View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Separations in Query Complexity Based on Pointer Functions

Andris Ambainis
Faculty of Computing
University of Latvia
andris.ambainis@lu.lv

Kaspars Balodis
FC and IMCS
University of Latvia
kbalodis@gmail.com

Aleksandrs Belovs
CWI
Amsterdam, the Netherlands
stiboh@gmail.com

Troy Lee
SPMS, NTU; and CQT and
MajuLab, NUS, Singapore
troyjlee@gmail.com

Miklos Santha
IRIF, CNRS, Paris, France;
and CQT, NUS, Singapore
miklos.santha@gmail.com

Juris Smotrovs
Faculty of Computing
University of Latvia
juris.smotrovs@lu.lv

ABSTRACT

In 1986, Saks and Wigderson conjectured that the largest separation between deterministic and zero-error randomized query complexity for a total boolean function is given by the function f on $n = 2^k$ bits defined by a complete binary tree of NAND gates of depth k, which achieves $R_0(f) = O(D(f)^{0.7537...})$. We show this is false by giving an example of a total boolean function f on n bits whose deterministic query complexity is $\Omega(n/\log(n))$ while its zero-error randomized query complexity is $\widetilde{O}(\sqrt{n})$. We further show that the quantum query complexity of the same function is $\widetilde{O}(n^{1/4})$, giving the first example of a total function with a super-quadratic gap between its quantum and deterministic query complexities.

We also construct a total boolean function g on n variables that has zero-error randomized query complexity $\Omega(n/\log(n))$ and bounded-error randomized query complexity $R(g) = \widetilde{O}(\sqrt{n})$. This is the first super-linear separation between these two complexity measures. The exact quantum query complexity of the same function is $Q_E(g) = \widetilde{O}(\sqrt{n})$.

These functions show that the relations $D(f) = O(R_1(f)^2)$ and $R_0(f) = \widetilde{O}(R(f)^2)$ are optimal, up to poly-logarithmic factors. Further variations of these functions give additional separations between other query complexity measures: a cubic separation between Q and R_0, a 3/2-power separation between Q_E and R, and a 4th power separation between approximate degree and bounded-error randomized query complexity.

All of these examples are variants of a function recently introduced by Göös, Pitassi, and Watson which they used to separate the unambiguous 1-certificate complexity from deterministic query complexity and to resolve the famous Clique versus Independent Set problem in communication complexity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

STOC'16, June 19–21, 2016, Cambridge, MA, USA
ACM. 978-1-4503-4132-5/16/06...$15.00
http://dx.doi.org/10.1145/2897518.2897524

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation

General Terms

Theory

Keywords

Deterministic algorithms, Randomized algorithms, Quantum algorithms, Monte Carlo, Las Vegas

1. INTRODUCTION

Query complexity has been very useful for understanding the power of different computational models. In the standard version of the query model, we want to compute a boolean function $f\colon \{0,1\}^n \to \{0,1\}$ on an initially unknown input $x \in \{0,1\}^n$ that can only be accessed by asking queries of the form $x_i = ?$. The advantage of query complexity is that we can often prove tight lower bounds and have provable separations between different computational models. This is in contrast to the Turing machine world where lower bounds and separations between complexity classes often have to rely on unproven assumptions. At the same time, the model of query complexity is simple and captures the essence of quite a few natural computational processes.

We use $D(f), R(f)$ and $Q(f)$ to denote the minimum number of queries in deterministic, randomized and quantum query algorithms[1] that compute f. It is easy to see that $Q(f) \leq R(f) \leq D(f)$ for any function f. For partial functions (that is, functions whose domain is a strict subset of $\{0,1\}^n$), huge separations are known between all these measures. For example, a randomized algorithm can tell if an n-bit boolean string has 0 ones or at least $n/2$ ones with a constant number of queries, while any deterministic algorithm requires $\Omega(n)$ queries to do this. Similarly, Aaronson and Ambainis [1] recently constructed a partial boolean function f on n variables that can be evaluated using one

[1]By default, we use $R(f)$ and $Q(f)$ to refer to bounded-error algorithms (i.e., algorithms that compute $f(x)$ correctly on every input x with probability at least 9/10).

	lower bound for all f	previous separation	this paper	function	result
$R_0(f)$	$\Omega(D(f)^{1/2})$ [13, 5, 23]	$O(D(f)^{0.753\cdots})$ [22]	$\widetilde{O}(D(f)^{1/2})$	$f_{2n,n}$	Corollary 7
$Q(f)$	$\Omega(D(f)^{1/6})$ [3]	$O(D(f)^{1/2})$ [12]	$\widetilde{O}(D(f)^{1/4})$	$f_{2n,n}$	Corollary 7
$R_1(f)$	$\Omega(R_0(f)^{1/2})$[18]	$O(R_0(f))$	$\widetilde{O}(R_0(f)^{1/2})$	$g_{n,n}$	Corollary 14
$Q_E(f)$	$\Omega(R_0(f)^{1/3})$[15]	$O(R_0(f)^{0.867\cdots})$[2]	$\widetilde{O}(R_0(f)^{1/2})$	$g_{n,n}$	Corollary 14
$Q(f)$	$\Omega(R_0(f)^{1/6})$[3]	$O(R_0(f)^{1/2})$ [12]	$\widetilde{O}(R_0(f)^{1/3})$	h_{n,n,n^2}	Corollary 19
$Q_E(f)$	$\Omega(R(f)^{1/3})$ [15]	$O(R(f)^{0.867\cdots})$ [2]	$\widetilde{O}(R(f)^{2/3})$	h_{1,n,n^2}	Corollary 21
$\widetilde{\deg}(f)$	$\Omega(D(f)^{1/6})$ [3]	$O(R(f)^{1/2})$ [19]	$\widetilde{O}(R(f)^{1/4})$	h_{1,n,n^2}	Corollary 23

Table 1: Attained separations.

quantum query but requires $\Omega(\sqrt{n})$ queries for randomized algorithms.

The situation is quite different for total functions.[2] Here it is known that $D(f), R(f)$, and $Q(f)$ are all polynomially related. In fact, $D(f) = O(R(f)^3)$ [18] and $D(f) = O(Q(f)^6)$ [3]. A popular variant of randomized algorithms is the zero-error (Las Vegas) model in which a randomized algorithm always has to output the correct answer, but the number of queries after which it stops can depend on the algorithm's coin flips. The complexity $R_0(f)$ is defined as the expected number of queries, over the randomness of the algorithm, for the worst case input x. A tighter relation $D(f) \leq R_0(f)^2$ is known for Las Vegas algorithms (this was independently observed by several authors [13, 5, 23]). Nisan has even shown $D(f) = O(R_1(f)^2)$ [18], where $R_1(f)$ is the one-sided error randomized complexity of f. Recently, Kulkarni and Tal [14], basing on a result by Midrijānis [16], showed that $R_0(f) = \widetilde{O}(R(f)^2)$, where the $\widetilde{O}$ notation hides poly-logarithmic factors.

While it has been widely conjectured that these relations are not tight, little progress has been made in the past 20 years on improving these upper bounds or exhibiting functions with separations approaching them. Between $D(f)$ and $R_0(f)$, the best separation known for a total function is the function NAND^k on $n = 2^k$ variables defined by a complete binary NAND tree of depth k. This function satisfies $R_0(\text{NAND}^k) = O(D(\text{NAND}^k)^{0.7537\cdots})$ [22]. Saks and Wigderson showed that this upper bound is optimal for NAND^k, and conjectured that this is the largest gap possible between $R_0(f)$ and $D(f)$ [20]. This function also provides the largest known gap between $R(f)$ and $D(f)$, and satisfies $R(\text{NAND}^k) = \Omega(R_0(\text{NAND}^k))$ [21]. This situation points to the broader fact that, as far as we are aware, no super-linear gap is known between $R(f)$ and $R_0(f)$ for a total function f. Between $Q(f)$ and $D(f)$, the largest known separation is quadratic, given by the OR function on n bits, which satisfies $Q(f) = O(\sqrt{n})$ [12] and $D(f) = \Omega(n)$.

1.1 Our Results

We improve the best known separations between all of these measures. In particular, we show that

- There is a function f with $R_0(f) = \widetilde{O}(D(f)^{1/2})$. This refutes the nearly 30 year old conjecture of Saks and Wigderson [20], and shows that the upper bounds $D(f) \leq R_0(f)^2$ and $D(f) = O(R_1(f)^2)$ are tight, up to poly-logarithmic factors.

- There is a function f with $R_1(f) = \widetilde{O}(R_0(f)^{1/2})$. This is also nearly optimal due to Nisan's result $D(f) = O(R_1(f)^2)$, as well as the result $R_0(f) = \widetilde{O}(R(f)^2)$ by Kulkarni and Tal. Previously, no super-linear separation was known even between $R(f)$ and $R_0(f)$.

- There is a function f with $Q(f) = \widetilde{O}(D(f)^{1/4})$. This is the first improvement in nearly 20 years to the quadratic separation given by Grover's search algorithm [12].

- Let $Q_E(f)$ be the exact quantum query complexity, the minimal number of queries needed by a quantum algorithm that stops after a fixed number of steps and outputs $f(x)$ with probability 1. We exhibit functions f_1, f_2 for which $Q_E(f_1) = \widetilde{O}(R_0(f_1)^{1/2})$ and $Q_E(f_2) = \widetilde{O}(R(f_2)^{2/3})$. This improves the best known separation of Ambainis from 2011 [2] giving an f for which $Q(f) = O(R(f)^{0.867\cdots})$. Prior to the work of Ambainis, no super-linear separation was known, the largest known separation being a factor of 2, attained for the PARITY function [9].

A full list of our results are given in the Table 1. Subsequent to our work, Ben-David [4] has additionally given a super-quadratic separation between $Q(f)$ and $R(f)$, exhibiting a function with $Q(f) = \widetilde{O}(R(f)^{2/5})$.

Other separations can be obtained from this table using relations between complexities in Figure 2.

1.2 Göös-Pitassi-Watson Function

All of our separations are based on an amazing function recently introduced by Göös, Pitassi, and Watson [11] to resolve the deterministic communication complexity of the Clique vs. Independent set problem, thus solving a long-standing open problem in communication complexity. They solved this problem by first solving a corresponding question in the query complexity model and then showing a general "lifting theorem" that lifts the hardness of a function in the deterministic query model to the hardness of a derived function in the model of deterministic communication complexity. In the query complexity model, their goal was to exhibit a total boolean function f that has large deterministic query complexity and small unambiguous 1-certificate complexity.[3]

[2] In the rest of the paper we will exclusively talk about total functions. Hence, we sometimes drop this qualification.

[3] A subcube is the set of strings consistent with a partial assignment $x_{i_1} = b_1, \ldots, x_{i_s} = b_s$. Its length is s, the number of assigned variables. The unambiguous 1-certificate complexity is the smallest s such that $f^{-1}(1)$ can be partitioned into subcubes of length s (whose corresponding partial assignments are consequently 1-certificates of f). See Section 2 for full definitions.

The starting point of their construction is the boolean function $f\colon \{0,1\}^M \to \{0,1\}$ with the input variables $x_{i,j}$ arranged in a rectangular grid $M = [n] \times [m]$. The value of the function is 1 if and only if there exists a unique all-1 column. The deterministic complexity of this function is $\Omega(nm)$, since it is hard to distinguish an input with precisely one zero in each column from the input in which one of the zeroes is flipped to one. It is also easy to construct a 1-certificate of length $n+m-1$: Take the all-1 column and one zero from each of the remaining columns. This certificate is not always unique, however, as there can be multiple zeroes in a column and any of them can be chosen in a certificate. Indeed, it is impossible to *partition* the set of all positive inputs into subcubes of small length.

Göös *et al.* added a surprisingly simple ingredient that solves this problem: pointers to cells in M. One can specify which zero to take from each column by requiring that there is a path of pointers that starts in the all-1 column and visits exactly one zero in all other columns, see Figure 1. Thus, the set of positive inputs breaks apart into a disjoint union of subcubes of small length. Since the pointers provide great flexibility in the positioning of zeroes, this function is still hard for a deterministic algorithm.

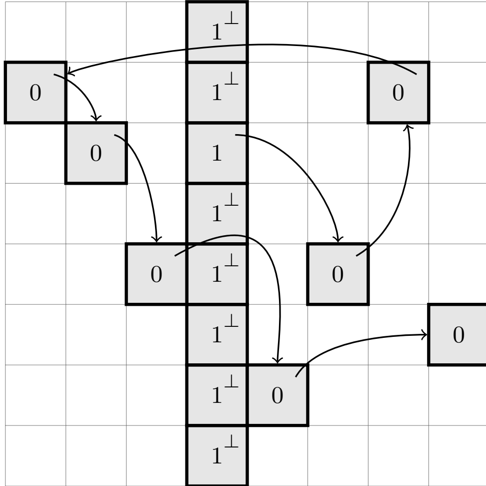

Figure 1: An example of a 1-certificate for the Göös-Pitassi-Watson function. The center of a cell $x_{i,j}$ shows $\mathrm{val}(x_{i,j})$ and the top right corner shows $\mathrm{point}(x_{i,j})$

Formally, the definition of the Göös-Pitassi-Watson function is as follows. Let n and m be positive integers, and $M = [n] \times [m]$ be a grid with n rows and m columns. Let $\widetilde{M} = M \cup \{\perp\}$. Elements in $\widetilde{M}$ are considered as pointers to the cells of M, where $\perp$ stands for the null pointer.

The function $g_{n,m}\colon (\{0,1\} \times \widetilde{M})^M \to \{0,1\}$ is defined as follows. We think of each tuple $v = (b,p) \in \{0,1\} \times \widetilde{M}$ in the following way. The element $b \in \{0,1\}$ of the pair is the *value* and the second element $p \in \widetilde{M}$ is the *pointer*. We will use the notation $\mathrm{val}(v) = b$ and $\mathrm{point}(v) = p$.

Although $g_{n,m}$ is not a boolean function, it can be converted into an associated boolean function by encoding the elements of the input alphabet $\Sigma = \{0,1\} \times \widetilde{M}$ using $\lceil \log |\Sigma| \rceil$ bits.

An input $(x_{i,j})_{(i,j)\in M}$ evaluates to 1 if and only if the following three conditions are satisfied (see Figure 1 for an illustration):

1. There is exactly one column b such that $\mathrm{val}(x_{i,b}) = 1$ for all $i \in [n]$. We call this the *marked column*.
2. In the marked column, there exists a unique cell a such that $x_a \neq (1, \perp)$. We call a the *special element*.
3. For the special element a, by following the pointers inductively defined as $p_1 = \mathrm{point}(x_a)$ and $p_{s+1} = \mathrm{point}(x_{p_s})$ for $s = 1, \ldots, m-2$ we visit every column except the marked column, and $\mathrm{val}(x_{p_s}) = 0$ for each $s = 1, \ldots, m-1$. We call $p_1, \ldots, p_{m-1}$ the *highlighted zeroes*.

For each positive input x, the all-1 column satisfying items (1) and (2) of the definition, and the highlighted zeroes from (3) give a unique minimal 1-certificate of x. Thus, the unambiguous 1-certificate complexity of this function is $n+m-1$. Göös *et al.* showed that this function has deterministic query complexity mn, giving a quadratic separation between the two when $n = m$.

1.3 Our Technique and Pointer Functions

As described in the previous section, Göös *et al.* showed how pointers can make certificates unambiguous without substantially increasing their size. This technique turns out to be quite powerful for other applications as well.

Using the Göös-Pitassi-Watson function, it is already possible to give a larger separation between randomized and deterministic query complexity than previously known. For instance, Mukhopadhyay and Sanyal [17], independently from our work, obtained separations $R(f) = \widetilde{O}(\sqrt{D(f)})$ and $R_0(f) = \widetilde{O}(D(f)^{3/4})$. However, these algorithms are rather complicated, and it is not known whether this function can realize an optimal separation between $R_0(f)$ and $D(f)$.

We instead modify the Göös-Pitassi-Watson function in various ways. For the separation between R_0 and D the key new idea we add is the use of *back pointers*; for the separation between Q and D, in addition to back pointers, we further replace the path of zeroes in the Göös-Pitassi-Watson function with a *balanced tree* whose leaves are the highlighted zeroes; finally, we consider a modification of the Göös-Pitassi-Watson function where there are *multiple* marked columns for the separation between Q and R_0. We now describe our modifications in more detail.

Back pointers.

A back pointer points either to a cell in M or to a column in $[m]$. For instance, in order to get a quadratic separation between $R_0(f)$ and $D(f)$, we require that each highlighted zero points back to the marked (all-1) column. It turns out that this function is still hard for a deterministic algorithm. A randomized algorithm, on the other hand, can take advantage of the back pointers to quickly find the all-1 column, if it exists. The algorithm begins by querying all elements in a column. Let Z be the set of zeroes in this column and $B(Z)$ be the set of columns pointed to by the back pointers in Z. If the value of function is 1, $B(Z)$ must contain the marked column. However, $B(Z)$ may also contain pointers to non-marked columns. We estimate the number of zeroes in each column of $B(Z)$ by sampling. If we find a zero in every column of $B(Z)$, then we can reject the input. On the other

hand, we can tune the sampling so that if no zero is found in a column $c \in B(Z)$, then, with high probability, c has at most $|Z|/2$ many zeroes. We then move to this column c and repeat the process. Even if c is not marked, we have made progress by halving the number of zeroes, and, in a logarithmic number of repetitions, we either find the marked column or reject.

In this way, back pointers to the marked column from the highlighted zeroes make the function easy for an R_0 algorithm, but hard for a deterministic one. Similarly, if we only require that at least half of the highlighted zeroes point back to the special element a from condition (2), the function becomes hard for R_0, but easy for R_1.

Making partial functions total.

From another vantage point, the pointer technique can essentially turn a partial function into a total one. This is beneficial as it is easy to prove separations for a partial function. Let us describe our separation between Monte Carlo and Las Vegas query complexities as an indicative example.

It is easy to provide a separation between $R(f)$ and $R_0(f)$ for partial functions. For example, consider the following partial boolean function f on m variables. For $x \in \{0,1\}^m$, the value of $f(x)$ is 1 if the Hamming weight $|x| \geq m/2$, and $f(x) = 0$ if $|x| = 0$. Otherwise, the function is not defined. The Monte Carlo query complexity of this function is $O(1)$, but its Las Vegas complexity is $m/2 + 1$, since it takes that many queries to reject the all-0 string.

How can we obtain a total function with the same property that there are either exactly 0 or at least $m/2$ marked elements? We define a variant of the Göös-Pitassi-Watson function, where we require that, in a positive input, at least $m/2$ of the highlighted zeroes point back to the special element a of condition (2). Consider an auxiliary function f on the columns of the grid M. For a column $j \in [m]$, $f(j) = 1$ if and only if the value of the original function is 1, and the highlighted zero in column j points back to a. Thus, by definition, either $f(j) = 0$ for all j, or $f(j) = 1$ for at least half of all $j \in [m]$. Given a column j, we can find a by analyzing the back pointers contained in column j. When a is found, it is easy to test whether the value of the function is 1 and the highlighted zero in column j points to a. Moreover, this procedure can be made deterministic and uses only $\widetilde{O}(n+m)$ queries.

Subsequent to our work, Ben-David [4] devised a different way of converting a partial function into a total one, and applied it to the forrelation problem [1] to give a function f with $Q(f) = \widetilde{O}(R(f)^{2/5})$, the first super-quadratic separation between these measures.

Use of a balanced tree.

Instead of a path through the highlighted zeroes as in condition (3) of the Göös-Pitassi-Watson function, we use a balanced binary tree with the zeroes being the leaves of the tree. This serves at least three purposes.

First, this allows for even greater flexibility in placing the zeroes. As they are the leaves of the tree, they are not required to point to other nodes. This helps in proving Las Vegas lower bounds.

Second, the tree allows "random access" to the highlighted zeroes. This is especially helpful for quantum algorithms. After the algorithm finds the marked column, it should check the highlighted zeroes. The last element of the path can be only accessed in m queries. But if we arrange the zeroes in a binary tree, each zero can be accessed in only a logarithmic number of queries, hence, they can be tested in $\widetilde{O}(\sqrt{m})$ queries using Grover's search.

Finally, it can make the function hard even for a Monte Carlo algorithm (if no back pointers are present). In the original Göös-Pitassi-Watson function, when a randomized algorithm finds a highlighted zero, it can follow the path starting from that cell. As the zeroes are arranged in a path, the algorithm can thus eliminate half of the potential marked columns on average. This fact is exploited by the algorithm of Mukhopadhyay and Sanyal [17].

Similarly, when an algorithm finds a node of the tree it can also explore the corresponding subtree. The difference is that the expected size of a subtree rooted in a node of the tree is only logarithmic. Thus, even if the algorithm finds this node, it does not learn much, and we are able to prove an $\Omega(nm/\log m)$ lower bound for a Monte Carlo algorithm.

In principle, all of the above problems can be solved by adding direct pointers from the special element (a in condition (2)) to a zero in each non-marked column (that is, using an $(m-1)$-ary tree of depth 1 instead of a binary tree of depth $O(\log m)$). The problem with this solution is that the size of the alphabet becomes exponential, rendering this construction useless for boolean functions.

Choice of separating functions.

We have outlined above three ingredients that can be added to the original Göös-Pitassi-Watson function: using various back pointers, identifying unmarked columns by a tree of pointers, and increasing the number of marked columns. These ingredients can be added in various combinations to produce different effects. In order to reduce the number of functions introduced, in this paper we stick to three variations:

- a function $f_{n,m}$ with back pointers to the marked column from each of the highlighted zeroes.
- a function $g_{n,m}$ with back pointers to the special element from half of the highlighted zeroes, and
- a function $h_{k,n,m}$ with k marked columns and no back pointers.

All three functions use a balanced binary tree.

This does not mean that a given separation cannot be proven with a different combination of ingredients. For example, as outlined above, the R_0 vs D separation can be proven for the original Göös-Pitassi-Watson function by equipping each highlighted zero with a back pointer to the marked column and not using the binary tree.

2. PRELIMINARIES

We let $[n] = \{1, 2, \ldots, n\}$. We use $f(n) = \widetilde{O}(g(n))$ to mean that there exists constants c, k and an integer N such that $|f(n)| \leq c|g(n)| \log^k(n)$ for all $n > N$.

In the remaining part of this section, we define the notion of query complexity for various models of computation. For more detail on this topic, the reader may refer to the survey [8]. Relations between various models are depicted in Figure 2.

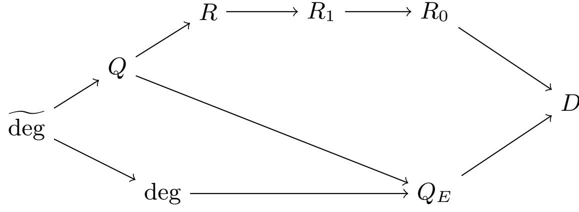

Figure 2: Relations between various complexities. An arrow means that complexity on the left is at most the complexity on the right.

Deterministic query complexity.

Let Σ be a finite set. A decision tree T on n variables and the input alphabet Σ is a rooted tree, where

- internal nodes are labeled by elements of $[n]$;
- every internal node v has degree $|\Sigma|$ and there is a bijection between the edges from v to its children and the elements of Σ;
- leaves are labeled from $\{0, 1\}$.

The output of the decision tree T on input $x \in \Sigma^n$, denoted $T(x)$, is determined as follows. Start at the root. If this is a leaf, then output its label. Otherwise, if the label of the root is $i \in [n]$, then follow the edge labeled by x_i (this is called a *query*) and recursively evaluate the corresponding subtree. We say that T computes the function $f \colon \Sigma^n \to \{0, 1\}$ if $T(x) = f(x)$ on every input x. The cost of T on input x, denoted $C(T, x)$, is the number of internal nodes visited by T on x. The deterministic query complexity $D(f)$ of f is the minimum over all decision trees T computing f of the maximum over all x of $C(T, x)$.

Randomized query complexity.

We follow the definitions for randomized query complexity given in [24, 18]. A randomized decision tree T_μ is defined by a probability distribution μ over deterministic decision trees. On input x, a randomized decision tree first selects a deterministic decision tree T according to μ, and then outputs $T(x)$. The expected cost of T_μ on input x is the expectation of $C(T, x)$ when T is picked according to μ. The worst-case expected cost of T_μ is the maximum over inputs x of the expected cost of T_μ on input x.

There are three models of randomized decision trees that differ in the definition of "computing" a function f.

- Zero-error (Las Vegas): It is required that the algorithm gives the correct output with probability 1 for every input x, that is, every deterministic decision tree T in the support of μ computes f.
- One-sided error: It is required that negative inputs are rejected with probability 1, and positive inputs are accepted with probability at least $1/2$.
- Two-sided error (Monte Carlo): It is required that the algorithm gives the correct output with probability at least $9/10$ for every input x.

The error probability in the one-sided and two-sided cases can be reduced to ε by repeating the algorithm $O(\log \frac{1}{\varepsilon})$ times.

We define randomized query complexities $R_0(f), R_1(f)$, and $R(f)$ as the minimum worst-case expected cost of a randomized decision tree to compute f in the zero, one-sided, and bounded-error sense, respectively.

Distributional query complexity.

A common way to show lower bounds on randomized complexity, and the way we will do it in this paper, is to consider distributional complexity [24]. The cost of a deterministic decision tree T with respect to a distribution ν, denoted $C(T, \nu)$, is $\mathbb{E}_{x \leftarrow \nu}[C(T, x)]$. The decision tree T computes a function f with distributional error at most δ if $\Pr_{x \leftarrow \nu}[T(x) = f(x)] \geq 1 - \delta$. Finally, the δ-error distributional complexity of T with respect to ν, denoted $\Delta_{\delta,\nu}(f)$, is the minimum of $C(T, \nu)$ over all T that compute f with distributional error at most δ.

Yao has shown the following:

Theorem 1 (Yao [24]). *For any distribution ν and a function f, $R_0(f) \geq \Delta_{0,\nu}(f)$ and $R(f) \geq \frac{1}{2}\Delta_{2/10,\nu}(f)$.*

In general, Yao shows that the δ-error randomized complexity of a function f is at least $\frac{1}{2}\Delta_{2\delta,\nu}(f)$, for any distribution ν. We obtain the constant $\frac{2}{10}$ on the right hand side as we have defined $R(f)$ for algorithms that err with probability at most $\frac{1}{10}$.

Quantum query complexity.

The main novelty in a quantum query algorithm is that queries can be made in superposition. For this exposition we assume $\Sigma = [|\Sigma|]$ (this identification can be made in an arbitrary way). The memory of a quantum query algorithm contains two registers, the query register H_Q which holds two integers $j \in [n]$ and $p \in \Sigma$ and the workspace H_W which holds an arbitrary value. A query on input x is encoded as a unitary operation O_x in the following way. On input x and an arbitrary basis state $|j, p\rangle |w\rangle \in H_Q \otimes H_W$,

$$O_x |j, p\rangle |w\rangle = |j, p + x_j \bmod |\Sigma|\rangle |w\rangle .$$

A quantum query algorithm begins in the initial state $|0, 0\rangle |0\rangle$ and on input x proceeds by interleaving arbitrary unitary operations independent of x and the operations O_x. The cost of the algorithm is the number of applications of O_x. The outcome of the algorithm is determined by a two-outcome measurement, specified by a complete set of projectors $\{\Pi_0, \Pi_1\}$. If $|\Psi_x\rangle$ is the final state of the algorithm on input x, the probability that the algorithm outputs 1 is $\|\Pi_1|\Psi_x\rangle\|^2$. The exact quantum query complexity of the function f, denoted $Q_E(f)$, is the minimum cost of a quantum query algorithm that outputs $f(x)$ with probability 1 for every input x. The bounded-error quantum query complexity of the function f, denoted $Q(f)$, is the minimum cost of a quantum query algorithm that outputs $f(x)$ with probability at least $9/10$ for every input x.

We will describe our quantum algorithms as classical algorithms which use the following well-known quantum algorithms as subroutines. Let O_x be a quantum oracle encoding a string $x \in \{0, 1\}^n$.

- Grover's search [12, 6]: Assume it is known that $|x| \geq t$. There is a quantum algorithm using $O(\sqrt{n/t})$ queries to O_x that finds an i such that $x_i = 1$ with probability at least $9/10$.

- Exact Grover's search [6]. Assume it is known that $|x| = t$. There is a quantum algorithm using $O(\sqrt{n/t})$ queries to O_x that finds an i such that $x_i = 1$ with certainty. The case $t = n/2$ is essentially the Deutsch-Jozsa problem [10].
- Approximate counting [7]: Let $t = |x|$. There is a quantum algorithm making $O(\sqrt{n})$ queries to O_x that outputs a number $\tilde{t}$ satisfying $|\tilde{t} - t| \le \frac{t}{10}$ with probability at least 9/10.
- Amplitude amplification [6]: Assume a quantum algorithm $\mathcal{A}$ prepares a state $|\psi\rangle = \alpha_0 |0\rangle |\psi_0\rangle + \alpha_1 |1\rangle |\psi_1\rangle$, where ψ, ψ_0 and ψ_1 are unit vectors, and α_0 and α_1 are real numbers. Thus, the success probability of $\mathcal{A}$, i.e., probability of obtaining 1 in the first register after measuring $|\psi\rangle$, is α_1^2.

  Assume a lower bound p is known on α_1^2. There exists a quantum algorithm that makes $O(1/\sqrt{p})$ calls to $\mathcal{A}$, and either fails, or generates the state $|1\rangle |\psi_1\rangle$. The success probability of the algorithm is at least 9/10.

In all of these quantum subroutines, the error probability can be reduced to ε by repeating the algorithm $O(\log \frac{1}{\varepsilon})$ times.

Polynomial degree.

Every boolean function $f\colon \{0,1\}^n \to \{0,1\}$ has a unique expansion as a multilinear polynomial $p = \sum_{S \subseteq [n]} \alpha_s \prod_{i \in S} x_i$. The *degree* of f, denoted $\deg(f)$, is the size of a largest monomial x_S in p with nonzero coefficient α_S. The *approximate degree* of f, denoted $\widetilde{\deg}(f)$, is

$$\widetilde{\deg}(f) = \min \left\{ \deg(g) \;\middle|\; |g(x) - f(x)| \le \tfrac{1}{10} \forall x \in \{0,1\}^n \right\}.$$

For any quantum algorithm that uses T queries to the quantum oracle O_x, its acceptance probability is a polynomial of degree at most $2T$ [3]. Therefore, $\deg(f) \le 2Q_E(f)$ and $\widetilde{\deg}(f) \le 2Q(f)$.

Certificate complexity.

A *partial assignment* in Σ^n is a string in $a \in (\Sigma \cup \{\star\})^n$. The length of a partial assignment is the number of non-star values. A string $x \in \Sigma^n$ is *consistent* with an assignment a if $x_i = a_i$ whenever $a_i \ne \star$. Every partial assignment defines a *subcube*, which is the set of all strings consistent with that assignment. For every subcube there is a unique partial assignment that defines it, and we define the length of a subcube as the length of this assignment.

For $b \in \{0,1\}$, a *b-certificate* for a function $f\colon \Sigma^n \to \{0,1\}$ is a partial assignment such that the value of f is b for all inputs in the associated subcube. The *b-certificate complexity* of f is the smallest number k such that the set $f^{-1}(b)$ can be written as a union of subcubes of length at most k. The *unambiguous b-certificate complexity* of f is the smallest number k such that the set $f^{-1}(b)$ can be written as a disjoint union of subcubes of length at most k.

Booleanizing a function.

While we define functions over a nonboolean alphabet Σ, it is more typical in query complexity to discuss boolean functions. Fix a surjection $b\colon \{0,1\}^{\lceil \log |\Sigma| \rceil} \to \Sigma$. For a function $f\colon \Sigma^n \to \{0,1\}$, we define the associated boolean function $\tilde{f}\colon \{0,1\}^{n\lceil \log |\Sigma| \rceil} \to \{0,1\}$ by $\tilde{f}(x) = f(b(x))$. A lower bound on f in the model where a query returns an element of Σ will also apply to $\tilde{f}$ in the model where a query returns a boolean value. Also, if f can be computed with t queries then we can convert this into an algorithm for computing $\tilde{f}$ with $t\lceil \log |\Sigma| \rceil$ queries by querying all the bits of the desired element. We will state our theorems for nonboolean functions where a query returns an element of Σ and the alphabet size $|\Sigma|$ will always be polynomial in the input length. By the remarks above, such separations can be converted into separations for the associated boolean function with a logarithmic loss.

3. SEPARATIONS AGAINST DETERMINISTIC COMPLEXITY

Let n, m, M, $\widetilde{M}$ be as in the definition of the Göös-Pitassi-Watson function. Let also $\widetilde{C} = [m] \cup \{\perp\}$ be the set of pointers to the columns of M. The input alphabet of our function is $\Sigma = \{0,1\} \times \widetilde{M} \times \widetilde{M} \times \widetilde{C}$. For $v \in \Sigma$, we call the elements of the quadruple the *value*, the *left pointer*, the *right pointer* and the *back pointer* of v, respectively. We use notation $\mathrm{val}(v)$, $\mathrm{lpoint}(v)$, $\mathrm{rpoint}(v)$, and $\mathrm{bpoint}(v)$ for them in this order.

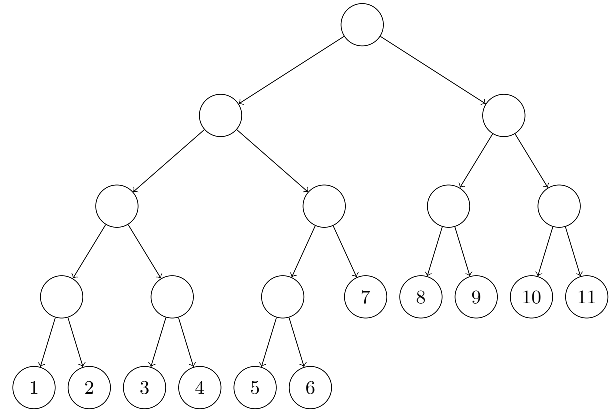

Figure 3: A balanced tree on 11 leaves.

Let T be a fixed balanced oriented binary tree with m leaves and $m-1$ internal vertices. For instance, we can make the following canonical choice. If $m = 2^k$ is a power of two, we use the completely balanced binary tree on m leaves, in which each leaf is at distance k from the root. Otherwise, assume $2^k < m < 2^{k+1}$. Take the completely balanced tree on 2^k leaves, and add a pair of children to each of its $m - 2^k$ leftmost leaves. An example is in Figure 3.

We have the following labels in T. The outgoing arcs from each node are labeled by 'left' and 'right'. The leaves of the tree are labeled by the elements of $[m]$ from left to right, with each label used exactly once. For each leaf $j \in [m]$ of the tree, the path from the root to the leaf defines a sequence of 'left' and 'right' of length $O(\log m)$, which we denote $T(j)$.

The function $f_{n,m}\colon \Sigma^M \to \{0,1\}$ is defined as follows. For an input $x = (x_{i,j})$, we have $f_{n,m}(x) = 1$ if and only if the following conditions are satisfied (for an illustration refer to Figure 4):

1. There is exactly one column $b \in [m]$ with $\mathrm{val}(x_{i,b}) = 1$ for all $i \in [n]$. We refer to it as the *marked column*.

2. In the marked column, there exists a unique cell a such that $x_a \neq (1, \bot, \bot, \bot)$. We call a the *special element*.
3. For each non-marked column $j \in [m] \setminus \{b\}$, let ℓ_j be the end of the path which starts at the special element a and follows the pointers lpoint and rpoint as specified by the sequence $T(j)$. We require that ℓ_j exists (no pointer on the path is $\bot$), ℓ_j is in the jth column, and $\text{val}(x_{\ell_j}) = 0$. We call ℓ_j the *leaves of the tree*.
4. Finally, for each non-marked column $j \in [m] \setminus \{b\}$, we require that $\text{bpoint}(x_{\ell_j}) = b$.

The values of $\text{lpoint}(x_{\ell_j})$ and $\text{rpoint}(x_{\ell_j})$ can be arbitrary.

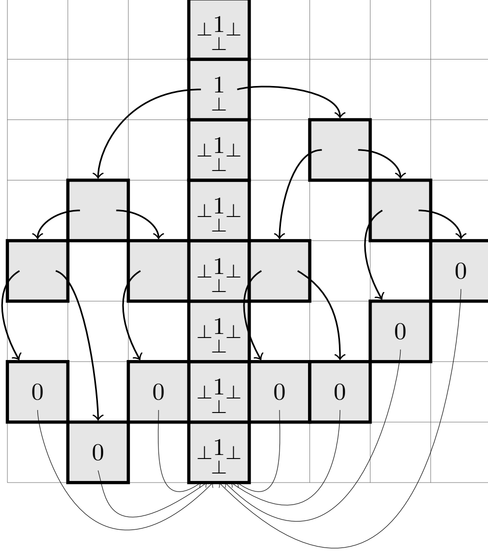

Figure 4: An example of a 1-certificate for the function $f_{8,8}$. The tree T is like in Figure 3 on the left. The center of a cell $x_{i,j}$ shows $\text{val}(x_{i,j})$, the bottom of the cell shows $\text{bpoint}(x_{i,j})$ and the bottom left and right sides show $\text{lpoint}(x_{i,j})$ and $\text{rpoint}(x_{i,j})$, respectively. Values and pointers that are not shown can be chosen arbitrarily.

THEOREM 2. *If $n = 2m$ and m is sufficiently large, the deterministic query complexity $D(f_{n,m}) \geq m^2$.*

PROOF. We describe an adversary strategy that ensures that the value of the function is undetermined after m^2 queries, provided $m \geq 4$. Assume a deterministic query algorithm queries a cell (i, j). Let k be the number of queried cells in column j, including the cell (i, j). If $k \leq m$, the adversary replies with $(1, \bot, \bot, \bot)$. Otherwise, the response is $(0, \bot, \bot, k - m)$.

Note that, after all the cells are queried in some column, it contains m cells with $(1, \bot, \bot, \bot)$ and one cell with $(0, \bot, \bot, b)$ for each $b \in [m]$.

CLAIM 3. *If there is a column $b \in [m]$ with at most m queried cells and there are at least $4m$ unqueried cells in total, then the function value is undetermined.*

PROOF. First, the adversarial strategy is such that no all-one column can ever be constructed, hence, by answering all remaining queries with value 0, the adversary can make the function evaluate to 0.

Now we show that the function value can also be set to 1. For each column $j \neq b$, define ℓ_j as follows. If column j contains an unqueried cell (i, j), let $\ell_j = (i, j)$, and assign the quadruple $(0, \bot, \bot, b)$ to this cell. If all elements in column j were queried, then, by the adversary strategy, it contains a cell with quadruple $(0, \bot, \bot, b)$. Let ℓ_j be this cell.

Next, the queried cells in column b only contain $(1, \bot, \bot, \bot)$. Assign the quadruple $(1, \bot, \bot, \bot)$ to the remaining cells in column b except for one special cell a. Using the cell a as the root construct a tree of pointers isomorphic to T using as internal vertices some of the remaining unqueried cells, and such that the jth leaf is ℓ_j. Finally, assign the quadruple $(1, \bot, \bot, \bot)$ to every other cell.

To carry out the construction above, we need $m - 2$ unqueried cells outside of column b and the set of ℓ_js to place the internal vertices of the tree. Since there are $2m$ cells in column b and $m - 1$ cells are used by ℓ_js, it suffices to have $4m$ unqueried cells to do this. ☐

It takes more than m^2 queries to ensure that each column contains more than m queried cells. As $2m^2 - 4m \geq m^2$ when $m \geq 4$, we obtain the required lower bound.

Algorithm 1 A Las Vegas randomized algorithm for the function $f_{n,m}$

VerifyColumn(j) tests whether column j is marked

1. If column j does not satisfy condition (2) of the definition of $f_{n,m}$, then reject. Otherwise, let a be the special element.
2. Following the left and right pointers from a and querying the elements along the way, check that the tree rooted at a satisfies conditions (3) and (4) of the definition of $f_{n,m}$. If it does, accept. Otherwise, reject.

TestColumn(c, k) always returns 'True' if column c has no zeroes. If it has more than $k/2$ zeroes, returns 'False' with probability $\geq 1 - 1/(nm)^2$. Returns anything in the intermediate cases.

1. Query $O(\frac{n}{k} \log(nm))$ random elements from column c. If no zero was found, return 'True'. Otherwise, return 'False'.

Main procedure of the algorithm

1. Let j be an arbitrary column in $[m]$, and $k \leftarrow n$.
2. Repeat the following actions:
   (a) Query all the elements of column j. If all of them have value 1, VerifyColumn(j).
   (b) If column j contains more than k zeroes, then query all the elements of M and output the value of the function.
   (c) Else, let C be the set of nonnull back pointers stored in the zero elements of column j. For each $c \in C$, TestColumn(c, k). If 'False' is obtained for all the columns, reject. Otherwise, let j be any column with outcome 'True'.
   (d) If $k = 0$, reject. Otherwise, let $k \leftarrow \lfloor k/2 \rfloor$, and repeat the loop.

THEOREM 4. *For the Las Vegas randomized complexity, we have $R_0(f_{n,m}) = \widetilde{O}(n+m)$.*

PROOF. For the description, see Algorithm 1. With each iteration of the loop in step 2, k gets reduced by half until it becomes zero, hence, after $O(\log n)$ iterations of the loop, the algorithm terminates.

Let us check the correctness of the algorithm. The algorithm only accepts in two places: on step 2(b), or in the VerifyColumn procedure. In both cases, the algorithm accepts only after it has found a 1-certificate, which means that it never accepts a negative input.

To see the algorithm always accepts a positive input, let the input x be positive with marked column b. Consider one iteration of the loop in step 2. If $j = b$, then the algorithm accepts in VerifyColumn(j) on step 2(a). Now assume $j \neq b$. Then, column j contains a zero with a back pointer to b, hence, the algorithm does not reject on step 2(c). The algorithm also does not reject on step 2(d) since, when $k = 0$, the condition in 2(b) applies.

Let us now estimate the expected number of queries made by the algorithm. Condition in step 2(b) is obviously not satisfied on the first iteration of the loop. On a specific later iteration, the probability this condition is satisfied is at most $1/(nm)^2$ by our definition of TestColumn(c, k). Since the loop is repeated $O(\log n)$ times, the contribution of step 2(b) to the complexity of the algorithm is $o(1)$.

If step 2(b) is not invoked, we have the following complexity estimates. VerifyColumn uses $O(m)$ queries, and it is called at most once. Apart from VerifyColumn, Step 2(a) uses n queries. Since $|C| \leq k$ on step 2(c), the number of queries in this step is $\widetilde{O}(n)$. Since there is only a logarithmic number of iterations of the loop in step 2, the total number of queries is $\widetilde{O}(n+m)$. □

Algorithm 2 A quantum algorithm for the function $f_{n,m}$

VerifyColumn(j) tests whether column j is marked

1. Use Grover's search to find an element a in column j with nonnull left or right pointer. If no element found, reject. If $\mathrm{val}(x_a) = 0$, reject.
2. Use Grover's search to verify that all elements in column j except a are equal to $(1, \perp, \perp, \perp)$. If not, reject.
3. Use Grover's search (over all $j \in [m]\setminus\{b\}$) to check that conditions (3) and (4) of the definition of $f_{n,m}$ are satisfied. If they are, accept. Otherwise, reject.

FindGoodBackPointer(j, k) if column j has $\leq \frac{11}{10}k$ zeroes and one of them has a back pointer to an all-1 column, finds a column containing $\leq k/2$ zeroes with probability $\geq 1/2k$.

1. Use Grover's search to find a zero v in column j.
2. If $\mathrm{bpoint}(x_v) = \perp$, return 'False'. Otherwise $c \leftarrow \mathrm{bpoint}(x_v)$.
3. Execute Grover's search for a zero in column c, assuming there are $\geq k/2$ of them. Return 'False' if Grover's search finds a zero, and 'True' otherwise.

Main procedure of the algorithm

1. Let j be an arbitrary column in $[m]$.
2. Repeat the following actions. If the loop does not finish after $10 \log n$ iterations, reject.
   (a) Use quantum counting to estimate the number of zeroes in column j with relative accuracy 1/10. Let k be the estimate. If $k = 0$, VerifyColumn(j).
   (b) Execute quantum amplitude amplification on the FindGoodBackPointer(j, k) subroutine amplifying for the output 'True' of the subroutine and assuming its success probability is at least $1/2k$. Let c be the corresponding value of the subroutine after amplification.
   (c) Set $j \leftarrow c$. Repeat the loop.

THEOREM 5. *The quantum query complexity $Q(f_{n,m}) = \widetilde{O}(\sqrt{n} + \sqrt{m})$.*

PROOF. The algorithm, Algorithm 2, is a quantum counterpart of Algorithm 1. We assume that every elementary quantum subroutine of the algorithm (e.g. Grover's search or quantum counting) is repeated sufficient number of times to reduce its error probability to at most $1/(nm)^2$. This requires a logarithmic number of repetitions, which can be absorbed into the $\widetilde{O}$ factor. Since the algorithm makes less than $O(n+m)$ queries, we may further assume that all the elementary quantum subroutines are performed perfectly.

The analysis is similar to Theorem 4. Again, the algorithm only accepts from VerifyColumn, which is called at most once. The three steps of VerifyColumn correspond to the three conditions defining a 1-input. Any negative input violates one of these conditions, and thus will fail one of these tests.

Now suppose we have a positive input x with marked column b. In this case, each non-marked column contains a zero with a back pointer to the marked column b. We want to argue that the algorithm accepts x with high probability. The following claim is the cornerstone of our analysis.

CLAIM 6. *If the input x is positive and column j contains at most $\frac{11}{10}k$ zeroes, then step 2(b) of the algorithm finds a column c containing at most $k/2$ zeroes with high probability.*

PROOF. We first claim that FindGoodBackPointer(j, k) returns 'True' with probability at least $1/2k$. Indeed, we assumed that the probability Grover's search on step 1 fails is negligible. Thus, with high probability, before execution of step 2, v is chosen uniformly at random from the at most $\frac{11}{10}k$ zeroes in column j. One of these zeroes has a back pointer to the marked column b. If it is chosen, step 3 returns 'True' with certainty, which proves our first claim.

Thus, amplitude amplification in step 2(b) of the main procedure will generate the 'True'-portion of the final state of the FindGoodBackPointer subroutine. Again, since we assume that the error probability of Grover's search on step 3 of FindGoodBackPointer is negligible, we may assume this portion of the state only contains columns c with at most $k/2$ zeroes. □

Consider the loop in step 2. We may assume quantum counting is correct in step 2(a). If $j = b$, then VerifyColumn(j) is called in step 2(a), and the algorithm accepts with high probability. So, consider the case $j \neq b$. Column j contains a zero, hence, with high probability, VerifyColumn is not executed on step 2(a). Thus, by Claim 6, the number of zeroes in column j gets reduced by a factor of 1.1/2. Therefore, after $10 \log n$ iterations, the number of zeroes in column j becomes zero, which means $j = b$, and the algorithm accepts with high probability.

We now estimate the complexity of the algorithm. Steps (1) and (2) of VerifyColumn take $\widetilde{O}(\sqrt{n})$ queries. For step (3) of VerifyColumn, we have to check that the tree of pointers rooted from x_a satisfies conditions (3) and (4) from the definition of $f_{n,m}$. We can check the correctness of a single path from the root to a leaf with $O(\log m)$ (classical) queries. Since there are m many paths, checking them all with Grover's search takes $\widetilde{O}(\sqrt{m})$ many queries. Overall, VerifyColumn takes $\widetilde{O}(\sqrt{n}+\sqrt{m})$ queries.

Grover's search in the FindGoodBackPointer(j,k) procedure uses $\widetilde{O}(\sqrt{n/k})$ queries. Since the success probability of FindGoodBackPointer(j,k) is at least $1/2k$, amplitude amplification repeats FindGoodBackPointer(j,k) $\widetilde{O}(\sqrt{k})$ times and the complexity of step 2(b) is $\widetilde{O}(\sqrt{n})$. Quantum counting in step 2(a) also uses $\widetilde{O}(\sqrt{n})$ queries.

Since we run the main loop at most $O(\log n)$ many times, the total complexity of the algorithm is $\widetilde{O}(\sqrt{n}+\sqrt{m})$.

Corollary 7. *There is a total boolean function f such that $R_0(f)=\widetilde{O}(D(f)^{1/2})$ and $Q(f)=\widetilde{O}(D(f)^{1/4})$.*

Proof. We first get these separations for a non-boolean function. Take $f_{n,m}$ with $n=2m$. Then the zero-error randomized query complexity is $\widetilde{O}(n)$ by Theorem 4, the quantum query complexity is $\widetilde{O}(\sqrt{n})$ by Theorem 5, and the deterministic query complexity is $\Omega(n^2)$ by Theorem 2. Since the size of the alphabet Σ is polynomial, this also gives the separations for the associated boolean function $\tilde{f}_{2m,m}$. □

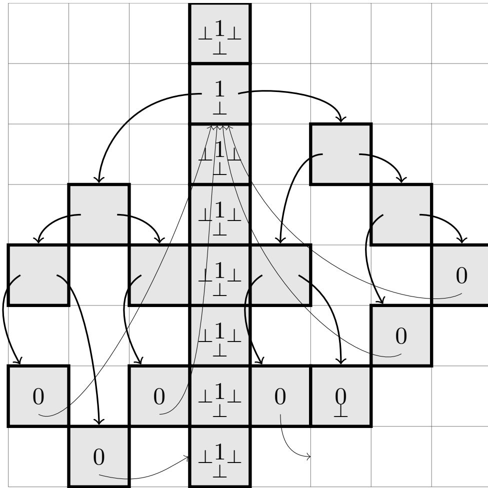

Figure 5: An example of a 1-certificate for the function $g_{8,8}$. The tree T is like in Figure 3 on the left. The center of a cell $x_{i,j}$ shows $\mathrm{val}(x_{i,j})$, the bottom of the cell shows $\mathrm{bpoint}(x_{i,j})$ and the bottom left and right sides show $\mathrm{lpoint}(x_{i,j})$ and $\mathrm{rpoint}(x_{i,j})$, respectively. Values and pointers that are not shown can be chosen arbitrarily. It is crucial that $m/2$ leaves point to the root a of the tree, and $m/2-1$ leaves point to something different.

4. SEPARATIONS AGAINST LAS VEGAS COMPLEXITY

In this section, we define a variant of the $f_{n,m}$ function from the last section. Let n, m, M, $\widetilde{M}$, T and $T(j)$ be as previously. The input alphabet is $\Sigma=\{0,1\}\times\widetilde{M}\times\widetilde{M}\times\widetilde{M}$, where we keep the names and notation of left, right and back pointers. Note that the back pointers now point to a cell of M, not a column.

Let m be even. The function $g_{n,m}\colon \Sigma^M\to\{0,1\}$ is defined like the function $f_{n,m}$ in Section 3 with condition 4 replaced by the following condition

4′ The set $G=\{j\in[m]\setminus\{b\}\mid \mathrm{bpoint}(x_{\ell_j})=a\}$ is of size exactly $m/2$.

For an illustration refer to Figure 5.

Theorem 8. *If n and m are sufficiently large, the Las Vegas randomized query complexity $R_0(g_{n,m})=\Omega(nm)$.*

Proof. We construct a hard probability distribution on negative inputs such that any Las Vegas randomized algorithm has to make $\Omega(nm)$ queries in expectation to reject an input sampled from it. Each input $x=(x_{i,j})$ in the hard distribution is specified by a function $\ell_x\colon[m]\to[n]$. The function specifies the positions of the leaves of the tree T in a possible positive instance. The definition of x is as follows

$$x_{i,j}=\begin{cases}(0,\bot,\bot,\bot), & \text{if } i=\ell_x(j);\\ (1,\bot,\bot,\bot), & \text{otherwise.}\end{cases} \tag{1}$$

The hard distribution is formed in this way from the uniform distribution on all functions ℓ_x. Thus, all pointers are null pointers, and each column contains exactly one zero element in a random position. The theorem obviously follows from the following two results. □

Claim 9. *Any Las Vegas algorithm for the function $g_{n,m}$ can reject an input x from the hard distribution (1) only if it has found at least $m/2$ zeroes or it has queried more than $n(m-1)-2m$ elements.*

Lemma 10. *Assume a Las Vegas algorithm can reject an input x from the hard distribution (1) only if it has found $\Omega(m)$ zeroes or it has queried $\Omega(nm)$ elements. Then, the query complexity of the algorithm is $\Omega(nm)$.*

Proof of Claim 9. Assume these conditions are not met. Then, we can construct a positive input y that is consistent with the answers to the queries obtained by the algorithm so far.

Let $B\subseteq[m]$ be the set of columns where no zero was found. By assumption, $|B|\ge m/2+1$. Choose an element $b\in B$ and a subset $G\subseteq B\setminus\{b\}$ of size $m/2$. Define $a=(\ell_x(b),b)$, set the value of y_a to 1 and its back pointer to $\bot$. For each column $j\in G$, define $y_{\ell_x(j),j}=(0,\bot,\bot,a)$. Finally, for the remaining columns $j\in B\setminus(G\cup\{b\})$, define $y_{\ell_x(j),j}=(0,\bot,\bot,\bot)$.

Remove from the tree T the leaf with label b. Let the resulting graph be T'. Put the root of T' into a, and, for each $j\neq b$, put the leaf of T' with label j into $(\ell_x(j),j)$. Put the remaining nodes of T' into the still unqueried cells of M preserving the structure of the graph. Set their values to 0 and their back pointers to $\bot$. Set all the remaining cells to $(1,\bot,\bot,\bot)$. The resulting input y is positive and consistent with the answers to the queries obtained by the algorithm. □

Proof of Lemma 10. By Theorem 1 it suffices to show that any deterministic algorithm $\mathcal{D}$ makes an expected $\Omega(nm)$ number of queries to find $\Omega(m)$ zeroes in an input from the hard distribution.

Consider a node S of the decision tree $\mathcal{D}$. Call a column $j \in [m]$ *compromised* in S if either a zero was found in it, or more than $n/2$ of its elements were queried. For an input x, let $A_t(x)$ be the number of compromised columns on input x after t queries. Similarly, let $B_t(x)$ be the number of queries made outside the compromised columns. Let us define

$$I_t(x) = A_t(x) + \frac{2}{n} B_t(x).$$

Note that $A_t(x)$ can only increase as t increases, whereas $B_t(x)$ can increase or decrease.

Claim 11. *For a non-negative integer t, we have*

$$\mathbb{E}_x\big[I_{t+1}(x)\big] - \mathbb{E}_x\big[I_t(x)\big] \le \frac{4}{n}, \tag{2}$$

where the expectation is over the inputs in the hard distribution.

Proof. Fix t. We say two inputs x and y are equivalent if they get to the same vertex of the decision tree after t queries. We prove that (2) holds with the expectation taken over each of the equivalency classes. Fix an equivalence class, let x be an input in the class, and (i,j) be the variable queried by $\mathcal{D}$ on the $(t+1)$st query on the input x. Note that (i,j), $A_t(x)$ and $B_t(x)$ do not depend on the choice of x.

Consider the following cases, where each case excludes the preceding ones. All expectations and probabilities are over the uniform choice of an input in the equivalence class.

- The jth column is compromised. Then $I_{t+1}(x) = I_t(x)$, and we are done.
- After the cell (i,j) is queried, more than half of the cells in the jth column have been queried. Then, $A_t(x)$ increases by 1, and $B_t(x)$ drops by $\lfloor n/2 \rfloor$. Hence, $\mathbb{E}_x\big[I_{t+1}(x)\big] \le \mathbb{E}_x\big[I_t(x)\big] + 1/n$.
- Consider the remaining case. We have $\Pr_x[i = \ell_x(j)] \le 2/n$. If $i = \ell_x(j)$, then $A_t(x)$ grows by 1, and $B_t(x)$ can only decrease. If $i \ne \ell_x(j)$, then $A_t(x)$ does not change, and $B_t(x)$ grows by 1. Thus, $\mathbb{E}_x\big[I_{t+1}(x)\big] - \mathbb{E}_x\big[I_t(x)\big] \le \frac{2}{n} + \frac{2}{n} = \frac{4}{n}$. ☐

Now we finish the proof of Lemma 10. Assume the algorithm can reject an input x only if it has found $c_1 m$ zeroes or it has queried $c_2 nm$ elements for some constants $c_1, c_2 > 0$.

Let $t = \lfloor c_1 nm/8 \rfloor$. Clearly, $\mathbb{E}_x[I_0(x)] = 0$ for all x. Claim 11 implies $\mathbb{E}_x[I_t(x)] \le c_1 m/2$. By Markov's inequality, $\Pr_x\big[I_t(x) \ge c_1 m\big] \le 1/2$. By our assumption, the probability the algorithm $\mathcal{D}$ has not rejected x after $t' = \min\{t,\ c_2 nm\}$ queries is at least $1/2$. Hence, the expected number of queries made by the algorithm is at least $t'/2 = \Omega(nm)$.

Theorem 12. *For one-sided error randomized and exact quantum query complexity, we have $R_1(g_{n,m}) = \widetilde{O}(n+m)$ and $Q_E(g_{n,m}) = \widetilde{O}(n+m)$.*

Proof. We reduce the problem to the task of evaluating the partial function h on m bits defined by the following relations: $h(0^m) = 0$, and $h(z) = 1$ whenever $|z| = m/2$. This function can be evaluated in $O(1)$ queries both by a randomized algorithm with 1-sided error and by an exact quantum algorithm.

For a positive input x, we call a column $j \in [m]$ *good* iff it belongs to the set G from Condition $4'$ on Page . Thus, a positive input has exactly $m/2$ good columns, whereas a negative one has none. Theorem 12 follows immediately from the following lemma.

Lemma 13. *There exists a deterministic subroutine that, given an index $j \in [m]$, accepts iff the column j is good in $\widetilde{O}(n+m)$ queries.*

Indeed, we use the subroutine from Lemma 13 as the input to the algorithm evaluating the function h. For example, for an algorithm with one-sided error, we choose an index $j \in [m]$ uniformly at random, and execute the subroutine of Lemma 13. If the input is negative, we always reject. If the input is positive, we accept with probability exactly $1/2$. The exact quantum algorithm is obtained similarly, using the Deutsch-Jozsa algorithm. ☐

Proof of Lemma 13. The subroutine is described in Algorithm 3. In the subroutine, I stores the set of (the first indices) of the cells in column j that can potentially contain the element ℓ_j back pointing to the special element a. The set B contains potentially marked columns.

Algorithm 3 A deterministic subroutine testing whether a column j is good

1. Let $I \leftarrow [n]$ and $B \leftarrow [m]$.
2. While $I \ne \emptyset$ and $|B| \ge 2$, repeat the following:
   (a) Let i be the smallest element of I. Let $a \leftarrow \mathrm{bpoint}(x_{i,j})$. If $a = \perp$, remove i from I, and continue with the next iteration of the loop.
   (b) Let j be the smallest number of a column in B that does not contain a. Follow the pointers from a as specified by the sequence $T(j)$. Let ℓ_j be the endpoint.
   (c) If ℓ_j exists, is located in column j and its value is 0, remove j from B. Otherwise, remove i from I.
3. If $|B| \ge 2$, reject. Otherwise, let b be the only element of B. Verify column b using a procedure similar to that in Algorithm 1.
4. If column b passes the verification, and j belongs to the set G from Condition $4'$ on Page , accept. Otherwise, reject.

As ensured by step 4, the subroutine only accepts if the input is positive (column b passes the verification), and the column j is good. Hence, we get no false positives. On the other hand, assume the input is positive with the marked column b, column j is good, and $\ell_j = (i,j)$. In this case, i cannot get removed from I due to goodness of column j and Condition 3 on Page , and b never gets eliminated from B as it contains no zeroes. Thus, the only possibility to exit the loop on step 2 is to have $B = \{b\}$. In this case, b passes the verification, and the algorithm accepts since column j is good. Hence, we get no false negatives as well.

The query complexity of each iteration of the loop in step 2 is $O(\log m)$. Also, with each iteration, either I or B get reduced by one element. Hence, the total number of

iterations of the loop does not exceed $n + m$. Finally, the verification in step 3 requires $O(n + m)$ queries. Thus, the query complexity of the algorithm is $\widetilde{O}(n + m)$. $\square$

Corollary 14. *There is a total boolean function f with $R_1(f) = \widetilde{O}(R_0(f)^{1/2})$ and $Q_E(f) = \widetilde{O}(R_0(f)^{1/2})$.*

Proof. We first obtain this separation for a non-boolean function. Take $g_{n,m}$ with $n = m$. The one-sided error randomized and exact quantum query complexity is $\widetilde{O}(n)$ by Theorem 12, and the Las Vegas query complexity is $\Omega(n^2)$ by Theorem 8. Since the size of the alphabet Σ is polynomial, this also gives the separations for the associated boolean function $\tilde{g}_{n,n}$. $\square$

5. OTHER SEPARATIONS AGAINST RANDOMIZED COMPLEXITY

In this section we define another modification of the function used in Section 3. Let n, m, M, $\widetilde{M}$ and T be as in Section 3, and let $k : 1 \le k < m$ be an integer. The new function $h_{k,n,m} \colon \Sigma^M \to \{0,1\}$ is defined as follows. The input alphabet is $\Sigma = \{0,1\} \times \widetilde{M} \times \widetilde{M} \times \widetilde{M}$. For $v \in \Sigma$, we call the elements of the quadruple the *value*, the *left pointer*, the *right pointer* and the *internal pointer* of $x_{i,j}$, respectively. We use notation $\mathrm{val}(v)$, $\mathrm{lpoint}(v)$, $\mathrm{rpoint}(v)$, and $\mathrm{ipoint}(v)$ for them in this order.

For an input $x = (x_{i,j})$, we have $h_{k,n,m}(x) = 1$ if an only if the following conditions are satisfied (for an illustration refer to Figure 6):

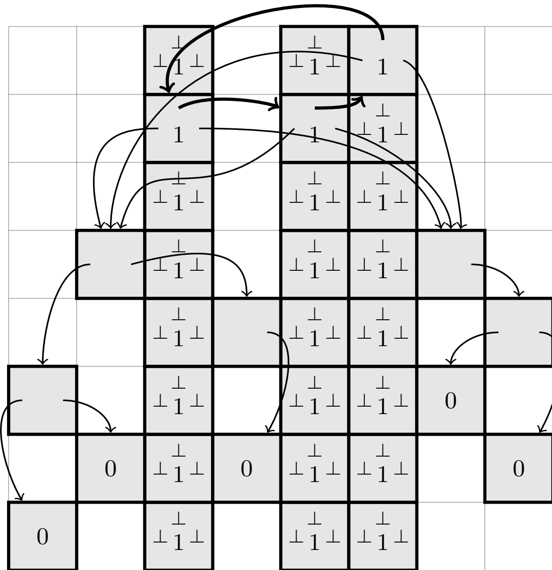

Figure 6: An example of a 1-certificate for the $h_{3,8,8}$ function. The tree T is like in Figure 3 on the left. The center of a cell $x_{i,j}$ shows $\mathrm{val}(x_{i,j})$, the top of the cell shows $\mathrm{ipoint}(x_{i,j})$ and the left and right sides show $\mathrm{lpoint}(x_{i,j})$ and $\mathrm{rpoint}(x_{i,j})$, respectively. Values and pointers that are not shown can be chosen arbitrarily.

1. There are exactly k columns $b_1, \ldots, b_k$ with $\mathrm{val}(x_{i,b_s}) = 1$ for all $i \in [n]$ and each $s \in [k]$. We refer to these as the *marked columns*.
2. Each marked column b_s contains a unique cell a_s such that $x_{a_s} \neq (1, \bot, \bot, \bot)$. We call a_s a *special element*.
3. We have $\mathrm{ipoint}(x_{a_s}) = a_{s+1}$ for all $s \in [k-1]$, and $\mathrm{ipoint}(x_{a_k}) = a_1$. Also, $\mathrm{lpoint}(x_{a_s}) = \mathrm{lpoint}(x_{a_t})$ and $\mathrm{rpoint}(x_{a_s}) = \mathrm{rpoint}(x_{a_t})$ for all $s, t \in [k]$.
4. For each non-marked column $j \in [m] \setminus \{b_1, \ldots, b_k\}$, let ℓ_j be the end of the path which starts at a special element a_s (whose choice is irrelevant) and follows the pointers lpoint and rpoint as specified by the sequence $T(j)$. We require that ℓ_j exists (no pointer on the path is $\bot$), ℓ_j is in the jth column, and $\mathrm{val}(x_{\ell_j}) = 0$.

We use this function in two different modes. The first one is the $k = 1$ case. Then, $h_{1,n,m}$ is essentially the $f_{n,m}$ function from Section 3 with the back pointers removed, i.e., it need not satisfy condition (4). In this mode, the function is hard for a Monte Carlo algorithm, but still has low approximate polynomial degree.

The second mode is the general k case. In this mode, the function is hard for a Las Vegas algorithm, but feasible for quantum algorithms. The proof of the lower bound is similar to Section 4. We need a different function because Algorithm 3 in Theorem 12 cannot be efficiently quantized.

Theorem 15. *If n and m are sufficiently large, the Las Vegas randomized query complexity $R_0(h_{k,n,m}) = \Omega(nm)$ for any $k < m/2$.*

Proof. The proof is similar to the proof of Theorem 8. We define the hard distribution (1) in exactly the same way. The theorem follows from Lemma 10 and the following claim. $\square$

Claim 16. *Any Las Vegas algorithm evaluating the function $h_{k,n,m}$ can reject an input x from the hard distribution (1) only if it has found $m - k + 1$ zeroes or it has queried more than $n(m-k) - 2m$ elements.*

Proof. Assume these conditions are not met. Then, we can construct a positive input y that is consistent with the answers to the queries obtained by the algorithm so far.

Indeed, choose a set $B = \{b_1, \ldots, b_k\}$ of columns where no zero was found. Define $a_s = (\ell_x(b_s), b_s)$ for each $s \in [k]$. These elements have not been queried yet. Define $\mathrm{val}(y_{a_s}) = 1$ for all s, as well as $\mathrm{ipoint}(y_{a_s}) = a_{s+1}$ for all $s \in [k-1]$, and $\mathrm{ipoint}(y_{a_k}) = a_1$.

Remove from the tree T the leaves with labels in B and the root. Let the resulting graph be T'. For each $j \notin B$, put the leaf of T' with label j into $(\ell_x(j), j)$, set its value to 0 and all pointers to $\bot$. Put the remaining nodes of T' into the still unqueried cells of M preserving the structure of the graph. Set their value to 0 and their internal pointers to $\bot$. Let u and v be the cells where the left and the right child of the root of T went. For each $s \in [k]$, set $\mathrm{lpoint}(y_{a_s}) = u$ and $\mathrm{rpoint}(y_{a_s}) = v$. Set all the remaining cells to $(1, \bot, \bot, \bot)$. The resulting input is positive, and consistent with the answers to the queries obtained by the algorithm. $\square$

Theorem 17. *If n and m are sufficiently large, the randomized query complexity $R(h_{1,n,m}) = \Omega\left(\frac{nm}{\log m}\right)$.*

Note that Theorem 17 only considers the case $k = 1$. It is proven in a similar fashion to Theorem 15 using the additional fact that the expected size of a subtree rooted in a node of a balanced tree is logarithmic. The proof is given in Section 6.

5.1 Quantum versus Las Vegas

THEOREM 18. *For the quantum query complexity, we have* $Q(h_{k,n,m}) = \widetilde{O}\big(\sqrt{nm/k} + \sqrt{kn} + k + \sqrt{m}\big)$.

PROOF. We search for a column consisting only of ones using Grover's search. Testing one column takes $O(\sqrt{n})$ queries. Also, in the positive case, there are k such columns, so we will find one after $O(\sqrt{nm/k})$ queries with high probability. If we do not find such a column, we reject.

In case we find a marked column j, we use Grover's search to check that it satisfies condition (2) of the definition of $h_{k,n,m}$. This requires $O(\sqrt{n})$ queries. Let a be the corresponding special element. We follow the internal pointer from a to find all the special elements and to check that condition (3) of the definition of $h_{k,n,m}$ is satisfied. This requires k queries. We use Grover's search to check that all the remaining elements of the marked columns are equal to $(1, \perp, \perp, \perp)$. This requires $O(\sqrt{kn})$ queries.

After that, we check condition (4) of the definition of $h_{k,n,m}$. Since there are less than m elements ℓ_j to check and each one can be tested in $O(\log m)$ queries, Grover's search can check this condition in $\widetilde{O}(\sqrt{m})$ queries. □

COROLLARY 19. *There is a total boolean function f with* $Q(f) = \widetilde{O}(R_0(f)^{1/3})$.

PROOF. We first obtain the separation for a non-boolean function. Take $h_{k,n,m}$ with $k = n$ and $m = n^2$. Then the quantum complexity is $\widetilde{O}(n)$ by Theorem 18, and the Las Vegas randomized complexity is $\Omega(n^3)$ by Theorem 15. Since the size of the alphabet Σ is polynomial, we obtain the required separations for the associated boolean function. □

5.2 Exact Quantum versus Monte Carlo

THEOREM 20. *For the exact quantum query complexity, we have* $Q_E(h_{k,n,m}) = O(n\sqrt{m/k} + kn + m)$.

PROOF. We use the exact version of Grover's search to find a column consisting only of ones. In the positive case, there are exactly k such columns, and testing each column takes n queries. Thus, the complexity of this step is $O(n\sqrt{m/k})$.

If we find a marked column j, we query all $x_{i,j}$ for $i \in [n]$ to check that it satisfies condition (2) of the definition of $h_{k,n,m}$. Let a be the corresponding special element. We follow the internal pointer from a to check that condition (3) is also satisfied and to find all the marked columns. We then check condition (2) on them as well. All this requires kn queries.

After that, we follow the left and right pointers from a, and check that condition (4) of the definition of $h_{k,n,m}$ is satisfied. This requires $O(m)$ queries. □

COROLLARY 21. *There exists a total boolean function f with* $Q_E(f) = \widetilde{O}(R(f)^{2/3})$.

PROOF. Take $h_{1,n,m}$ with $m = n^2$. Then the exact quantum query complexity is $O(n^2)$ by Theorem 20 and the Monte Carlo randomized query complexity is $\widetilde{\Omega}(n^3)$ by Theorem 17. Since the size of the alphabet Σ is polynomial, this also gives the separation for the associated boolean function $\tilde{h}_{1,n,m}$. □

5.3 Approximate Polynomial Degree versus Monte Carlo

THEOREM 22. *Let* $\tilde{h}_{1,n,m}\colon \{0,1\}^{nm\lceil \log|\Sigma|\rceil} \to \{0,1\}$ *be the boolean function associated to* $h_{1,n,m}$. *The approximate polynomial degree* $\widetilde{\deg}(\tilde{h}_{1,n,m}) = \widetilde{O}(\sqrt{n} + \sqrt{m})$

PROOF. For $j \in [m]$, let $g_j\colon \{0,1\}^{nm\lceil \log|\Sigma|\rceil} \to \{0,1\}$ be defined as follows. The value $g_j(x)$ is 1 if $\tilde{h}_{1,n,m}(x) = 1$, and j is the marked column. Otherwise, $g_j(x) = 0$.

For each $j \in [m]$, the function $g_j(x)$ can be evaluated in $\widetilde{O}(\sqrt{n} + \sqrt{m})$ quantum queries using a variant of the Verify-Column procedure in Algorithm 2. Repeating this quantum algorithm $O(\log m)$ times, we may assume that its error probability is at most $1/(10m)$. We then use the connection between quantum query algorithms and polynomial degree of [3] to construct a polynomial $p_j(x)$ of degree $2T$ (where T is the number of queries) that is equal to the acceptance probability of this algorithm. The polynomial $p_j(x)$ is of degree $\widetilde{O}(\sqrt{n} + \sqrt{m})$ and satisfies $0 \le p_j(x) \le 1/(10m)$ if $g_j(x) = 0$, and $1 - 1/(10m) \le p_j(x) \le 1$ otherwise.

We then define a polynomial $p(x) = \sum_j p_j(x)$ as follows. If $h_{1,n,m}(x) = 0$, then all $g_j(x) = 0$ and $0 \le p(x) \le 1/10$. Otherwise, there is unique $b \in [m]$ such that $g_b(x) = 1$, and all other $g_j(x) = 0$. In this case, $1 - 1/(10m) \le p(x) \le 11/10$. Thus, $p(x)$ is an approximating polynomial to $h_{1,n,m}$ and its degree is $\widetilde{O}(\sqrt{n} + \sqrt{m})$. □

COROLLARY 23. *There is a total boolean function f with* $\widetilde{\deg}(f) = \widetilde{O}(R(f)^{1/4})$.

PROOF. Take $\tilde{h}_{1,n,m}$ from Theorem 22 with $m = n$. Then the approximate degree is $\widetilde{O}(\sqrt{n})$, and the Monte Carlo randomized query complexity is $\Omega(n^2)$ by Theorem 17. □

6. PROOF OF THEOREM 17

LEMMA 24. *Assume T is a balanced binary tree with m leaves, and at least a $1/4$ fraction of its nodes are marked. Let u be sampled from all the marked nodes of T uniformly at random. The expected size of the subtree rooted at u does not exceed $C_0 \log m$ for some constant C_0.*

PROOF. Let us first consider the case when T is a complete balanced binary tree with $2^k - 1$ nodes and all its nodes are marked. Then, the expected size of the subtree is (where i is the height of the node u)

$$\sum_{i=1}^{k} \frac{2^{k-i}}{2^k - 1} \cdot (2^i - 1) \le k. \tag{3}$$

In the general case, T can be embedded into a complete balanced binary tree T' with $2^k - 1$ nodes, where $k = \lceil \log m \rceil + 1$. Mark in T' all the nodes marked in T. Again, an $\Omega(1)$ fraction of the nodes is marked. Hence, for each node u, a probability that u is sampled from the marked nodes of T' is at most a constant times its probability to be sampled from all the nodes of T'. Thus, the expected size of the subtree is at most a constant times the value in (3). □

By Theorem 1, it suffices to construct a hard distribution on inputs and show that any deterministic decision tree that computes $h_{1,n,m}$ with distributional error less than $2/10$ on the hard distribution makes an $\Omega(nm/\log m)$ expected

number of queries. By Markov's inequality, it suffices to show that any deterministic decision tree that performs this task with error 3/8 has *depth* $\Omega(nm/\log m)$. We now define the hard distribution.

Let T be the balanced binary tree from the definition of $h_{1,n,m}$. Denote by r the root of the tree, and by T^{N} the set of internal nodes of T. The latter has cardinality $m-1$.

An input $x = (x_{i,j})$ is defined by a quadruple $(v_x, \pi_x, \ell_x^{\mathrm{L}}, \ell_x^{\mathrm{N}})$, where

- $v_x \in \{0,1\}$, it will be the value of the function $h_{1,n,m}$ on x;
- $\pi_x \colon T^{\mathrm{N}} \to [m]$ is an injection, it specifies to which columns the internal nodes of T will go; and
- $\ell_x^{\mathrm{L}} \colon [m] \to [n]$ and $\ell_x^{\mathrm{N}} \colon [m] \to [n]$ are functions satisfying $\ell_x^{\mathrm{L}}(j) \neq \ell_x^{\mathrm{N}}(j)$ for each $j \in [m]$. They specify the rows where the leaves and the internal nodes of the tree T land in column j.

The definition is as follows. Remove the leaf with the label $\pi_x(r)$ from T. For each column $j \neq \pi_x(r)$, put the leaf j into the cell $\big(\ell_x^{\mathrm{L}}(j), j\big)$, set all its pointers to $\perp$ and its value to 0. Then, for each internal node u of the tree, put it at $\big(\ell_x^{\mathrm{N}}(\pi_x(u)), \pi_x(u)\big)$, set its left and right pointers so that the structure of the tree is preserved. If $u \neq r$, assign its internal pointer to $\perp$, and set its value to 0. Otherwise, if $u = r$, assign its internal pointer to itself, and set its value to v_x. Assign the quadruple $(1, \perp, \perp, \perp)$ to all other cells. It is easy to see that the value of the function $h_{1,n,m}$ on this input is v_x.

The hard distribution is defined as the uniform distribution over the quadruples $(v_x, \pi_x, \ell_x^{\mathrm{L}}, \ell_x^{\mathrm{N}})$ subject to the constraint $\ell_x^{\mathrm{L}}(j) \neq \ell_x^{\mathrm{N}}(j)$ for each $j \in [m]$.

Let $\mathcal{D}$ be a deterministic decision tree of depth

$$D = \frac{nm}{64 C_0 \log m}.$$

We will prove that $\mathcal{D}$ errs on x, sampled from the hard distribution, with probability at least 3/8.

Let x be an input from the hard distribution, and consider the vertex S of $\mathcal{D}$ after t queries to x. We say that a column $j \in [m]$ and the corresponding tree element $\pi_x^{-1}(j)$ (if it exists) are *compromised* on the input x after t queries if at least one of the following three conditions is satisfied:

- one of the cells $(\ell_x^{\mathrm{L}}(j), j)$ and $(\ell_x^{\mathrm{N}}(j), j)$ has been queried;
- more than a half of the cells in the jth column have been queried; or
- in the tree T there exists an ancestor u of $\pi_x^{-1}(j)$ such that one of the above two conditions is satisfied for $\pi_x(u)$.

Let $A_t(x)$ denote the number of compromised columns, and $B_t(x)$ denote the number of cells queried outside the compromised columns, both after t queries. Consider the following quantity

$$I_t(x) = \min\left\{ A_t(x) + \frac{4C_0 \log m}{n} B_t(x),\ \ \frac{m}{2} \right\}.$$

Note that $A_t(x)$ can only increase as t increases, whereas $B_t(x)$ can increase or decrease.

Claim 25. *For a non-negative integer t, we have*

$$\mathbb{E}_x\big[I_{t+1}(x)\big] - \mathbb{E}_x\big[I_t(x)\big] \le \frac{8C_0 \log m}{n}, \tag{4}$$

where the expectation is over the inputs in the hard distribution.

We will prove Claim 25 a bit later. Now let us show how it implies the theorem. Clearly, $I_0(x) = 0$ for all x. Claim 25 implies that $\mathbb{E}_x[I_D(x)] \le m/8$. By Markov's inequality, the probability that $I_D(x) \ge m/2$ is at most 1/4.

Let x be an input satisfying $I_D(x) < m/2$. Thus, x has less than $m/2$ compromised columns after D queries. In particular, the variable $a = \big(\ell_x^{\mathrm{N}}(\pi_x(r)), \pi_x(r)\big)$, corresponding to the root of T, has not been queried. Let y be the input given by $(1 - v_x, \pi_x, \ell_x^{\mathrm{L}}, \ell_x^{\mathrm{N}})$. They only differ in a, and $h_{1,n,m}(x) \neq h_{1,n,m}(y)$. Hence, the decision tree $\mathcal{D}$ errs on exactly one of them.

This means that $\mathcal{D}$ errs on x sampled from the hard distribution with probability at least 3/8.

Proof of Claim 25. We divide the inputs of the hard distribution into equivalence classes, and prove that (4) holds with the expectation over each of the classes. We say that two inputs x and y are equivalent if the following three conditions hold:

- after t queries, the decision tree $\mathcal{D}$ gets to the same vertex on x and y;
- the set C of compromised columns is the same in x and y;
- for all $j \in C$, $\pi_x^{-1}(j) = \pi_y^{-1}(j)$.

Fix an equivalence class, let x be an input in the class, and (i,j) be the variable queried by $\mathcal{D}$ on the $(t+1)$st query on the input x. Note that (i,j), as well as $A_t(x)$ and $B_t(x)$ do not depend on the choice of x. Consider the following cases, where each case excludes the preceding ones. All expectations and probabilities are over the uniform choice of an input in the equivalence class.

- We have $I_t(x) = m/2$. Then, $I_{t+1}(x) \le m/2$, and we are done.
- The jth column is compromised. Then $I_{t+1}(x) = I_t(x)$, and we are done.
- After the cell (i,j) is queried, more than half of the cells in the jth column have been queried. As $I_t(x) < m/2$, less than half of the columns are compromised. By Lemma 24 with non-compromised nodes marked, the expected growth of $A_t(x)$ is at most $C_0 \log m$. On the other hand, the drop in $B_t(x)$ is at least $\lfloor n/2 \rfloor$. Hence, $\mathbb{E}_x\big[I_{t+1}(x)\big] \le \mathbb{E}_x\big[I_t(x)\big]$.
- Consider the remaining case. We have
$$\Pr_x\big[i \in \big\{\ell_x^{\mathrm{L}}(j), \ell_x^{\mathrm{N}}(j)\big\}\big] \le 4/n.$$
  - If i is one of $\ell_x^{\mathrm{L}}(j)$ or $\ell_x^{\mathrm{N}}(j)$, then, as in the previous case, the expected growth of $A_t(x)$ is at most $C_0 \log m$, and $B_t(x)$ can only decrease.
  - If $i \neq a_x(j)$, then $A_t(x)$ does not change, and $B_t(x)$ grows by 1.

Thus,

$$\begin{aligned}\mathbb{E}_x\big[I_{t+1}(x)\big] - \mathbb{E}_x\big[I_t(x)\big] &\le \frac{4}{n} \cdot C_0 \log m + \frac{4C_0 \log m}{n} \\ &= \frac{8C_0 \log m}{n}. \quad \square\end{aligned}$$

7. ACKNOWLEDGMENTS

This research is partially funded by the Singapore Ministry of Education and the National Research Foundation, also through NRF RF Award No. NRF-NRFF2013-13, and the Tier 3 Grant "Random numbers from quantum processes," MOE2012-T3-1-009. This research is also partially supported by the European Commission IST STREP project Quantum Algorithms (QALGO) 600700, by the ERC Advanced Grant MQC, Latvian State Research Programme NeXIT project No. 1, by the project 271/2012 from the Latvian Council of Science, and by the French ANR Blanc program under contract ANR-12-BS02-005 (RDAM project).

This work was done while A.B. was at the University of Latvia. Part of it was also done while A.B. was at Centre for Quantum Technologies, Singapore. A.B. thanks Miklos Santha for hospitality.

8. REFERENCES

[1] S. Aaronson and A. Ambainis. Forrelation: A problem that optimally separates quantum from classical computing. In *Proc. of 47th ACM STOC*, pages 307–316, 2015.

[2] A. Ambainis. Superlinear advantage for exact quantum algorithms. In *Proc. of 45th ACM STOC*, pages 891–900, 2013.

[3] R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf. Quantum lower bounds by polynomials. *Journal of the ACM*, 48(4):778–797, 2001. arXiv:quant-ph/9802049.

[4] S. Ben-David. A super-Grover separation between randomized and quantum query complexities. arXiv:1506.08106, 2015.

[5] M. Blum and R. Impagliazzo. Generic oracles and oracle classes. In *Proc. of 28th IEEE FOCS*, pages 118–126, 1987.

[6] G. Brassard, P. Høyer, M. Mosca, and A. Tapp. Quantum amplitude amplification and estimation. In *Quantum Computation and Quantum Information: A Millennium Volume*, volume 305 of *AMS Contemporary Mathematics Series*, pages 53–74, 2002.

[7] G. Brassard, P. Høyer, and A. Tapp. Quantum counting. In *Proc. of 25th ICALP*, volume 1443 of *LNCS*, pages 820–831. Springer, 1998. arXiv:quant-ph/9805082.

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

[9] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 454(1969):339–354, 1998.

[10] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proc. of the Royal Society London A*, 439:553–558, 1992.

[11] M. Göös, T. Pitassi, and T. Watson. Deterministic communication vs. partition number. In *Proc. of 56th IEEE FOCS*, pages 1077–1088, 2015.

[12] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. of 28th ACM STOC*, pages 212–219, 1996.

[13] J. Hartmanis and L. A. Hemachandra. One-way functions, robustness, and non-isomorphism of NP-complete sets. In *Proceedings of 2nd Structure in Complexity Theory*, pages 160–173, 1987.

[14] R. Kulkarni and A. Tal. On fractional block sensitivity. ECCC:2013/168, 2013.

[15] G. Midrijānis. Exact quantum query complexity for total boolean functions. arXiv:quant-ph/0403168, 2004.

[16] G. Midrijānis. On randomized and quantum query complexities. arXiv:quant-ph/0501142, 2005.

[17] S. Mukhopadhyay and S. Sanyal. Towards better separation between deterministic and randomized query complexity. arXiv:1506.06399, 2015.

[18] N. Nisan. CREW PRAMs and decision trees. *SIAM Journal on Computing*, 20(6):999–1007, 1991.

[19] N. Nisan and M. Szegedy. On the degree of boolean functions as real polynomials. *Computational Complexity*, 4(4):301–313, 1994.

[20] M. Saks and A. Wigderson. Probabilistic Boolean decision trees and the complexity of evaluating game trees. In *Proc. of 27th IEEE FOCS*, pages 29–38, 1986.

[21] M. Santha. On the Monte Carlo boolean decision tree complexity of read-once formulae. *Random Structures and Algorithms*, 6(1):75–87, 1995.

[22] M. Snir. Lower bounds for probabilistic linear decision trees. *Theoretical Computer Science*, 38:69–82, 1985.

[23] G. Tardos. Query complexity or why is it difficult to separate $\mathbf{NP}^A \cap \mathbf{coNP}^A$ from $\mathbf{P}^A$ by a random oracle. *Combinatorica*, 9:385–392, 1990.

[24] A. C. Yao. Probabilistic computations: toward a unified measure of complexity. In *Proc. of 18th IEEE FOCS*, pages 222–227, 1977.