Simplified Quantum Algorithm for the Oracle Identification Problem

Leila Taghavi
QuOne Lab, Phanous Research and Innovation Centre, Tehran, Iran

In the oracle identification problem we have oracle access to bits of an unknown string $x$ of length $n$, with the promise that it belongs to a known set $C \subseteq \{0,1\}^n$. The goal is to identify $x$ using as few queries to the oracle as possible. We develop a quantum query algorithm for this problem with query complexity $O\left(\sqrt{\frac{n \log M}{\log(n/\log M) + T}}\right)$, where $M$ is the size of $C$. This bound is already derived by Kothari in 2014, for which we provide a more elegant simpler proof.

I. INTRODUCTION

In the oracle identification problem, we have query access to bits of an unknown $n$-bit string $x$, with the promise that it belongs to a known set $C \subseteq \{0,1\}^n$ of size $M$. We want to determine $x$ while minimizing the number of queries to bits of $x$. We denote this problem by OIP($C$). Here, we are interested in quantum algorithms for this problem in which case we assume we can query bits of $x$ in superposition.

We are interested in the query complexity of OIP($C$) in the worst case with the promise that $|C| = M$. To this end, for a given $M$ and $n$, we say that $Q(OIP(M,n)) = q$ if there exists a bounded error quantum query algorithm that solves any OIP($C$) where $|C| = M$ with at most $q$ queries. We note that for some sets $C$ the quantum query complexity might be less than $Q(OIP(M,n))$, yet here we consider the worst case query complexity over such choices of $C$.

Characterization of $Q(OIP(M,n))$ for $M \leq n$ is easy; by reducing the problem to Grover’s search on $M$ elements we find that $Q(OIP(M,n)) = O(\sqrt{M})$. For the hard case of $n < M \leq 2^n$, Kothari [5] characterized the quantum query complexity of OIP($M,n$) by proposing an algorithm beyond a simple Grover’s search. This algorithm is based on ideas from classical learning theory in combination with a composition property of the so called filtered $\gamma_2$-norm. Here, we give a simpler direct proof of the same result that eliminates the need for the filtered $\gamma_2$-norm. Our proof is based on the framework of [2] that converts a classical algorithm into an improved quantum one.

Assume that we have a classical algorithm that computes a function $f : [\ell]^n \rightarrow [m]$ with query complexity $T$. Moreover, assume that we have a guessing algorithm that tries to predict values of queried bits, making at most $G$ mistakes. Using these classical algorithms, we can design a quantum query algorithm for computing $f$ with query complexity $O(\sqrt{GT})$. This result in [2] is proven based on the framework of non-binary span programs (NBSP) [1], which is a generalization of the span programs [9] for functions with non-binary input and/or output alphabets. In this paper, once again using the framework of NBSPs we prove a generalization of the aforementioned result of [2]. Next, using that generalization we give a simple proof of the following bound on $Q(OIP(M,n))$.

Theorem 1. Suppose that we are given a set $C \subseteq \{0,1\}^n$, where $|C| = M$ and $n < M \leq 2^n$. Also suppose that we have query access to bits of a string $x \in \{0,1\}^n$ with the promise that $x \in C$. Then quantum query complexity of identifying $x$ is $O\left(\sqrt{\frac{n \log M}{\log(n/\log M) + T}}\right)$.

Note that the bound of this theorem is tight; it is shown by Kothari [5] that there exists a set $C$ of size $M$ for which $Q(OIP(C)) = \Omega\left(\sqrt{\frac{n \log M}{\log(n/\log M) + T}}\right)$.

II. CLASSICAL TO QUANTUM QUERY ALGORITHM

A classical query algorithm can be modeled using a decision tree. A decision tree is a directed acyclic graph which depicts the sequence of queried bits along the algorithm. Any internal node of this tree is labeled by an index $i \in [n]$, and output edges of this node are labeled using possible outcomes of the query of $x_i$. Furthermore, every leaf of the decision tree is labeled by an output value of the algorithm. Let $P_x$ be the unique path from the root to a leaf, for which the labels of edges matches the bits of $x$.

* E-mail me at: l.taghavi@gmail.com
Theorem 2. Assume that we have a decision tree $T$ for a function $f : D_f \rightarrow \{m\}$ with $D_f \subseteq \{0,1\}^n$ whose depth is $T$. Furthermore, assume that for a $G$-coloring of the edges of $T$, the number of red edges in each path from the root to leaves of $T$ is at most $G$. Let $G_x$ be the number of red edges in $P_x$, and for $1 \leq g \leq G_x$, let $T_{g,x}$ be the number of black edges in $P_x$ after the $g$-th red edge and before the next red one. Also let $T_{0,x}$ be the number of black edges before the first red edge in $P_x$. Then there exists a bounded error quantum query algorithm that computes the function $f$ with query complexity

$$O \left( \max_x \sum_{g=0}^{G_x} \sqrt{T_{g,x}} \right).$$

Before getting into the proof of this theorem, let us make an intuition about its statement. Think of the guessing algorithm as a reference that answers to our queries. We look for its mistakes and truncate the classical algorithm into $G + 1$ parts, where $G$ is the number of mistakes that the guessing algorithm makes. In each of these truncated pieces, the guessing algorithm predicts all the necessary queries correctly, so in our quantum algorithm we can follow the classical one without making any queries. To count the number of queries, note that there exists a variant of Grover search algorithm that finds the first marked element in a list of $n$ elements making $O(\sqrt{j})$ queries, where $j$ is the index of this marked element [5]. Using this result we can find the $i$th mistake of the guessing algorithm using $O(\sqrt{T_{i-1,x}})$ queries. The total number of queries will then be $\sum_{g=0}^{G_x} \sqrt{T_{g,x}} \log n$, where the extra log factor is needed for error reduction [7]. In our proof we design a non-binary span program for this problem that mimics the behavior of the stated algorithm without getting this extra log $n$ factor in its complexity.

To present this span program first we need to develop some notations. Let $V(T)$ be the vertex set of $T$. Then for every internal vertex $v \in V(T)$, its associated index is denoted by $J(v)$, i.e., $J(v)$ is the index $1 \leq j \leq n$ that is queried by the classical algorithm at node $v$. The two outgoing edges of $v$ are indexed by elements of $\{0,1\}$ and connect $v$ to two other vertices. We denote these vertices by $N(v,0)$ and $N(v,1)$. That is, $N(v,q)$, for $q \in \{0,1\}$, is

![Decision Tree Diagram](image)

**FIG. 1:** A decision tree for a function that outputs 1 if the input string contains at least two 0s. Any leaf of the tree is labeled with its associated output, and other vertices are labeled with the index that the classical algorithm queries in that vertex. A $G$-coloring is represented by the edge coloring of the decision tree using two colors black and red. This guessing algorithm always guesses that the output of the queried bit is 1 (black edges). Edges with label 0 are colored red and indicate wrong guesses of the guessing algorithm. The depth of this tree is $T = n$ which is the classical query complexity of this algorithm for this function. The maximum number of wrong guesses of the guessing algorithm is $G = 2$ since after seeing the first two mistakes (red edges) the output of the function will be revealed.

We consider *guessing algorithms* along with decision trees. A guessing algorithm is a classical algorithm that predicts the values of queries. Such an algorithm can be represented by an edge-coloring of the decision tree. In this coloring we color any edge associated to an output of the guessing algorithm black and color the rest of edges red. We call such a coloring of edges of a decision tree a $G$-coloring. More formally, a G-coloring is a coloring of the edges of the decision tree using two colors black and red in such a way that every node has exactly one outgoing edge with black color. See Figure 1 for an example of a decision tree and its G-coloring.

To prove our main result, here we first give a bound on the quantum query complexity of a problem based on a decision tree and a G-coloring on it. This result is proven based on ideas from [2].

**Theorem 2.** Assume that we have a decision tree $T$ for a function $f : D_f \rightarrow \{m\}$ with $D_f \subseteq \{0,1\}^n$ whose depth is $T$. Furthermore, assume that for a $G$-coloring of the edges of $T$, the number of red edges in each path from the root to leaves of $T$ is at most $G$. Let $G_x$ be the number of red edges in $P_x$, and for $1 \leq g \leq G_x$, let $T_{g,x}$ be the number of black edges in $P_x$ after the $g$-th red edge and before the next red one. Also let $T_{0,x}$ be the number of black edges before the first red edge in $P_x$. Then there exists a bounded error quantum query algorithm that computes the function $f$ with query complexity

$$O \left( \max_x \sum_{g=0}^{G_x} \sqrt{T_{g,x}} \right).$$

Before getting into the proof of this theorem, let us make an intuition about its statement. Think of the guessing algorithm as a reference that answers to our queries. We look for its mistakes and truncate the classical algorithm into $G + 1$ parts, where $G$ is the number of mistakes that the guessing algorithm makes. In each of these truncated pieces, the guessing algorithm predicts all the necessary queries correctly, so in our quantum algorithm we can follow the classical one without making any queries. To count the number of queries, note that there exists a variant of Grover search algorithm that finds the first marked element in a list of $n$ elements making $O(\sqrt{j})$ queries, where $j$ is the index of this marked element [5]. Using this result we can find the $i$th mistake of the guessing algorithm using $O(\sqrt{T_{i-1,x}})$ queries. The total number of queries will then be $\sum_{g=0}^{G_x} \sqrt{T_{g,x}} \log n$, where the extra log factor is needed for error reduction [7]. In our proof we design a non-binary span program for this problem that mimics the behavior of the stated algorithm without getting this extra log $n$ factor in its complexity.

To present this span program first we need to develop some notations. Let $V(T)$ be the vertex set of $T$. Then for every internal vertex $v \in V(T)$, its associated index is denoted by $J(v)$, i.e., $J(v)$ is the index $1 \leq j \leq n$ that is queried by the classical algorithm at node $v$. The two outgoing edges of $v$ are indexed by elements of $\{0,1\}$ and connect $v$ to two other vertices. We denote these vertices by $N(v,0)$ and $N(v,1)$. That is, $N(v,q)$, for $q \in \{0,1\}$, is
the next vertex that is reached from $v$ after following the outgoing edge with label $q$. We also represent the G-coloring of edges of $T$ by a function $C(v, q) \in \{\text{black, red}\}$ where $v$ is an internal vertex, $q \in \{0, 1\}$. $C(v, q)$ is the color of the outgoing edge of $v$ with label $q$. Also let $b(v)$ be the number of black edges after the last red edge in the path from the root to the vertex $v$. These notations are depicted in an example in Figure 2.

**Proof.** For every $x \in D_f$ there is an associated leaf of the tree $T$ that is reached once we follow edges of the tree with labels $x_j$, starting from the root. In order to find $f(x)$ it suffices to find this associated leaf because this is what the classical query algorithm does; once we find the leaf associated to $x$, we find the path that the classical query algorithm would take and then find $f(x)$. Thus in order to compute $f$, we may compute another function $\bar{f}$ which given $x$ outputs its associated leaf of $T$, and to prove an upper bound on the quantum query complexity, it suffices to design an algorithm for $\bar{f}$.

We use the framework of non-binary span programs (NBSPs) for bounding the quantum query complexity of $\bar{f}$. For more details on NBSPs we refer to [1] and here we only present the ingredients of an NBSP through the example of our particular function $\bar{f}$:

- The first ingredient of an NBSP is a finite-dimensional vector space $\mathcal{V}$ that is called the input space. Here, in our problem the input space is determined by the orthonormal basis indexed by vertices of the decision tree $T$:

\[
\{ |v\rangle \mid v \in V(T) \},
\]  

(2)

- An NBSP contains some target vectors $|t_0\rangle, |t_2\rangle, \ldots, |t_{m-1}\rangle \in \mathcal{V}$, one for any possible value of the function. Here, the output values of $\bar{f}$ are indexed by leaves $u$ of $T$ and we let

\[
|t_u\rangle = |r\rangle - |u\rangle,
\]

(3)

where $r \in V(T)$ is the root of the tree.

- The input vectors $I_{j, q}$ of an NBSP are some subsets $I_{j, q} \subseteq \mathcal{V}$ for every $1 \leq j \leq n$ and $q \in \{0, 1\}$. Here, the input sets are defined by

\[
I_{j, q} = \left\{ \sqrt{W_{C(v, q), b(v)}(|v\rangle - |N(v, q)\rangle)} \right\} |v\rangle \in V(T) \text{ s.t. } J(v) = j,
\]

(4)

where $W_{\text{black, } b}$ and $W_{\text{red, } b}$ are positive real numbers to be determined [3].

Having all these ingredients, we let $I \subseteq \mathcal{V}$ be

\[
I = \bigcup_{j=1}^{n} \bigcup_{q\in|t|} I_{j, q},
\]

(5)
and for every $x \in D_f$ we define the set of available vectors $I(x)$ by

$$I(x) = \bigcup_{j=1}^{n} I_j, x_j.$$  

(6)

Now we say that the above NBSP evaluates $\tilde{f}$ if for every $x$ the target vector $|t_\alpha\rangle$ belongs to the span of available vectors $I(x)$ if and only if $\alpha = \tilde{f}(x)$. Furthermore, there should be negative and positive witnesses for this, that are explained below.

For every vertex $v$ of $T$, let $P_v$ be the (unique) path from the root $r$ to the vertex $v$. Then, for every $x \in D_f$ there exists a path $P_x = P_{f(x)}$ from the root of the decision tree to the leaf $\tilde{f}(x)$.

Thus, the target vector $|t_{\tilde{f}(x)}\rangle$ equals

$$|t_{\tilde{f}(x)}\rangle = |r\rangle - |\tilde{f}(x)\rangle = \sum_{v \in P_x} \frac{1}{\sqrt{W_C(v, x_j(i)), b(v)}} \left\{ \sqrt{W_C(v, x_j(i)), b(v)} \langle v \rangle - \langle N(v, x_j(i)) \rangle \right\},$$

(7)

where the vectors in the braces are all available for $x$. This shows that $|t_{\tilde{f}(x)}\rangle$ is the only target vector belonging to the span of $I(x)$. We show that there exists a negative witness for it, defined by

$$|\tilde{w}_x\rangle = \sum_{v \in P_x} |v\rangle \in V.$$  

(8)

We note that that $|\tilde{w}_x\rangle$ is orthogonal to all available vectors in $I(x)$. Moreover, it satisfies $\langle \tilde{w}_x | t_\alpha\rangle = \langle \tilde{w}_x | r\rangle = 1$ for every $\alpha \neq \tilde{f}(x)$. These two facts ensure that $|t_\alpha\rangle$ for $\alpha \neq \tilde{f}(x)$ does not belong to the span of $I(x)$.

Now based on the results of $\mathbb{H}$ the above NBSP gives a bound on the query complexity of $\tilde{f}$. To evaluate this bound we need to estimate two quantities called the positive complexity and negative complexity of the NBSP. The positive complexity denoted by $\text{ws}^+(P, w, \tilde{w})$ is the maximum of the squared norm of the coefficient-vector in the expansion (7). For any $x$ this norm is computed as

$$\text{ws}^+(x) = \sum_{g=0}^{G_x} \left( \frac{1}{W_{\text{red},T_g,x}} + \sum_{b=0}^{T_{g,x}-1} \frac{1}{W_{\text{black},b}} \right).$$

(9)

The negative complexity for any $x$ denoted by $\text{ws}^-(x)$ is equal to the sum of the squared inner product of $|\tilde{w}_x\rangle$ with all vectors in $I$. To compute this quantity we need to compute the overlap of $|\tilde{w}_x\rangle$ with all vectors of the form

$$\sqrt{W_C(v, q), b(v)} \left( |v\rangle - |N(v, q)\rangle \right).$$

(10)

We note that such a vector contributes in the negative complexity (is not orthogonal to $|\tilde{w}_x\rangle$) only if its corresponding edge $\{v, N(v, q)\}$ leaves the path $P_x$, i.e., they have only the vertex $v$ in common. In this case, the contribution would be equal to $W_C(v, q), b(v)$, which is the weight of that edge. Therefore, we have

$$\text{ws}^-(x) = \sum_{g=0}^{G_x} \left( \frac{1}{W_{\text{black},T_g,x}} + \sum_{b=0}^{T_{g,x}-1} \frac{1}{W_{\text{red},b}} \right).$$

(11)

Now letting $W_{\text{red},b} = 1/W_{\text{black},b} = \sqrt{b+1} - \sqrt{b}$, both the positive and negative witnesses are bounded by

$$\sum_{g=0}^{G_x} \left( \frac{1}{\sqrt{T_{g,x}} + 1 - \sqrt{T_{g,x}}} + \sum_{b=0}^{T_{g,x}-1} \sqrt{b+1} - \sqrt{b} \right) = O \left( \sum_{g=0}^{G_x} \sqrt{T_{g,x}} \right),$$

(12)

where we used the fact that

$$\left( \sqrt{T_{g,x}} + 1 - \sqrt{T_{g,x}} \right)^{-1} = \left( \frac{T_{g,x} + 1}{T_{g,x}} - 1 \right)^{-1} = O \left( \frac{1}{\sqrt{T_{g,x}}} \right),$$

(13)

$$O \left( \frac{1}{\sqrt{T_{g,x}}} \right).$$

(14)
Putting these together the complexity of the NBSP is defined and bounded by
\[
\text{wsize} = \sqrt{\max_x \text{wsize}^+(x) \cdot \max_x \text{wsize}^-(x)} = O\left(\max_x \sum_{g=0}^{G_x} \sqrt{T_{g,x}}\right).
\] (15)

This is also a bound on the quantum query complexity of \(\tilde{f}\), and that of \(f\).

\[\square\]

**III. PROOF OF THE MAIN RESULT**

Recall that in the oracle identification problem we are given a set \(C \subseteq \{0,1\}^n\), where \(|C| = M\), and query access to a string \(x \in \{0,1\}^n\), with the promise that \(x \in C\). We want to identify the input \(x\) using the least number of queries to the input oracle. We use Theorem 2 to design a quantum query algorithm for this problem. To apply this theorem we need a classical algorithm and a guessing algorithm, for which we borrow ideas from [5]. The classical algorithm chooses the order in which we query bits of \(x\). For any set \(C\), we need to compute the maximum of this quantity over all possible choices of \(\pi\)

\[\text{Proof of Theorem 1}\]

The classical algorithm for this problem is as follows. We first apply Lemma 3 to the set \(C^{(1)} = C\) and obtain a string \(s^{(1)}\) and a permutation \(\pi_1\). We then query the indices one by one using the order defined by the permutation \(\pi_1\) (the \(i\)-th query is the index \(\pi_1(i)\)). The guessing algorithm predicts the value of this query to be \(s^{(1)}(\pi_1(i))\). We continue to query until we see a mismatch between a queried value and the corresponding bit of \(s^{(1)}\), that corresponds to a wrong guess. Suppose that the first mismatch is in the \(p_1\)-th query, so we have \(x^{(1)}_{\pi_1(p_1)} \neq s^{(1)}_{\pi_1(p_1)}\).

It is now guaranteed that \(x \in C^{(1)}_{p_1}\) and we can eliminate at least a fraction \(1 - \frac{1}{\max\{2,p_1\}}\) of members of \(C^{(1)}\) from possible candidates for \(x\). We then apply Lemma 3 once again to the set \(C^{(2)} = C^{(1)}_{p_1}\) and obtain a new string \(s^{(2)}\) and a new permutation \(\pi_2\). We then continue to query using the order defined by \(\pi_2\) until we find a mismatch between a queried value and the string \(s^{(2)}\). This again eliminates some members of \(C^{(2)}\). We continue this process until we reach some set \(C^{(r)}\) with \(|C^{(r)}| = 1\) and output the only member of \(C^{(r)}\) as the value of \(x\).

In the above algorithm, the number of wrong guesses (red edges observed in the decision tree) is \(r\). Moreover, using Theorem 2, the quantum query complexity of this problem is \(\sum_{i=1}^{r-1} \sqrt{p_i}\). Indeed, to find an estimate on the quantum query complexity, we need to compute the maximum of this quantity over all possible choices of \(r\) and \(p_i\)’s. To this end, note that the total number of black edges in any path from the root to a leaf is at most \(n\) and

\[
\sum_{i=1}^{r-1} p_i \leq n.
\] (17)

We also using Lemma 3 know that the \(i\)-th wrong guess reduces the size of \(C^{(i)}\) by a factor of \(\max\{2,p_i\}\). Therefore, we have

\[
\prod_{i=1}^{r-1} \max\{2,p_i\} \leq M.
\] (18)

Thus, using Theorem 2 the quantum query complexity of this problem is the maximum value of

\[
\sum_{i=1}^{r-1} \sqrt{p_i},
\] (19)
FIG. 3: An example for the decision tree and its coloring based on the algorithm of Theorem 1. In this example we assume that the input set $C$ is equal to $\{0000, 0001, 0011, 0111, 1111\}$. In the left we see the updates of the set $C$ after each query, the order in which we query is determined using Lemma 3. For any update of the set $C$, the queried index which is the most informative bit based on lemma 3 is colored red. In the right the associated decision tree is depicted. Note that any colored node of the decision tree is associated to the set $C$ in the left with the same color.

subject to the constraints (17) and (18). Kothari showed that the optimal value of this optimization problem is bounded by $O\left(\sqrt{\frac{n \log M}{\log(n/ \log M) + 1}}\right)$. The proof is by finding the optimal value of another linear program that upper-bounds the optimal value of this optimization problem and is easier to solve (see Appendix B of the full version of [5] for more details).

Putting these together we conclude that the quantum query complexity of the oracle identification problem is $O\left(\sqrt{\frac{n \log M}{\log(n/ \log M) + 1}}\right)$.

As an example assume that $C = \{0000, 0001, 0011, 0111, 1111\}$. Figure 3 shows how using Theorem 1 we can discover any given input $x \in C$. In the left we see possible updates in the set $C$ along the algorithm based on queries. In the right the associated decision tree is depicted. Note that similar colors in the left picture and the decision tree determines the same stage of the algorithm, for example in the root of the decision tree which we show using color blue, the set $C$ is the leftmost set in the left picture which has blue color too. We also color the index that we query in each step of the algorithm by red color in $C$. Vertices with green color are the leaves of the decision tree, which we label by an input $x$.

To conclude, note that the classical algorithm that we used here is based on Khotari’s [5]. This algorithm is based on repeated applications of the Grover search algorithm to find the mismatches between $x$ and $s^{(i)}$ and then uses properties of the $\gamma_2$-norm to eliminate the log factor that has been added for error reduction. Using our approach this log factor does not appear in the query complexity in the first place. This can be generalized to any problem of this form, i.e., for any algorithm with repeated calls to Grover search we can use the same idea to eliminate the log factor we get for error reduction.
a. **Acknowledgements**  I would like to thank Salman Beigi for his helpful discussion and suggestions.

[1] Salman Beigi and Leila Taghavi. Span Program for Non-binary Functions. *Quantum Information and Computation (QIC)*, 19(9&10):0760–0792, 2019.

[2] Salman Beigi and Leila Taghavi. Quantum Speedup Based on Classical Decision Trees. *Quantum*, 4:241, March 2020.

[3] Andrew M. Childs, Robin Kothari, Maris Ozols, and Martin Roetteler. Easy and hard functions for the boolean hidden shift problem. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 22, pages 50–79, 2013.

[4] Tibor Hegedûs. Generalized teaching dimensions and the query complexity of learning. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 108–117, 1995.

[5] Robin Kothari. An optimal quantum algorithm for the oracle identification problem. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 25, pages 482–493, 2014.

[6] Cedric Yen-Yu Lin and Han-Hsuan Lin. Upper bounds on quantum query complexity inspired by the Elitzur-Vaidman bomb tester. *Theory of Computing*, 12:1–35, 2016.

[7] The Grover search algorithm is a bounded error algorithm meaning that the probability of getting a correct answer is at least $\frac{2}{3}$. By successive calls of this algorithm on different sets, the errors in different outputs aggregate and the probability of getting the correct final answer becomes increasingly small. To reduce this error we can repeat each Grover call $\log n$ times and get the majority vote. This makes the total error bounded with the cost of a $\log n$ factor.

[8] Note that the weight of every edge $(v, q)$ not only depends on its color but also depends on $b(v)$, that is the number of black edges after the last red edge in the path from root to $v$.

[9] Ben W Reichardt. Span programs and quantum query complexity: The general adversary bound is nearly tight for every boolean function. In *Foundations of Computer Science, 2009. FOCS’09. 50th Annual IEEE Symposium on*, pages 544–551. IEEE, 2009.