Quantum Adversary (Upper) Bound[*]

Shelby Kimmel[†]

Abstract: We describe a method to upper bound the quantum query complexity of Boolean formula evaluation problems, using fundamental theorems about the general adversary bound. This nonconstructive method can give an upper bound on query complexity without producing an algorithm. For example, we describe an oracle problem which we prove (non-constructively) can be solved in $O(1)$ queries, where the previous best quantum algorithm uses a polylogarithmic number of queries. We then give an explicit $O(1)$-query algorithm for this problem based on span programs.

1 Introduction

The general adversary bound has proven to be a powerful concept in quantum computing. Originally formulated as a lower bound on the quantum query complexity of Boolean functions [6], it was proven to be a tight bound both for the query complexity of evaluating discrete finite functions and for the query complexity of the more general problem of state conversion [8]. The general adversary bound is the culmination of a series of adversary methods [1, 2]. While the adversary method in its various forms has been useful in finding lower bounds on quantum query complexity [3, 7, 11], the general adversary bound itself can be difficult to apply, as the quantity for even simple, few-bit functions must usually be calculated numerically [6, 11].

One of the nicest properties of the general adversary bound is that it behaves well under composition [8]. This fact has been used to lower bound the query complexity of evaluating composed total functions,

---

Key words and phrases: quantum algorithms, quantum query complexity, adversary bound, nand tree

---

and to create optimal algorithms for composed total functions [11]. Here, we extend one of the composition results to partial Boolean functions, and use it to upper bound the query complexity of Boolean functions. We do this by obtaining an *upper* bound on the general adversary bound.

Generally, finding an upper bound on the general adversary bound is just as difficult as finding an algorithm, as they are dual problems [8]. However, using the composition property of the general adversary bound, when given an algorithm for a Boolean function f composed d times, we obtain an upper bound on the general adversary bound of f. Due to the tightness of the general adversary bound and query complexity, this procedure gives an upper bound on the query complexity of f, but because it is nonconstructive, it doesn't give any hint as to what the corresponding algorithm for f might look like. The procedure is a bit counter-intuitive: we obtain information about an algorithm for a simpler function by creating an algorithm for a more complex function. This is similar in spirit to the tensor power trick, where an inequality between two terms is proven by considering tensor powers of those terms[1].

We describe a class of oracle problems called CONSTANT-FAULT DIRECT TREES (introduced by Zhan et al. [13]), for which this method proves the existence of an $O(1)$ query algorithm. While this method does not give an explicit algorithm, we show that a span program algorithm achieves this bound. The previous best algorithm for CONSTANT-FAULT DIRECT TREES has a query complexity that is polylogarithmic in the size of the problem.

In Section 2 we describe the upper bound on the general adversary bound. In Section 3 we apply this bound to CONSTANT-FAULT DIRECT TREES and prove the existence of a constant query algorithm. In Section 4 we describe the span program based quantum algorithm for CONSTANT-FAULT DIRECT TREES.

2   A Nonconstructive Upper Bound on Query Complexity

Our procedure for creating a nonconstructive upper bound on query complexity relies on the fact that the general adversary bound behaves well under composition and is a tight lower bound on quantum query complexity. The standard definition of the general adversary bound is not necessary for our purposes, but can be found in [7], and an alternate definition appears in Appendix A.

Our procedure applies to Boolean functions. A function f is Boolean if $f : S \rightarrow \{0, 1\}$ with $S \subseteq \{0, 1\}^n$. Given a Boolean function f and a natural number d, we define f^d, "f composed d times," recursively as $f^d = f \circ (f^{d-1}, \dots, f^{d-1})$, where $f^1 = f$.

Now we state the main result:

Theorem 2.1. *Suppose we have a (possibly partial) Boolean function f that is composed d times, f^d, and a quantum algorithm for f^d that requires $O(J^d)$ queries. Then $Q(f) = O(J)$, where $Q(f)$ is the bounded-error quantum query complexity of f.*

(For background on bounded-error quantum query complexity and quantum algorithms, see [1].) There are seemingly similar results in the literature; for example, Reichardt proves in [9] that the query complexity of a function composed d times, when raised to the $1/d^{th}$ power, is equal to the adversary bound of the

---

[1]See Terence Tao's blog, *What's New* "Tricks Wiki article: The tensor power trick," http://terrytao.wordpress.com/2008/08/25/tricks-wiki-article-the-tensor-product-trick/

function, in the limit that d goes to infinity. This result gives insight into the exact query complexity of a function, and its relation to the general adversary bound. In contrast, our result is a *tool* for upper bounding query complexity, possibly without gaining any knowledge of the exact query complexity of the function.

One might think that Theorem 2.1 is useless because an algorithm for f^d usually comes from composing an algorithm for f. If J is the query complexity of the algorithm for f, one expects the query complexity of the resulting algorithm for f^d to be at least J^d. In this case, Theorem 2.1 gives no new insight. Luckily for us, composed quantum algorithms do not always follow this scaling. If there is a quantum algorithm for f that uses J queries, where J is not optimal (i.e. is larger than the true bounded error quantum query complexity of f), then the number of queries used when the algorithm is composed d times can be much less than J^d. If this is the case, and if the non-optimal algorithm for f is the best known, Theorem 2.1 promises the existence of an algorithm for f that uses fewer queries than the best known algorithm, but, as Theorem 2.1 is nonconstructive, it gives no information as to the form of the algorithm.

We need two lemmas to prove Theorem 2.1:

Lemma 2.2. *For any Boolean function $f : S \to \{0,1\}$ with $S \subseteq \{0,1\}^n$ and natural number d,*

$$\mathrm{ADV}^{\pm}(f^d) \geq (\mathrm{ADV}^{\pm}(f))^d. \tag{2.1}$$

Høyer et al. [6] prove Lemma 2.2 for total Boolean functions[2], and the result is extended to more general total functions in [8]. Our contribution is to extend the result in [8] to partial Boolean functions. While Theorem 2.1 still holds for total functions, the example we consider later in the paper involves partial functions. The proof of Lemma 2.2 closely follows the proof in [8] and can be found in Appendix A.

Lemma 2.3. *(Lee, et al. [8]) For any function $f : S \to E$, with $S \in D^n$, and E, D finite sets, the bounded-error quantum query complexity of f, $Q(f)$, satisfies*

$$Q(f) = \Theta(\mathrm{ADV}^{\pm}(f)). \tag{2.2}$$

We now prove Theorem 2.1:

Proof. Given an algorithm for f^d that requires $O(J^d)$ queries, by Lemma 2.3,

$$\mathrm{ADV}^{\pm}(f^d) = O(J^d). \tag{2.3}$$

Combining Eq. (2.3) and Lemma 2.2,

$$(\mathrm{ADV}^{\pm}(f))^d = O(J^d). \tag{2.4}$$

Raising both sides to the $1/d^{th}$ power,

$$\mathrm{ADV}^{\pm}(f) = O(J). \tag{2.5}$$

---

[2]While the statement of Theorem 11 in [6] seems to apply to partial functions, it is mis-stated; their proof actually assumes total functions.

We now have an upper bound on the general adversary bound of f. Finally, using Lemma 2.3 again, we obtain

$$Q(f) = O(J). \tag{2.6}$$

□

3 Example where the General Adversary Upper Bound is Useful

In this section we describe a function, called the 1-FAULT NAND TREE, for which Theorem 2.1 gives a better upper bound on query complexity than any previously known quantum algorithm. The 1-FAULT NAND TREE was proposed by Zhan et al. [13] to obtain a super-polynomial speed-up for a partial Boolean formula, and is a specific type of CONSTANT-FAULT DIRECT TREE, which was mentioned in Section 1. We first define the NAND TREE, and then explain the allowed inputs to the 1-FAULT NAND TREE.

The NAND TREE is a complete, binary tree of depth d, where each node is assigned a bit value. The leaves are assigned arbitrary values, and any internal node v is given the value $\mathrm{NAND}(val(v_1), val(v_2))$, where v_1 and v_2 are v's children, and $val(v_i)$ is the value of the node v_i.

To evaluate the NAND TREE, one must find the value of the root given an oracle for the values of the leaves. (The NAND TREE is equivalent to solving NAND^d, although the composition we will use for Theorem 2.1 is not the composition of the NAND function, but of the NAND TREE as a whole.) For arbitrary inputs, Farhi et al. showed that there exists an optimal quantum algorithm in the Hamiltonian model to solve the NAND TREE in $O(2^{0.5d})$ time [5], and this was extended to a standard discrete algorithm with quantum query complexity $O(2^{0.5d})$ [4, 10]. Classically, the best algorithm requires $\Omega(2^{0.753d})$ queries [12]. Here, we consider the 1-FAULT NAND TREE, which is a NAND TREE with a promise that the inputs satisfy certain conditions.

Definition 3.1. *(1-FAULT NAND TREE [13])* Consider a NAND TREE of depth d (as described above). Then to each node v, we assign an integer $\kappa(v)$ such that:

- $\kappa(v) = 0$ for leaf nodes.

- Otherwise v has children v_1 and v_2

    If $val(v_1) = val(v_2)$, $\kappa(v) = \max_{i \in \{1,2\}} \kappa(v_i)$,

    If $val(v_1) \neq val(v_2)$, let v_i be the node such that $val(v_i) = 0$. Then $\kappa(v) = 1 + \kappa(v_i)$.

A tree satisfies the 1-fault condition if $\kappa(v) \leq 1$ for any node v in the tree.

Notation: When a node has one child with value 1 and one child with value 0 ($val(v_1) \neq val(v_2)$), we call the node v a **fault**. (Since $\mathrm{NAND}(0,1) = \mathrm{NAND}(1,0) = 1$, fault nodes must have value 1, although not all 1-valued nodes are faults.)

The 1-fault condition is a limit on the amount and location of faults within the tree. In a 1-FAULT NAND TREE, if a path moving from a root to a leaf encounters any fault node and then passes through the 0-valued child of the fault node, there can be no further fault nodes on the path. An example of a 1-FAULT NAND TREE is given in Figure 1.

The condition of the 1-FAULT NAND TREE may seem strange, but it has a nice interpretation when considering the correspondence between NAND TREES and game trees[3]. The 1-FAULT NAND TREE corresponds to a game in which, if both players play optimally, there is at most one point in the sequence of play where a player's choice affects the outcome of the game. Furthermore, if a player makes the wrong choice at the decision point, the game again becomes a single-decision game, where if both players play optimally for the rest of play, there is at most one point where a player's choice affects the outcome of the game.
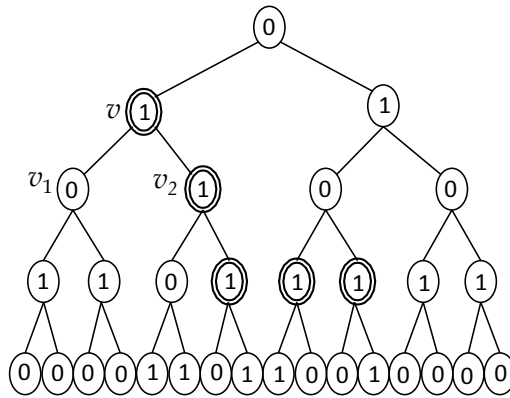

Figure 1: An example of a 1-FAULT NAND TREE of depth 4. Fault nodes are highlighted by a double circle. The node v is a fault since one of its children (v_1) has value 0, and one (v_2) has value 1. Among v_1 and its children, there are no further faults, as required by the 1-fault condition. There can be faults among v_2 and its children, and indeed, v_2 is a fault. There can be faults among the 1-valued child of v_2 and its children, but there can be no faults below the 0-valued child.

Zhan et al. [13] describe a quantum algorithm for the d-depth 1-FAULT NAND TREE that requires $O(d^2)$ queries to an oracle for the leaves. However, when the d-depth 1-FAULT NAND TREE is composed $\log d$ times, their algorithm requires only $O(d^3)$ queries. Here we see an example where the number of queries required by a composed algorithm does not scale exponentially in the number of compositions, which is critical for applying Theorem 2.1. Applying Theorem 2.1 to the algorithm for the 1-FAULT NAND TREE composed $\log d$ times, we find that an upper bound on the query complexity of the 1-FAULT NAND TREE is $O(1)$. This is a large improvement over $O(d^2)$ queries. Zhan et al. prove $\Omega(\text{poly} \log d)$ is a lower bound on the classical query complexity of 1-FAULT NAND TREES. An identical argument can be used to show that CONSTANT-FAULT NAND TREES (from Definition 3.1, trees satisfying $\kappa(v) \leq c$ with c a constant) have query complexity $O(1)$.

In fact, Zhan et al. find algorithms for a broad range of trees, where instead of NAND, the evaluation tree is composed of a type of Boolean function called a *direct* function. A direct function is a generalization of a monotonic Boolean function, and includes functions like majority, threshold, and their negations. For the exact definition, which involves span programs, see [13]. Similarly to CONSTANT-FAULT NAND

---

[3]See Scott Aaronson's blog, *Shtetl-Optimized*, "NAND now for something completely different," http://www.scottaaronson.com/blog/?p=207

TREES, Zhan et al. give a quantum algorithm for CONSTANT-FAULT DIRECT TREES requiring $O(d^2)$ queries and prove $\Omega(\text{poly}\log d)$ is a lower bound on the classical query complexity, while Theorem 2.1 can be used to prove the existence of $O(1)$-query algorithms for CONSTANT-FAULT DIRECT TREES.

4 Span Program Algorithm for CONSTANT-FAULT DIRECT TREES

The structure of CONSTANT-FAULT DIRECT TREES can be quite complex, and it is not obvious that there should be an $O(1)$-query algorithm. Inspired by the knowledge of the algorithm's existence, thanks to Theorem 2.1, we found a span program algorithm for CONSTANT-FAULT DIRECT TREES that requires $O(1)$ queries. It makes sense that the optimal algorithm uses span programs, not just because span programs can always be used to create optimal algorithms [8], but because Theorem 2.1 is based on properties of the general adversary bound, and there is strong duality between the general adversary bound and span programs.

Span programs are linear algebraic representations of Boolean functions, which have an intimate relationship with quantum algorithms. In particular, Reichardt proves [9] that given a span program P for a function f, there is a function of the span program called the witness size, such that one can create a quantum algorithm for f with query complexity $Q(f)$ satisfying

$$Q(f) = O(\text{WITNESS SIZE}(P)) \tag{4.1}$$

Thus, creating a span program for a function is equivalent to creating a quantum query algorithm.

There have been many iterations of span program-based quantum algorithms, due to Reichardt and others [8, 9, 11]. Zhan et al. create algorithms for direct Boolean functions [13] using the span program formulation described in Definition 2.1 in [9], one of the earliest versions (we will not go into the details of span programs in this paper). Using the more recent advancements in span program technology, we show here:

Theorem 4.1. *Given an evaluation tree composed of the direct Boolean function f, with the promise that the tree satisfies the k-fault condition (k a natural number), there is a quantum algorithm that evaluates the tree using $O(w^k)$ queries, where w is a constant that depends on f. In particular, for a* CONSTANT-FAULT DIRECT TREE *(k a constant), the algorithm requires $O(1)$ queries.*

While Theorem 2.1 promises the existence of $O(1)$-query quantum algorithms for CONSTANT-FAULT DIRECT TREES, Theorem 4.1 gives an explicit $O(1)$-query quantum algorithm for these problems. The proof combines properties of the witness size of direct Boolean functions with a more current version of span program algorithms.

First we define a k-FAULT DIRECT TREE, which is a Boolean evaluation tree made up of a direct Boolean function composed many times, with a promise on the input. The definition of direct Boolean functions are a bit technical and are given in [13]; here we will just use their properties. Given a direct Boolean function f with n inputs, a DIRECT TREE for f is a depth-d, n-partite complete graph, where each node is given a Boolean value. The value of the node v, $val(v)$ is given by $val(v) = f(val(v_1), \ldots, val(v_n))$ where v_i is the ith child node of v. The values of the leaves are given via an oracle, and the goal is to find the value of the root of the tree. A k-FAULT DIRECT TREE is a DIRECT TREE with inputs that satisfy certain conditions; the definition is similar to Definition 3.1 for 1-FAULT NAND TREES:

Definition 4.2. Let T be a DIRECT TREE for f. Let each node be labeled as *fault* or *trivial* based on the values of its children and the specific function f used. For each node, depending on the values of its children, a set of its child nodes are labeled *strong* in relation to the node, and the remaining child nodes are labeled *weak*. For trivial nodes, all children are strong. Then to each node v we assign an integer $\kappa(v)$ such that:

- $\kappa(v) = 0$ for leaf nodes.

- Otherwise v has children $\{v_1, \ldots, v_n\}$

  If v is trivial, then $\kappa(v) = \max_{i \in \{1,\ldots,n\}} \kappa(v_i) = \max_{i:v_i \text{ is strong}} \kappa(v_i)$.

  If v is a fault, then $\kappa(v) = 1 + \max_{i:v_i \text{ is strong}} \kappa(v_i)$.

A tree satisfies the k-fault condition if $\kappa(v_r) \leq k$ where v_r is the root.

Notice that the restriction $\kappa(v_r) \leq k$ is slightly relaxed compared to Definition 3.1, where it was required that $\kappa(v) \leq 1$ for all nodes v in the whole tree. Thus the span program algorithm we describe below applies to an even broader range of trees than were included in the discussion in Section 3.

For $f = \text{NAND}$, a node whose two children have the same value (either both 0 or both 1), is trivial, and a node with one 0-valued child and one 1-valued child is a fault. Furthermore, for $f = \text{NAND}$, for fault nodes, the 0-valued child is strong, and the 1-valued child is weak. One can verify that with these designations, Definition 3.1 corresponds to Definition 4.2 for $f = \text{NAND}$.

For a function $f : S \to \{0,1\}$, with $S \subseteq \{0,1\}^n$, input $x \in S$, and span program P, the weighted witness size on input x is $\text{WSIZE}_s(P,x)$ where $s \in (\mathbb{R}^+)^n$ is the weighting vector. When $s = (1,\ldots,1)$, we write $\text{WSIZE}_1(P,x)$. Following [9], we rewrite Eq. (4.1) as

$$Q(f) = O\left(\max_{x \in S} \text{WSIZE}_1(P,x)\right) \tag{4.2}$$

In [13], Zhan et al. show that for any direct Boolean function f, one can create a span program P with the following properties[4]:

- $\text{WSIZE}_1(P,x) = 1$ if input x makes the function trivial.

- $\text{WSIZE}_1(P,x) \leq w$, if input x makes the function a fault, where w is a constant depending only on f.

- For $\text{WSIZE}_s(P,x)$, s_j do not affect the witness size, where the jth input bit is weak.

To create an algorithm, we will combine these facts with Eq. (4.2) and the following composition lemma:

---

[4]We note that Zhan et al. use a different version of span programs than those used to prove Eq (4.2). However Reichardt shows in [9] how to transform from one span program formulation to another, and proves that there is a transformation from the span program formulation used by Zhan et al. to the one needed for Lemma 4.3 that does not increase the witness size and that uses the weighting vector in the same way.

Lemma 4.3. *(based on Theorem 4.3 in [9]) Let $f : S \to \{0,1\}$, $S \subseteq \{0,1\}^n$ and $g : C \to \{0,1\}$, $C \subseteq \{0,1\}^m$, and consider the composed function $(f \circ g)(x)$ with $x = (x^1, \ldots, x^n)$, $x^i \in C$ and $g(x^i) \in S \ \forall i$. Let $\tilde{x} = (g(x^1), \ldots, g(x^n)) \in S$. Let G be a span program for g, F be a span program for f, and $s \in (\mathbb{R}^+)^{n \times m}$. Then there exists a span program P for $f \circ g$ such that*

$$\text{WSIZE}_s(P,x) \leq \text{WSIZE}_r(F,\tilde{x}) \leq \text{WSIZE}_1(F,\tilde{x}) \max_{i \in [n]} \text{WSIZE}_{s^i}(G,x^i) \tag{4.3}$$

where $r = (\text{WSIZE}_{s^1}(G,x^1), \ldots, \text{WSIZE}_{s^n}(G,x^n))$ and s^i is a vector of the ith set of m elements of s.

The main difference between this lemma and that in [9] is that the witness size here is input dependent, as is needed for partial functions. We also use a single inner function g instead of n different functions g_i, but we allow each inner function to have a different input. The proof of this result follows exactly using the proof of Theorem 4.3 in [9], so we will not repeat it here.

Using the properties of strong and weak nodes in direct Boolean functions, we see that $\text{WSIZE}_r(F,\tilde{x})$ in (4.3) doesn't depend on $r_i = \text{WSIZE}_{s^i}(G,x^i)$ for weak inputs i. Thus, we can rewrite Eq. (4.3) as

$$\text{WSIZE}_s(P,x) \leq \text{WSIZE}_1(F,\tilde{x}) \max_{\substack{i \in [n] \\ \text{s.t. } i \text{ is strong}}} \text{WSIZE}_{s^i}(G,x^i). \tag{4.4}$$

We now prove Theorem 4.1:

Proof. For a direct function f, we know there is a span program P such that $\text{WSIZE}_1(P,x) \leq w$ for all fault inputs and $\text{WSIZE}_1(P,x) = 1$ for trivial inputs. We will show that this implies the existence of a span program for the K-FAULT DIRECT TREE with witness size $\leq w^k$.

We will use an inductive proof on k, the number of faults. For the base case, consider a depth 1 tree. This is just a single direct Boolean function. If its input makes the function a fault, using the properties of direct Boolean functions, there is a span program for this input with witness size at most w. If the depth 1 tree has has an input that makes the function trivial, there is a span program for this input with witness size at most 1. Thus there exists quantum algorithm with query complexity $O(1)$ that evaluates this tree.

Consider a depth d, k-fault tree T with input x. We can think of this instead as a single direct function f (with input $\tilde{x}$ and span program P), composed with n subtrees of depth $d-1$, where we label the ith subtree T^i. Let P_{T^i} be a span program for T^i, and we call the input to that subtree x^i. If $\tilde{x}$ makes f a fault, then by Eq (4.4) we know there exists a span program P_T for T such that:

$$\text{WSIZE}_1(P_T,x) \leq \text{WSIZE}_1(P,\tilde{x}) \times \max_{\substack{i \in [n] \\ i:i \text{ is strong}}} \text{WSIZE}_1(P_{T^i},x^i)$$

$$\leq w \times \max_{\substack{i \in [n] \\ i:i \text{ is strong}}} \text{WSIZE}_1(P_{T^i},x^i). \tag{4.5}$$

Now if we take the subtree T^{i^*} that maximizes the 2nd line, then by the definition of k-fault trees, T^{i^*} is a $(k-1)$-fault tree. By inductive assumption, there is a span program for T^{i^*} satisfying $\text{WSIZE}_1(P_{T^{i^*}},x^i) \leq w^{k-1}$, so T satisfies $\text{WSIZE}_1(P_T,x) \leq w^k$, and there is a quantum algorithm for the tree that uses $O(w^k)$ queries.

Given the same setup, but now assuming the input $\tilde{x}$ makes f trivial, then by Eq (4.4) we have:

$$\text{WSIZE}_1(P_T, x) \leq \text{WSIZE}_1(P, \tilde{x}) \times \max_{\substack{i \in [n] \\ i:i \text{ is strong}}} \text{WSIZE}_1(P^i, x^i)$$

$$= 1 \times \max_{\substack{i \in [n] \\ i:i \text{ is strong}}} \text{WSIZE}_1(P^i, x^i). \tag{4.6}$$

Now if we take the subtree T^{i^*} that maximizes the 2nd line, then by the definition of fault trees, T^{i^*} is a κ fault tree with $\kappa \leq k$. But we know if $\kappa \leq k - 1$, then $\text{WSIZE}_1(P_{T^{i^*}}, x^i) \leq w^{k-1}$ by inductive assumption, so we're done in that case. So instead we assume $\kappa = k$. Thus we have reduced the problem to a smaller depth tree, and we can repeat the above procedure until we find the first subtree with a fault at its root (in which case we are back to the previous case) or show that there are no further faults in the tree (in which case the tree can be evaluated in $O(1)$ queries). Since the tree has finite depth, this procedure will terminate. □

5 Conclusions

We describe a method for upper bounding the quantum query complexity of Boolean functions using the general adversary bound. Using this method, we show that CONSTANT-FAULT DIRECT TREES can always be evaluated using $O(1)$ queries. Furthermore, we create an algorithm with a matching upper bound using span programs.

We would like to find other examples where Theorem 2.1 is useful, although we suspect that CONSTANT-FAULT DIRECT TREES are a somewhat unique case. It is clear from the span program algorithm described in Section 4 that Theorem 2.1 will not be useful for composed functions where the base function is created using this type of span program. However, there could be other types of quantum walk algorithms, for example, to which Theorem 2.1 might be applied. In any case, this work suggests that new ways of upper bounding the general adversary bound could give us a second window into quantum query complexity beyond algorithms.

Beside the practical application of Theorem 2.1, the result tells us something abstract and general about the structure of quantum algorithms. There is a natural way that quantum algorithms should compose, and if an algorithm does not compose in this natural way, then one knows that something is non-optimal.

6 Acknowledgements

A   Composition Proof

In this section, we will prove Lemma 2.2:

Lemma 2.2. *For any Boolean function $f : S \rightarrow \{0,1\}$ with $S \subseteq \{0,1\}^n$ and natural number d,*

$$\mathrm{ADV}^{\pm}(f^d) \geq (\mathrm{ADV}^{\pm}(f))^d. \tag{2.1}$$

This proof follows Appendix C from Lee et al. [8] very closely, including most notation. The difference between this Lemma and that in [8] is that f is allowed to be partial. We write out most of the proof again because it is subtle where the partiality of f enters the proof, and to allow this appendix to be read without constant reference to [8].

First, we use an expression for the general adversary bound derived from the dual program of the general adversary bound:

$$\mathrm{ADV}^{\pm}(g) = \max_{W,\,\Omega \circ I = \Omega} W \bullet J$$

$$\text{subject to } W \circ G = 0$$

$$\Omega \pm W \circ \Delta_i \succeq 0$$

$$\mathrm{Tr}(\Omega) = 1 \tag{A.1}$$

where $g : C \rightarrow \{0,1\}$, with $C \subseteq \{0,1\}^m$ and all matrices are indexed by $x, y \in C$, so e.g. $[W]_{xy}$ is the element of W in the row corresponding to input x and column corresponding to input y. W can always be chosen to be symmetric. G satisfies $[G]_{xy} = \delta_{g(x),g(y)}$, and Δ_i satisfies $[\Delta_i]_{xy} = 1 - \delta_{x_i,y_i}$, with x_i the value of the ith bit of the input x. We call Δ_i the filtering matrix. $\succeq 0$ means positive semidefinite, J is the all 1's matrix, and $W \bullet J$ means take the sum of all elements of W. When $\circ$ is used between uppercase or Greek letters, it denotes Hadamard product, while between lowercase letters, it denotes composition.

We want to determine the adversary bound for a composed function $f \circ g$ consisting of the functions $g : C \rightarrow \{0,1\}$ with $C \subseteq \{0,1\}^m$ and $f : S \rightarrow \{0,1\}$ with $S \subseteq \{0,1\}^n$. We consider the input to $f \circ g$ to be a vector of inputs $x = (x^1, \ldots, x^n)$ with $x^i \in C$. Given an input x to the composed function, we denote the input to the f part of the function as $\tilde{x}$: $\tilde{x} = (g(x^1), \ldots, g(x^n))$. Let (W, Ω) be an optimal solution for g with $\mathrm{ADV}^{\pm}(g) = d_g$ and (V, Λ) be an optimal solution for f with $\mathrm{ADV}^{\pm}(f) = d_f$. To clarify the filtering matrices, we say Δ_q^g is indexed by inputs to g, Δ_p^f is indexed by inputs to f, and $\Delta_{(p,q)}^{f \circ g}$ is indexed by inputs to the composed function $f \circ g$. (So $\Delta_{(p,q)}^{f \circ g}$ refers to the $(pm+q)^{\text{th}}$ bit of the input string.)

We assume that the initial input $x = (x^1, \ldots, x^n)$ is valid for the g part of the composition, i.e. $x^i \in C$ $\forall i$. A problem might arise if $\tilde{x}$, the input to f, is not an element of S. This is an issue that Lee et al. do not have to deal with, but which might affect the proof. Here we show that the proof goes through with small modifications.

The main new element we introduce is a set of primed matrices, which extend the matrices indexed by inputs to f to be indexed by all elements of $\{0,1\}^n$, not just those in S. For a primed matrix A', indexed by $x, y \in \{0,1\}^n$, if $x \notin S$ or $y \notin S$, then $[A']_{xy} = 0$. We use similar notation for matrices indexed by $x = (x^1, \ldots, x^n)$ where $\tilde{x} \in S$; we create primed matrices by extending the indeces to all inputs x by making those elements with $\tilde{x} \notin S$ have value 0. Notice if the extended matrices (W', Ω') are a solution to the

dual program, then the reduced matrices (W, Ω) are also a solution. For matrices A' indexed by $\{0,1\}^n$, we define a new matrix $\tilde{A}'$ indexed by C^n, as $[\tilde{A}']_{xy} = [A']_{\tilde{x}\tilde{y}}$, where $\tilde{x}$ is the output of the g functions on the input x, and likewise for $\tilde{y}$ and y. $\tilde{A}'$ expands each element of A' into a block of elements.

Before we get to the main lemma, we will need a few other results:

Lemma A.1. *[8] Let M' be a matrix labeled by $x \in \{0,1\}^n$, and $\tilde{M}'$ be defined as above. Then if $M' \succeq 0$, $\tilde{M}' \succeq 0$.*

Proof. This claim is stated without proof in [8]. $\tilde{M}'$ is created by turning all of the elements of M' into block matrices with repeated inputs. When an index $x \in \{0,1\}^n$ is expanded to a block of k elements, there are $k-1$ eigenstates of $\tilde{M}'$ that only have nonzero elements on this block and that have eigenvalue 0. By considering all 2^n blocks (each element of $\{0,1\}^n$ becomes a block) we obtain $2^n(k-1)$ 0-valued eigenvectors. Next we use the eigenvectors $\vec{v}^i$ of M' to create new vectors $\vec{\tilde{v}}^i$ in the space of $\tilde{M}'$. We give every element in the xth block of $\vec{\tilde{v}}^i$ the value $\vec{v}^i(x)/k$, where $\vec{v}^i(x)$ is the xth element of $\vec{v}^i$. The vectors $\vec{\tilde{v}}^i$ complete the basis with the 0-valued eigenvectors, and are orthogonal to the 0-valued vectors, but not to each other. However, the $\vec{\tilde{v}}^i$ have the property that $\vec{\tilde{v}}^{iT} \tilde{M}' \vec{\tilde{v}}^j = \delta_{ij} \lambda_i$ where λ_i is the eigenvalue of $\vec{v}^i$, so $\lambda_i \geq 0$. Thus using these vectors as a basis, we have that $\vec{u}^T \tilde{M}' \vec{u} \geq 0$ for all vectors $\vec{u}$. $\square$

The following is identical to Claim C.1 from [8] and follows because there is no restriction that g be a total function. Thus we state it without proof:

Lemma A.2. *For a function g, there is a solution to the dual program, (W, Ω), such that $\mathrm{ADV}^{\pm}(g) = d_g$, $d_g \Omega \pm W \succeq 0$, and $\sum_{x:g(x)=1} \Omega(x,x) = \sum_{x:g(x)=0} \Omega(x,x) = 1/2$.*

In Lemma A.3, we will show that $\mathrm{ADV}^{\pm}(f \circ g) = \mathrm{ADV}^{\pm}(f)\mathrm{ADV}^{\pm}(g)$, which implies Lemma 2.2.

Lemma A.3. *A solution to the dual program for $f \circ g$ is (U, Υ), where $(U', \Upsilon') = (c \times \tilde{V}' \circ (d_g \Omega + W)^{\otimes n}, c \times d_g^{n-1} \tilde{\Lambda}' \circ \Omega^{\otimes n})$ and $c = 2^n d_g^{-(n-1)}$. (U, Υ) give the adversary bound $\mathrm{ADV}^{\pm}(f \circ g) = d_g d_f$.*

Proof. The first thing to check is that U' and Υ' are valid primed matrices, or otherwise we can not recover U and Υ. Because each of U' and Υ' are formed by Hadamard products with primed matrices, they themselves are also primed matrices.

We next calculate the objective function, and afterwards check that (U', Υ') satisfy the conditions of the dual program.

The objective function gives:

$$
\begin{aligned}
J \bullet (c\tilde{V}' \circ (d_g \Omega + W)^{\otimes n}) &= c \sum_{\substack{a,b \in S \\ f(a) \neq f(b)}} [V]_{ab} \sum_{\substack{x,y \\ \tilde{x}=a, \tilde{y}=b}} \prod_i (d_g[\Omega]_{x^i y^i} + [W]_{x^i y^i}) \\
&= c \sum_{\substack{a,b \in S \\ f(a) \neq f(b)}} [V]_{ab} \prod_i \sum_{\substack{x^i, y^i \\ g(x^i)=a_i \\ g(y^i)=b_i}} (d_g[\Omega]_{x^i y^i} + [W]_{x^i y^i}) \quad\quad \text{(A.2)}
\end{aligned}
$$

where in the first line we've replaced V' by V because adding extra 0's does not affect the sum. In the second line, a_i and b_i are the ith bits of a and b respectively, and we've changed the order of multiplication

and addition. This ordering change is not affected by the fact that f is partial, since the first summation already fixes an input to f.

We now examine the sum

$$\sum_{\substack{x^i, y^i \\ g(x^i) = a_i \\ g(y^i) = b_i}} (d_g [\Omega]_{x^i y^i} + [W]_{x^i y^i}). \tag{A.3}$$

We consider the cases $a_i = b_i$, and $a_i \neq b_i$ separately. When $a_i = b_i$, because $W \circ G = 0$, we know that $[W]_{x^i y^i} = 0$, so in this case, only $[\Omega]_{x^i y^i}$ is non-zero. Since Ω is diagonal, it only has non-zero values when $x^i = y^i$, and using Lemma A.2, the sum is $d_g / 2$. When $a_i \neq b_i$, then $x^i \neq y^i$, so $[\Omega]_{x^i y^i} = 0$. In this case, the sum will include exactly half of the elements of W: either those elements with $g(x^i) = 0$ and $g(y^i) = 1$, or with $g(x^i) = 1$ and $g(y^i) = 0$. Since W is symmetric, this amounts to $\frac{1}{2} W \bullet J = d_g / 2$. Multiplying n times for the product over the $i's$ and using the definition of the objective function for f gives the final result:

$$J \bullet (c \tilde{V}' \circ (d_g \Omega + W)^{\otimes n}) = c \times d_f \left(\frac{d_g}{2} \right)^n = d_f d_g \tag{A.4}$$

Now we show that U' and Υ' satisfy the conditions of the dual program. We require that $[U']_{xy} = 0$ for $(f \circ g)(x) = (f \circ g)(y)$. Notice $U' = 0$ whenever $\tilde{V}' = 0$, and $[\tilde{V}']_{xy} = 0$ for $(f \circ g)(x) = (f \circ g)(y)$, so this requirement holds. Likewise Υ' is a diagonal matrix because it can only be nonzero where $\Omega^{\otimes n}$ is non-zero, and $\Omega^{\otimes n}$ is diagonal.

Next we will show that $\Upsilon' \pm U' \circ (\Delta_{(p,q)}^{f \circ q})' \succeq 0$. From Lemma A.2 and from the conditions on the dual programs for f and g, we have $d_g \Omega \pm W \succeq 0$, $\Omega \pm W \circ \Delta_q^g \succeq 0$, and $\Lambda' \pm V' \circ (\Delta_p^f)' \succeq 0$. Then by Lemma A.1, $\tilde{\Lambda}' \pm \tilde{V}' \circ (\tilde{\Delta}_p^f)' \succeq 0$. Since tensor and Hadamard products preserve semidefinite positivity, we get

$$0 \preceq (\tilde{\Lambda}' \pm \tilde{V}' \circ (\tilde{\Delta}_p^f)') \circ \left((d_g \Omega + W)^{\otimes(p-1)} \otimes (\Omega + W \circ \Delta_q^g) \otimes (d_g \Omega + W)^{\otimes(n-p)} \right), \tag{A.5}$$

where these matrices are indexed by all elements of C^n. $[W]_{x^i y^i} = 0$ for $x = y$ while $\tilde{\Lambda}'$ is only nonzero for elements $[\tilde{\Lambda}']_{xy}$ with $x = y$, so any terms involving a Hadamard of W and $\tilde{\Lambda}'$ are 0. Similarly, the Ω in the pth tensor product is only nonzero for $x^p = y^p$, but for these inputs, the term $(\tilde{\Delta}_p^f)'$ is always zero, so in fact the non-zero terms of this Ω do not contribute. Thus we are free to replace this Ω with $d_g \Omega \circ \Delta_q^g$. We obtain

$$0 \preceq d_g^{n-1} \tilde{\Lambda}' \circ \Omega^{\otimes n} \pm (\tilde{V}' \circ (\tilde{\Delta}_p^f)') \circ \left((d_g \Omega + W)^{\otimes(p-1)} \otimes (d_g \Omega \circ \Delta_q^g + W \circ \Delta_q^g) \otimes (d_g \Omega + W)^{\otimes(n-p)} \right)$$

$$0 \preceq d_g^{n-1} \tilde{\Lambda}' \circ \Omega^{\otimes n} \pm (\tilde{V}' \circ (\tilde{\Delta}_p^f)') \circ \left((d_g \Omega + W)^{\otimes n} \circ \{ J^{\otimes(p-1)} \otimes \Delta_q^g \otimes J^{\otimes(n-p)} \} \right). \tag{A.6}$$

Finally, the term $(\tilde{\Delta}_p^f)'$ can be written as $J - G$ acting on only the pth term in the tensor product $(d_g \Omega + W)^{\otimes n}$, so we need to evaluate $(J - G) \circ (d_g \Omega + W) \circ \Delta_q^g$. We have $(J - G) \circ \Omega = \Delta_q^g \circ \Omega = 0$, and $(J - G) \circ W = W$, so we can remove $(\tilde{\Delta}_p^f)'$ without altering the expression.

Now the term $\{ J^{\otimes(p-1)} \otimes \Delta_q^g \otimes J^{\otimes(n-p)} \}$ is almost $\Delta_{(p,q)}^{f \circ g}$, except it is like a primed matrix; its indeces are all elements in C^n, not just valid inputs to f, yet it is not primed, in that some of its elements to

non-valid inputs to f are non-zero. However it is involved in a Hadamard product with $\tilde{V}'$, a primed matrix, so all of the terms corresponding to non-valid inputs are zeroed, and we can make it be a primed matrix without affecting the expression. We obtain

$$0 \preceq d_g^{n-1} \tilde{\Lambda}' \circ \Omega^{\otimes n} \pm \left(\tilde{V}' \circ \left((d_g \Omega + W)^{\otimes n} \circ (\Delta_{(p,q)}^{f \circ g})' \right) \right), \tag{A.7}$$

which is precisely the positivity constraint of the dual program.

Finally, we need to check that $\mathrm{Tr}(c d_g^{n-1} \tilde{\Lambda}' \circ \Omega^{\otimes n}) = 1$:

$$\begin{aligned}
\mathrm{Tr}(c d_g^{n-1} \tilde{\Lambda}' \circ \Omega^{\otimes n}) &= c d_g^{n-1} \sum_{a \in S} [\Lambda]_{aa} \sum_{x : \tilde{x} = a} \prod_i [\Omega]_{x^i x^i} \\
&= c d_g^{n-1} \sum_{a \in S} [\Lambda]_{aa} \prod_i \sum_{x^i : g(x^i) = a_i} [\Omega]_{x^i x^i} \\
&= c d_g^{n-1} \left(\frac{1}{2} \right)^n = 1,
\end{aligned} \tag{A.8}$$

where all of the tricks here follow similarly from the discussion following Eq. (A.2). □

Lemma 2.2 now follows from Lemma A.3 along with a simple inductive argument.

References

[1] ANDRIS AMBAINIS: Quantum lower bounds by quantum arguments. In *Proc. 32nd ACM STOC*, pp. 636–643, 2000. [doi:http://doi.acm.org/10.1145/335305.335394] 1, 2

[2] ANDRIS AMBAINIS: Polynomial degree vs. quantum query complexity. *J. Comput. Syst. Sci.*, 72(2):220–238, 2006. [doi:10.1016/j.jcss.2005.06.006] 1

[3] ANDRIS AMBAINIS, LOÏCK MAGNIN, MARTIN ROETTELER, AND JÉRÉMIE ROLAND: Symmetry-assisted adversaries for quantum state generation. In *Proc. 24th IEEE CCC*, pp. 167–177, 2011. [doi:10.1109/CCC.2011.24] 1

[4] ANDREW M. CHILDS, RICHARD CLEVE, STEPHEN P. JORDAN, AND DAVID YONGE-MALLO: Discrete-query quantum algorithm for NAND trees. *Theory of Computing*, 5(5):119–123, 2009. [doi:10.4086/toc.2009.v005a005] 4

[5] EDWARD FARHI, JEFFREY GOLDSTONE, AND SAM GUTMANN: A quantum algorithm for the hamiltonian nand tree. *Theory of Computing*, 4(1):169–190, 2008. [doi:10.4086/toc.2008.v004a008] 4

[6] PETER HØYER, TROY LEE, AND ROBERT ŠPALEK: Negative weights make adversaries stronger. In *Proc. 39th ACM STOC*, pp. 526–535, 2007. [doi:http://doi.acm.org/10.1145/1250790.1250867] 1, 3

[7] PETER HØYER, JAN NEERBEK, AND YAOYUN SHI: Quantum complexities of ordered searching, sorting, and element distinctness. *Algorithmica*, 34:429–448, 2008. [doi:http://dx.doi.org/10.1007/s00453-002-0976-3] 1, 2

[8] TROY LEE, RAJAT MITTAL, BEN W. REICHARDT, ROBERT ŠPALEK, AND MARIO SZEGEDY: Quantum query complexity of state conversion. In *Proc. 52nd IEEE FOCS*, pp. 344 –353, 2011. [doi:10.1109/FOCS.2011.75] 1, 2, 3, 6, 10, 11

[9] BEN W. REICHARDT: Span programs and quantum query complexity: The general adversary bound is nearly tight for every boolean function. In *Proc. 50th IEEE FOCS*, pp. 544–551, 2009. [doi:10.1109/FOCS.2009.55] 2, 6, 7, 8

[10] BEN W. REICHARDT: Reflections for quantum query algorithms. In *Proc. 22nd ACM-SIAM SODA*, pp. 560–569, 2011. [arXiv:1005.1601] 4

[11] BEN W. REICHARDT AND ROBERT ŠPALEK: Span-program-based quantum algorithm for evaluating formulas. In *Proc. 40th ACM STOC*, pp. 103–112, 2008. [doi:http://doi.acm.org/10.1145/1374376.1374394] 1, 2, 6

[12] MICHAEL SAKS AND AVI WIGDERSON: Probabilistic boolean decision trees and the complexity of evaluating game trees. In *Proc. 27th IEEE FOCS*, pp. 29–38, 1986. [doi:http://doi.ieeecomputersociety.org/10.1109/SFCS.1986.44] 4

[13] BOHUA ZHAN, SHELBY KIMMEL, AND AVINATAN HASSIDIM: Super-polynomial quantum speed-ups for boolean evaluation trees with hidden structure. In *Proc. 3rd ACM ITCS*, pp. 249–265, 2012. [doi:10.1145/2090236.2090258] 2, 4, 5, 6, 7

AUTHOR

Shelby Kimmel
Ph.D. Candidate
Massachusetts Institute of Technology
skimmel@mit.edu
http://www.mit.edu/~skimmel