] A. Belovs, "Span programs for functions with constant-sized 1-certificates," in *Proc. of 44th ACM STOC*, 2012, pp. 77–84.

[13] Y. Zhu, "Quantum query complexity of subgraph containment with constant-sized certificates," 2011, arXiv:1109.4165.

[14] T. Lee, F. Magniez, and M. Santha, "A learning graph based quantum query algorithm for finding constant-size subgraphs," 2011, arXiv:1109.5135.

[15] A. Belovs and T. Lee, "Quantum algorithm for k-distinctness with prior knowledge on the input," 2011, arXiv:1108.3022.

[16] A. Belovs and B. Reichardt, "Span programs and quantum algorithms for st-connectivity and claw detection," 2012, arXiv:1203.2603.

[17] H. Buhrman and R. de Wolf, "Complexity measures and decision tree complexity: a survey," *Theoretical Computer Science*, vol. 288, pp. 21–43, 2002.

[18] A. Ambainis, "Quantum lower bounds by quantum arguments," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 750–767, 2002.

[19] P. Høyer, T. Lee, and R. Špalek, "Negative weights make adversaries stronger," in *Proc. of 39th ACM STOC*, 2007, pp. 526–535.

[20] B. Reichardt, "Reflections for quantum query algorithms," in *Proc. of 22nd ACM-SIAM SODA*, 2011, pp. 560–569.

[21] T. Lee, R. Mittal, B. Reichardt, R. Špalek, and M. Szegedy, "Quantum query complexity of the state conversion problem," in *Proc. of 52nd IEEE FOCS*, 2011, pp. 344–353.

[22] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, "Tight bounds on quantum searching," *Fortschritte der Physik*, vol. 46, no. 4-5, pp. 493–505, 1998.