Towards Better Separation between Deterministic and Randomized Query Complexity[*]

Sagnik Mukhopadhyay and Swagato Sanyal

Tata Institute of Fundamental Research
Mumbai, India
`sagnik@tifr.res.in, swagatos@tcs.tifr.res.in`

Abstract

We show that there exists a Boolean function F which gives the following separations among deterministic query complexity ($D(F)$), randomized zero error query complexity ($R_0(F)$) and randomized one-sided error query complexity ($R_1(F)$): $R_1(F) = \widetilde{O}(\sqrt{D(F)})$ and $R_0(F) = \widetilde{O}(D(F))^{3/4}$. This refutes the conjecture made by Saks and Wigderson that for any Boolean function f, $R_0(f) = \Omega(D(f))^{0.753..}$. This also shows widest separation between $R_1(f)$ and $D(f)$ for any Boolean function. The function F was defined by Göös, Pitassi and Watson who studied it for showing a separation between deterministic decision tree complexity and unambiguous non-deterministic decision tree complexity. Independently of us, Ambainis *et al* proved that different variants of the function F certify optimal (quadratic) separation between $D(f)$ and $R_0(f)$, and polynomial separation between $R_0(f)$ and $R_1(f)$. Viewed as separation results, our results are subsumed by those of Ambainis *et al.* However, while the functions considered in the work of Ambainis *et al* are different variants of F, in this work we show that the original function F itself is sufficient to refute the Saks-Wigderson conjecture and obtain widest possible separation between the deterministic and one-sided error randomized query complexity.

1998 ACM Subject Classification F.1.1 [Computation by Abstract Devices]: Models of Computation – Relations between models, F.1.2 [Computation by Abstract Devices]: Modes of Computation – Probabilistic computation

Keywords and phrases Deterministic Decision Tree, Randomized Decision Tree, Query Complexity, Models of Computation

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2015.206

1 Introduction

In computational complexity theory, one major goal is to prove limitations of existing computational models which helps us to understand the computational power that each model exhibits. Among the vast array of computational models that are studied in the literature, one of the simplest is query model (or decision tree model) where an algorithm computing a boolean function is given query access to the input. The algorithm queries different bits of the input, possibly in adaptive fashion, and computes the function on the input based on the query responses. The algorithm is charged not for the computation but for the number of bits it queries. It is easy to see that n is a trivial upper bound on the number of queries that any algorithm makes to evaluate the function where n is the input size. The objective is to minimize the number of queries made.

[*] S. Mukhopadhyay is supported by a TCS fellowship and S. Sanyal was supported by a DAE fellowship.

For a Boolean function f, the deterministic query complexity, $D(f)$, of f is defined to be the maximum (over inputs) number of queries the best deterministic query algorithm for f makes. For many well studied boolean functions, such as parity, threshold functions, the deterministic decision tree complexity is n - such functions are called *evasive* functions. As observed by Rivest and Vuillemin [7], most boolean functions are evasive. In this work we are mainly concerned about the power of the query model when we allow randomness. We want to ask the following question: how many queries can we save for evaluating f if we allow the query algorithm to toss coins. A randomized query algorithm can be thought of as a distribution over deterministic query algorithms. It can also be viewed as a query algorithm where each node has an additional power of tossing coins . After querying the variable associated with any internal node of the tree, the algorithm decides which input bit to query depending on the responses to the queries so far (i.e. the current node in the tree) and the value of the coin tosses while in that node. It is not hard to see that the two definitions are equivalent. We look at the following complexity measures that are well studied. The bounded-error randomized query complexity $R(f)$ of f is defined to be the number of queries made on the worst input by the best randomized query algorithm for f that is correct with probability $2/3$ on every input. The zero error randomized query complexity of f, $R_0(f)$, is the expected number of queries made on the worst input by the best randomized algorithm for f that gives correct answer on each input with probability 1. Finally the one-sided randomized query complexity of f, $R_1(f)$, is the number of queries made on the worst input by the best algorithm that is correct on every input with probability at least $2/3$, and in addition correct on every 0-input with probability 1. The choice of the constant $2/3$ in our definitions is arbitrary, and could be replaced by any constant in the definition of $R_1(f)$, and any constant greater than $1/2$ in the definition of $R(f)$, while changing the number of queries by a constant factor.

Relationships between these query complexity measures have been extensively studied in the literature. That randomization can save more than a constant factor of queries has been known for a long time. Snir [10] showed a $O(n^{\log_4 3})$ randomized linear query algorithm (a more powerful model than what we discussed) for complete binary NAND tree function for which the deterministic linear query complexity is $\Omega(n)$. Later on Saks and Wigderson [8] gave a $\Theta(n^{0.753\ldots})$ randomized query algorithm for the same function. They also showed that for uniform rooted ternary majority tree function, the randomized query complexity is $O(n^{0.893\ldots})$ and deterministic query complexity is $\Omega(n)$ - the authors credited R. Boppana for this example. All these example showed that randomized query complexity can be substantially lower than its deterministic counterpart.
In their paper, Saks and Wigderson made the following conjecture.

▶ **Conjecture 1** (Saks and Wigderson [8]). *For any boolean function f, $R_0(f) = \Omega(D(f)^{0.753\ldots})$.*

Saks and Wigderson conjectured that the complete binary NAND tree function exhibits the widest separation possible between these two measures of complexity. During this work, the best separation known between deterministic decision tree complexity and zero error, one-sided error and bounded error randomized query complexities was the one exhibited by the complete binary NAND tree function. Also, no separation among the different randomized query complexity measures was known.

For complete binary NAND tree function F, Santha [9] showed that $R(F) = (1-2\epsilon)R_0(F)$ where ϵ is the error probability. So, for this function, we have $R(F) = \Theta(D(F)^{0.753\ldots})$. It is easy to see that $D(f) \geq R(f), R_0(f), R_1(f)$. Blum and Impagliazzo [3], Tardos [11] and Hartmanis and Hemachandra [5] independently showed that $R_0(f) \geq \sqrt{D(f)}$. Nisan [6] showed that for any Boolean function f, $R(f) \geq \sqrt[3]{D(f)}/27$ and $R_1(f) \geq \sqrt{D(f)}$. The biggest gap known so far between $D(f)$ and $R(f)$ for any f is much less than cubic and little progress has been made in last 20 years to improve the state of the art.

The main results of this work are Theorems 1 and 2. Theorem 1 refutes Conjecture 1 by Saks and Wigderson.

▶ **Theorem 1.** *There exists a Boolean function F for which $R_0(F) = \widetilde{O}(D(F)^{3/4})$.*

It is to be noted that this result does not match the lower bound of $R_0(f)$ in terms of $D(f)$. We also show quadratic separation between deterministic and one-sided randomized query complexity which is achieved by the same function.

▶ **Theorem 2.** *There exists a Boolean function F for which $R_1(F) = \widetilde{O}(\sqrt{D(F)})$.*

This separation matches the lower bound, upto logarithmic factors, on $R_1(f)$ in terms of $D(f)$ for any function f. These results give better separation between the corresponding complexity measures than what is known during this work.

The function F which yields these separation results was first introduced by Göös et al [4] for showing a gap between deterministic decision tree complexity and unambiguous non-deterministic decision tree complexity and resolving the famous *clique vs independent set* problem. We will define the function in Section 1.1.

Independently of us, Ambainis et al [2] proved various separation results between different query complexity measures. Among several other results, the authors prove the existence of a function f for which $R_0(f) = \widetilde{O}(\sqrt{D(f)})$. In view of the lower bound, this is the widest separation possible between these two measures. This also refutes the conjecture by Saks and Wigderson. Moreover, since $R_0(f) = \Omega(R_1(f))$, this also certifies the same separation as that of Theorem 2. However, the authors use a variant of the function F which was introduced by Göös et al [4] to show this separation. In our work, we showed that the original function F itself is sufficient to refute Saks-Wigderson conjecture and to show the widest possible separation between $D(f)$ and $R_1(f)$ for any boolean function f.

1.1 The Göös-Pitassi-Watson Function

We define the function F now. The domain of F is $\mathcal{D} = \{0,1\}^{n(1+\lceil \log n \rceil)}$. An input $M \in \mathcal{D}$ to F is viewed as a matrix of dimension $\sqrt{n} \times \sqrt{n}$. Each cell $M_{i,j}$ of M consists of two parts:

1. A *bit-entry* $b_{i,j} \in \{0,1\}$.
2. A *pointer-entry* $p_{i,j} \in \{0,1\}^{\lceil \log n \rceil}$. $p_{i,j}$ is either a valid pointer to some other cell of M, or is interpreted as $\perp$ (null). If $p_{i,j}$ is not a valid pointer to some other cell, we write "$p_{i,j} = \perp$".

Now, we define what we call a *valid pointer chain.* Assume that $t = \sqrt{n}$. For an input M to F, a sequence $((i_1, j_1), \ldots, (i_t, j_t))$ of indices in $[\sqrt{n}] \times [\sqrt{n}]$ is called a *valid* pointer chain if:

1. $b_{i_1,j_1} = 1$;
2. $b_{i_2,j_2} = \ldots = b_{i_t,j_t} = 0$;
3. $\forall k < i_1, p_{k,j_1} = \perp$;
4. for $\ell = 1, \ldots, t-1, p_{i_\ell,j_\ell} = (i_{\ell+1}, j_{\ell+1})$ and $p_{i_t,j_t} = \perp$;
5. $\{b_1, \ldots, b_t\} = \lfloor t \rfloor$;

F evaluates to 1 on M iff the following is true:

1. M contains a unique all 1's column j_1, i.e., there exists $j_1 \in [\sqrt{n}]$ such that $\forall i \in [\sqrt{n}]$, $b_{i,j_1} = 1$.

2. There exists a valid pointer chain $((i_1,j_1),\ldots,(i_t,j_t))$. This means that the column j_1 has a cell with non-null pointer entry. (i_1,j_1) is the cell on column j_1 with minimum row index whose pointer-entry is non-null. Starting from p_{i_1,j_1}, if we follow the successive pointers, the following conditions are satisfied: In each step except the last, the cell reached by following the pointer-entry of the cell in the previous step, contains a 0 as bit-entry and a non-null pointer as pointer-entry. In the last step, the cell contains a zero as bit-entry and a null pointer ($\perp$) as pointer-entry. Also, this pointer chain covers each column of M exactly once.

By a simple adversarial strategy, Göös et al. [4] showed that $D(F) = \widetilde{\Omega}(n)$. Our contribution is to show the following results.

▶ **Lemma 3.** *For the function F defined above, $R_0(F) = \widetilde{O}(n^{3/4})$.*

▶ **Lemma 4.** *For the function F defined above, $R_1(F) = \widetilde{O}(\sqrt{n})$.*

Clearly, Lemmas 3 and 4 imply Theorems 1 and 2 respectively.

2 Randomized One-sided Error Query Algorithm for F

We show that the randomized one-sided error query complexity of F is $\widetilde{O}(\sqrt{n})$. We first provide intuition for our one-sided error query algorithm for F before formally describing it.

Broad idea: Our algorithm errs on one side: on 0-inputs it always outputs 0 and on 1-inputs it outputs 1 with high probability.
The algorithm attempts to find a 1-certificate. If it fails to find a 1-certificate, it outputs 0. We show that on every 1-input, with high probability, the algorithm succeeds in finding a 1-certificate. The 1-certificate our algorithm looks for consists of:
1. A column j, all of whose bit-entries are 1's.
2. All null pointers of column j till its first non-null pointer-entry.
3. The pointer chain of length $\sqrt{n}$ that starts from the first non-null pointer entry, and in the next $\sqrt{n}-1$ hops, visits all the other columns. The bit entries of all the other cells of the pointer chain than the one in this column are 0.

To find a 1-certificate, the algorithm tries to find columns with 0-cells on them, and adds those columns to a set of discarded columns that it maintains. To this end, a first natural attempt is to repeatedly sample a cell randomly from M, and if its bit-entry is 0, try to follow the pointer originating from that cell. Following the chain, each time we visit a cell with bit-entry 0, we can discard the column on which the cell lies. We can expect that, with high probability, after sampling $O(\sqrt{n})$ cells, we land up on some cell in the middle portion of the correct pointer chain that is contained in the 1-certificate (we call this the *principal chain*). Then if we follow that pointer we spend $O(\sqrt{n})$ queries, and eliminate a constant fraction of the existing columns.

The problem with this approach is possible existence of other long pointer chains, than the principal chain. It may be the case that we land up on one such chain, of $\Omega(\sqrt{n})$ length, which passes entirely through the columns that we have already discarded. Thus we end up spending $\Omega(\sqrt{n})$ queries, but can discard only one column (the one we began from).

To bypass this problem, we start by observing that the principal chain passes through every column, and hence in particular through every undiscarded column. Let N be the number of undiscarded column at some stage of the algorithm. Note that the length of the principal chain is $\sqrt{n}$. Therefore if we start to follow it from a randomly chosen cell on it,

Algorithm 1

```
1: procedure MilestoneTrace($M, \mathcal{C}, i, j$)
2:     Read $b_{i,j}$;
3:     if $b_{i,j} = 1$ then return ;
4:     end if
5:     $step$:=0;
6:     $discard$:=1;
7:     $current$:=$(i, j)$;
8:     $seen$:=$\{j\}$;
9:     while $step \leq 100\sqrt{n} \cdot \frac{discard}{|\mathcal{C}|}$ do
10:        read the pointer-entry of $current$;
11:        $step \leftarrow step+1$;
12:        $current \leftarrow$ pointer-entry of $current$;
13:        if $current$ is $\perp$ then goto step21;
14:        end if
15:        read bit-entry of $current$;
16:        if $current$ is on a column $k$ in $\mathcal{C} \setminus seen$ and bit-entry of $current$ is 0 then
17:            $seen \leftarrow seen \cup \{k\}$;
18:            $discard \leftarrow discard+1$;
19:        end if
20:    end while
21:    $\mathcal{C} \leftarrow \mathcal{C} \setminus seen$;
22: end procedure
```

we are expected to see an undiscarded column in roughly another $\sqrt{n}/N$ hops. In view of this, we modify our algorithm as follows: while following a pointer chain, we check if on an average we are seeing one undiscarded column in every $O(\sqrt{n}/N)$ hops. If this check fails, we abandon following the pointer, sample another random cell from M, and continue. Our procedure MilestoneTrace does this pointer-traversal. We can prove that conditioned on the event that we land up on the principal chain, the above traversal algorithm enables us to eliminate a constant fraction of the existing undiscarded columns with high probability. We also show that spending $O(\sqrt{n}/N)$ queries for each column we eliminate is enough for us to get the desired query complexity bound.
After getting hold of the unique all 1's column, the final step is to check if all its bit-entries are indeed 1's, and if that can be completed into a full 1-certificate. That can clearly be done in $\widetilde{O}(\sqrt{n})$ queries. The VerifyColumn procedure does this.

2.1 The Algorithm

In this section we give the formal description and analysis of our one-sided error query algorithm for F: Algorithm 3. Algorithm 3 uses two procedures: VerifyColumn (see Algorithm 2) and MilestoneTrace (see Algorithm 1). As outlined in the previous section, VerifyColumn, given a column, checks if all its bit-entries are 1 and whether it can be completed into a 1-certificate. MilestoneTrace procedure implements the pointer traversal algorithm that we described in the preceding paragraph. We next describe the MilestoneTrace procedure in a little more detail. We recall from the last section that the algorithm discards columns in course of its execution. We denote the set of undiscarded columns by $\mathcal{C}$.

Algorithm 2

```
1:  procedure VerifyColumn($M, k$)
2:      Check if all the bit-entries of cells in the $k$-th column of $M$ are 1; If not, output 0;
3:
4:      if All the pointer-entries of cells in the the $k$-th column of $M$ are $\perp$ then
5:          Output 0;
6:      end if
7:      if The pointer chain starting from the first non-null pointer in column $k$ is valid then
8:          Output 1;
9:      else
10:         Output 0;
11:     end if
12: end procedure
```

MilestoneTrace procedure: The functions of the variables used are as follows:

1. *step*: Contains the number of pointer-entries queried so far. A bit query is always accompanied by a pointer query, unless the bit is 1 in which case the traversal stops. So upto logarithmic factor, the value in *step* gives us the number of bits queried.
2. *seen*: Set of columns that were undiscarded before the current run of MilestoneTrace, and that have so far been seen and marked for discarding.
3. *discard*: size of *seen*
4. *current*: Contains the indices of the cell currently being considered.

The condition in the *while* loop checks if the number of queries spent is not too much larger than $\frac{\sqrt{n}}{|\mathcal{C}|}$ at any point in time. The *if* condition in line 13 checks if the current pointer-entry is null. If it is null, $\mathcal{C}$ is updated, and control returns to Algorithm 3. The condition in line 13 checks if the pointer chain has reached its end.

To analyse Algorithm 3, we need to prove two statements about MilestoneTrace, which we now informally state. Assume that the algorithm is run on a 1-input.

1. Conditioned on the event that a cell (i, j) randomly chosen from the columns in $\mathcal{C}$ is on the principal chain, a call to MilestoneTrace$(M, \mathcal{C}, i, j)$ serves to eliminate a constant fraction of surviving columns with high probability.
2. For upper bounding the number of queries, it is enough to ensure that the average number of queries spent for each eliminated column is not too much larger than $\frac{\sqrt{n}}{|\mathcal{C}|}$. Note that $|\mathcal{C}|$ is the number of undiscarded columns during the start of the MilestoneTrace procedure.

In Section 2.2, we prove that Algorithm 3 makes $\widetilde{O}(\sqrt{n})$ queries on every input. In Section 2.3 we prove that Algorithm 3 succeeds with probability 1 on 0-inputs and with probability at least 2/3 on 1-inputs.

2.2 Query complexity of Algorithm 3

In this subsection we analyse the query complexity of Algorithm 3. We bound the total number of $b_{i,j}$'s and $p_{i,j}$'s read by the algorithm. Upto logarithmic factors, that is the total number of bits queried. For the rest of this subsection, one query will mean one query to a bit-entry or a pointer-entry of some cell.

Algorithm 3

```
1: $\mathcal{C}$ := set of columns in $M$.
2: for $t = 1$ to $O(\sqrt{n}\log n)$ do
3:     if $|\mathcal{C}| < 100$ then
4:         goto step 10;
5:     end if
6:     Sample a column $j$ from $\mathcal{C}$ uniformly at random;
7:     Sample $i \in [\sqrt{n}]$ uniformly at random;
8:     MilestoneTrace$(M, \mathcal{C}, i, j)$;
9: end for
10: if $|\mathcal{C}| > 100$ or $|\mathcal{C}| = 0$ then
11:     Output 0;
12: else
13:     Read all columns in $\mathcal{C}$;
14:     if There is a column $k$ with all bit-entries equal to 1 then
15:         VerifyColumn$(M, k)$;
16:     else
17:         Output 0;
18:     end if
19: end if
```

We first analyse the MilestoneTrace procedure. Recall that $\mathcal{C}$ denotes the set of undiscarded columns.

▶ **Lemma 5.** *Let i, j be such that $b_{i,j} = 0$. Let Q and D respectively be the number of queries made and number of columns discarded by a call to* MilestoneTrace$(M, \mathcal{C}, i, j)$. *Then,*

$$Q \le D \cdot \frac{200\sqrt{n}}{|\mathcal{C}|} + 3$$

Proof. We note that the variable *step* contains the number of pointer queries made so far, and the variable *discard* maintains the number of columns marked so far for discarding. Every time the *while* loop is entered, $step \le 100\sqrt{n} \cdot \frac{discard}{|\mathcal{C}|}$. In each iteration of the *while* loop, step goes up by 1. So at any point, $step \le 100\sqrt{n} \cdot \frac{discard}{|\mathcal{C}|} + 1$. The lemma follows by observing that the total number of bit-entries queried is at most one more than total number of pointer-entries queried. ◀

We now use Lemma 5 to bound the total number of queries made by Algorithm 3.

▶ **Lemma 6.** *Algorithm 3 makes $\widetilde{O}(\sqrt{n})$ queries on each input.*

Proof. Whenever $b_{i,j} = 1$, MilestoneTrace$(M, \mathcal{C}, i, j)$ returns after reading $b_{i,j}$. So the total number of queries made by all calls to MilestoneTrace$(M, \mathcal{C}, i, j)$ on such inputs is $\widetilde{O}(\sqrt{n})$.
After leaving the *while* loop, the total number of queries required to read constantly many columns in $\mathcal{C}$ and to run VerifyColumn is $\widetilde{O}(\sqrt{n})$.
Since inside the *while* loop all the queries are made inside the MilestoneTrace procedure, it is enough to show that the total number of queries made by all calls to MilestoneTrace$(M, \mathcal{C}, i, j)$ on inputs for which $b_{i,j} = 0$ is $\widetilde{O}(\sqrt{n})$.
Let $t = \widetilde{O}(\sqrt{n})$ be the total number of calls to MilestoneTrace on such inputs, made

in the entire run of Algorithm 3. Let s_i be the value of $|\mathcal{C}|$ when the i-th call to MILESTONETRACE is made, and let s_{t+1} be the value of $|\mathcal{C}|$ after the execution of the t-th call to MILESTONETRACE completes . Let Δs_i and Δq_i respectively be the number of columns discarded and number of queries made in the i-th call to MILESTONETRACE. Since $\mathcal{C}$ shrinks only when $b_{i,j} = 0$, we have $\Delta s_i = s_i - s_{i+1}$ for $i = 1 \ldots t$. Since $s_1 = \sqrt{n}$, we have that for $i = 2, \ldots, t$, $s_i = \sqrt{n} - \sum_{j=1}^{i-1} \Delta s_j$.

From Lemma 5 we have $\Delta q_i \le \Delta s_i \cdot \frac{200\sqrt{n}}{s_i} + 3$ for $i = 1, \ldots, t$. Substituting $\sqrt{n} - \sum_{j=1}^{i-1} \Delta s_j$ for s_i when $i > 1$, and adding, we have,

$$\begin{aligned}
\sum_{i=1}^{t} \Delta q_i &\le 200\sqrt{n} \cdot \sum_{i=1}^{t} \frac{\Delta s_i}{s_i} + 3t \\
&= 200\sqrt{n} \cdot \left(\frac{\Delta s_1}{\sqrt{n}} + \sum_{i=2}^{t} \frac{\Delta s_i}{\sqrt{n} - \sum_{j=1}^{i-1} \Delta s_j} \right) + \widetilde{O}(\sqrt{n}) \\
&\le 200\sqrt{n} \cdot \left(\left(\frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n}-1} + \ldots + \frac{1}{\sqrt{n} - \Delta s_1 + 1} \right) + \right. \\
&\qquad \left(\frac{1}{\sqrt{n} - \Delta s_1} + \frac{1}{\sqrt{n} - \Delta s_1 - 1} + \ldots + \frac{1}{\sqrt{n} - \Delta s_1 - \Delta s_2 + 1} \right) + \\
&\qquad \ldots + \left(\frac{1}{\sqrt{n} - \sum_{j=1}^{t-1} \Delta s_j} + \frac{1}{\sqrt{n} - \sum_{j=1}^{t-1} \Delta s_j - 1} + \right. \\
&\qquad \left. \left. \ldots + \frac{1}{\sqrt{n} - \sum_{j=1}^{t-1} \Delta s_j - \Delta s_t + 1} \right) \right) \\
&\qquad + \widetilde{O}(\sqrt{n}) \\
&\le O(\sqrt{n}) \cdot \left(\sum_{i=1}^{\sqrt{n}} \frac{1}{i} \right) + \widetilde{O}(\sqrt{n}) \\
&= O(\sqrt{n} \log n) + \widetilde{O}(\sqrt{n}) \\
&= \widetilde{O}(\sqrt{n}).
\end{aligned}$$

Hence proved. ◀

2.3 Success Probability of Algorithm 3

In this section we prove that Algorithm 3 outputs correct answer with probability 1 on 0-inputs and with probability at least 2/3 on 1-inputs. We start by a proving a probability statement (Lemma 7) that will help us in the analysis.

Let $x_1, \ldots, x_\ell$ be non-negative real numbers and $\sum_{i=1}^{\ell} x_i = N$. We say that an index $I \in [\ell]$ is *bad* if there exists a non-negative integer $0 \le D \le N - I$ such that

$$\sum_{i=I}^{I+D} x_i > 100(D+1) \cdot \frac{N}{\ell}$$

We say that an index I is *good* if I is not bad.

▶ **Lemma 7.** *Let I be chosen uniformly at random from $[\ell]$. Then,*

$$\mathbb{P}[I \text{ is good}] > \frac{99}{100}$$

Proof. We show existence of a set $K = \{J_1, \cdots, J_t\}$ of disjoint sub-intervals of $[1, \ell]$ with integer end-points, having the following properties:

1. Every bad index is in some interval $J_i \in K$.
2. $\forall 1 \le i \le t, \sum_{j \in J_i} x_j > 100|J_i| \cdot \frac{N}{\ell}$.

It then follows that the number of bad indices is upper bounded by $\sum_{i\in[t]} |J_i|$ (by property 1). But $N \ge \sum_{i\in[t]} \sum_{j\in J_i} x_j > 100 \cdot \frac{N}{\ell} \sum_{i\in t} |J_i|$, which gives us that $\sum_{i\in t} |J_i| < \frac{\ell}{100}$. In the above chain of inequalities, the first inequality follows from the disjointness of J_i's and the second inequality follows from property 2.

Now we describe a greedy procedure to obtain such a set K of intervals. Let J be the smallest bad index. Then there exists a D such that $\sum_{i\in[J,J+D]} x_i > 100(D+1) \cdot \frac{N}{\ell}$. We include the interval $[J, J+D]$ in K. Then let J' be the smallest bad index greater than $J+D$. Then there exists a D' for which $\sum_{i\in[J',J'+D']} x_i > 100(D'+1) \cdot \frac{N}{\ell}$. We include $[J', J'+D']$ in K. We continue in this way till there is no bad index which is not already contained in some interval in K. It is easy to verify that the intervals in the set K thus formed are disjoint, and the set K satisfies properties 1 and 2. ◀

Let us begin by showing that algorithm 3 is correct with probability 1 on 0-inputs of F.

▶ **Lemma 8.** *If Procedure* VerifyColumn *outputs* 1 *on inputs M and k, then M is a* 1*-input of F.*

Proof. VerifyColumn outputs 1 only if the column k has all its bit-entries equal to 1, and if the pointer chain starting from the first non-null pointer entry is valid (recall the definition of a *valid pointer chain* from Section 1.1). From the definition of F, for such inputs F evaluates to 1. ◀

▶ **Corollary 9.** *Let M be a* 0*-input of of F. Then algorithm 3 outputs* 0 *with probability* 1.

Proof. The corollary follows by observing that if algorithm 3 returns 1, a call to Verify-Column also returns 1, and hence from Lemma 8 the input is a 1-input of F. ◀

Let us now turn to 1-inputs of F. Let M be a 1-input of F, that we fix for the rest of this subsection. Without explicit mention, for the rest of the subsection we assume that Algorithm 3 is run on M. Since M is a 1-input, by the definition of F, there is a column C such that all its bit-entries are 1, and the pointer chain starting from the first non-null pointer-entry of C is valid. Call this pointer chain the *principal chain.* Let $(C = c_1, \ldots, c_{\sqrt{n}})$ be the order of columns of M in which the pointer chain crosses them. Let $(C = m_1, \ldots, m_{|\mathcal{C}|})$ be the order of the columns of $\mathcal{C}$ in which the pointer chain crosses them. Note that the column C always belongs to $\mathcal{C}$, as a column is discarded only if the bit-entry of some cell on it is 0. Define X_i to be the number of c_j's between m_i and m_{i+1}, including m_i, if $i < |\mathcal{C}|$, and the number of c_j's after m_i, including m_i, if $i = |\mathcal{C}|$. Clearly $\sum_{i=1}^{|\mathcal{C}|} X_i = \sqrt{n}$.

▶ **Lemma 10.** *Let (i, j) be a randomly chosen cell on the restriction of the principal chain to the columns in $\mathcal{C}$. Let $j = m_\ell \in \{m_1, \ldots, m_{|\mathcal{C}|}\}$. Let $|\mathcal{C}| = N \ge 100$. Then with probability at least $\frac{97}{100}$ over the choice of (i, j), a run of the procedure* MilestoneTrace *on inputs $M, \mathcal{C}, i, j$ shrinks the size of $\mathcal{C}$ to at most $\frac{99N}{100}$.*

Proof. By applying Lemma 7 on the sequence $(X_i)_{i=1}^{|\mathcal{C}|}$ described in the paragraph preceding this lemma, except with probability at least $1/100 + 1/100 + 1/|\mathcal{C}| \leq 3/100$, ℓ is a good index, $\ell < \frac{99N}{100}$ (i.e. the column j has at least $\frac{N}{100}$ columns of $\mathcal{C}$ ahead of it on the principal chain), and $j \neq C$. Since $j \neq C$, the bit-entry of the cell sampled is 0, and hence procedure MilestoneTrace does not return control in step 3. In the procedure MilestoneTrace, if *current* is on the principal chain, the condition in line 13 cannot be satisfied unless *current* is the last cell on the chain. Now, if the condition in the while loop is violated while *current* is on the principal chain, it implies that j is a bad index. Thus with probability at least $1 - 3/100 = 97/100$, the procedure does not terminate as long as all the $\frac{N}{100}$ columns ahead of j are not seen. Since all columns in $\mathcal{C}$ that are seen are discarded, we have the lemma. ◀

Now, let us bound the number of iterations of the *for* loop of algorithm 3 required to shrink $|\mathcal{C}|$ by a factor of $1/100$.

▶ **Lemma 11.** *Assume that at a stage of execution of algorithm 3 where the control is in the beginning of the* for *loop,* $|\mathcal{C}| = N$*. Then except with probability* $1/25$*, after* $10\sqrt{n}$ *iterations of the* for *loop,* $|\mathcal{C}|$ *will become at most* $99N/100$*.*

Proof. The probability that a cell on the principal chain is sampled in steps 6 and 7 is $\frac{1}{\sqrt{n}}$. So the probability that in none of the $10\sqrt{n}$ executions of steps 6 and 7, a cell on the principal chain is picked is $(1 - \frac{1}{\sqrt{n}})^{10\sqrt{n}} \leq \frac{1}{100}$. Conditioned on the event that a cell on the principal chain is sampled, from lemma 10, except with probability $3/100$, $|\mathcal{C}|$ reduces by a factor of $1/100$ in the following run of MilestoneTrace. Union bounding we have that except with probability $1/100 + 3/100 = 1/25$, after $10\sqrt{n}$ iterations of the *for* loop, $|\mathcal{C}| \leq 99N/100$. ◀

Let t be the minimum integer such that $\sqrt{n} \cdot \left(\frac{99}{100}\right)^t < 100$. Thus $t = O(\log n)$. For $i = 1, \ldots, t$, let the random variable Y_i be equal to the index of the first iteration of the *for* loop of Algorithm 3 after which $|\mathcal{C}| \leq \sqrt{n}. \left(\frac{99}{100}\right)^i$. Let $Z_1 = Y_1$ and for $i = 2, \ldots, t$ define $Z_i = Y_i - Y_{i-1}$. From Lemma 11, for each i we have $\mathbb{E}[Z_i] \leq 25 \times 10\sqrt{n} = O(\sqrt{n})$. By linearity of expectation, we have $\mathbb{E}[\sum_{i=1}^{t} Z_i] = O(\sqrt{n} \log n)$. By Markov's inequality, with probability at least $2/3$, $\sum_{i=1}^{t} Z_i = O(\sqrt{n} \log n)$. Thus, if we choose the constant hidden in the number of iterations of the *for* loop of Algorithm 3 large enough, then with probability at least $2/3$, $|\mathcal{C}|$ shrinks to less than 100. Then the VerifyColumn procedure reads all the columns in $\mathcal{C}$ and outputs the correct value of F. Thus we have proved the following Lemma.

▶ **Lemma 12.** *With probability at least* $2/3$*, algorithm 3 outputs* 1 *on a* 1*-input.*

Lemma 4 follows from Lemma 6, Corollary 9 and Lemma 12.

3 Randomized Zero-error Query Algorithm for F

We first present a randomized query algorithm which satisfies the following conditions: If the algorithm outputs 0 then the given input is a 0-input (the algorithm actually exhibits a 0-certificate) and if the given input is a 0-input, then the algorithm outputs 0 with high probability. This algorithm makes $\tilde{O}(n^{3/4})$ queries in worst case. For the randomized zero-error algorithm we run Algorithm 3 and this algorithm one after another. If Algorithm 3 outputs 1 then we stop and output 1. Else, if Algorithm 4 says 0, we stop and output

0. Otherwise, we repeat. By the standard argument of $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$ we get the randomized zero-error algorithm. Though the query complexity of Algorithm 3 is $\tilde{O}(\sqrt{n})$, we get the zero-error query complexity of F to be $\tilde{O}(n^{3/4})$ because of the query complexity of Algorithm 4.

Now we define *column covering* and *column span* which we will use next.

▶ **Definition 13.** For two columns C_i and C_j ($C_i(C_j)$ denotes the i-th (j-th) column) in input matrix M, we say C_j is covered by C_i if there is a cell (k,i) in C_i and a sequence $(\beta_1,\delta_1),\ldots,(\beta_t,\delta_t)$ of pairs from $[\sqrt{n}] \times [\sqrt{n}]$ such that:

1. $b_{k,i} = 0$,
2. $\delta_t = j$,
3. for all $\ell \in [t]$, $b_{\beta_\ell,\delta_\ell} = 0$ and
4. $p_{k,i} = (\beta_1,\delta_1)$ and for $\ell = 1,\ldots,t-1, p_{(\beta_\ell,\delta_\ell)} = (\beta_{\ell+1},\delta_{\ell+1})$.
5. For $1 \le k < \ell \le t, \delta_k \ne \delta_\ell$ and for $1 \le k \le t, i \ne \delta_k$.

▶ **Definition 14.** For a column C, we define Span_C to be the subset of columns in M which consists of C and any column which is covered by C.

We first give an informal description of the algorithm and then we proceed to formally analyze the algorithm in Section 3.1. As mentioned before this is also a one-sided algorithm, i.e., it errs on one side but it errs on the different side than that of Algorithm 3. The 0-certificates it attempts to capture are as follows:

1. If each of the columns has a cell with bit-entry 0, then the function evaluates to 0. Those bit-entries form a 0-certificate. If there are many 0's in each column, The algorithms may capture such a certificate in the first phase (*sparsification*).
2. Two columns C_1 and C_2 in M such that $C_1 \notin \mathsf{Span}_{C_2}$ and $C_2 \notin \mathsf{Span}_{C_1}$. Existence of two such columns makes the existence of a valid pointer chain impossible. This is captured in the second phase of the algorithm.
3. Lastly, if there is a column all of whose bit-entries are 1, which does not have a valid pointer chain, then that is also a 0-certificate. The algorithm may capture such a certificate in the last phase.

The algorithm proceeds as follows: The main goal of the algorithm is to eliminate any column where it finds a 0 in any of its cells. First the algorithm filters out the columns with large number of 0's with high probability by random sampling. The algorithm probes $\widetilde{O}(n^{1/4})$ locations at random in each column and if it finds any 0 in any column, it eliminates that column. This step is called *sparsification.* After sparsification, we are guaranteed that all the columns have small number of 0's. Now the remaining columns can have either of the following two characteristics: First, a large number of the columns in existing column set have large span. This implies that if we choose a column randomly from the existing columns, the column will span a large number of columns (i.e., a constant fraction of existing columns) with high probability and we can eliminate all of them. The algorithm does this exactly in the **procedure A** of the second phase. The other case can be where most of the columns have small spans. We can show that if this is the case, then if we pick two random columns C_i and C_j from the set of existing columns, C_i will not lie in the span of C_j and vice-versa with high probability, certifying that F is 0. This case is taken care of in the **procedure B** of the second phase of the algorithm.

The algorithm runs **procedure A** and **procedure B** one after another for logarithmic number of steps. If at any point of the iteration, the algorithm finds two columns which are not in span of each other, the algorithm outputs 0 and terminates. Otherwise, as the **procedure A** decreases the number of existing columns by a constant factor in each iteration, with logarithmic number of iteration, either we completely exhaust the column set, which is again a 0-certificate, or we are left with a single column. Then the algorithm checks the remaining column and the validity of the pointer chain if that column is an all 1's column and answers accordingly. This captures the third kind of 0-certificate as mentioned before.

In Algorithm 4, we set τ to be the least number such that $\sqrt{n} \cdot (\frac{99}{100})^{\tau} \leq 1$. Clearly $\tau = O(\log n)$. We also set α to an appropriate constant.

3.1 Analysis of Algorithm 4

Let's first look at the query complexity of the algorithm.

▶ **Lemma 15.** *The query complexity of Algorithm 4 is $\widetilde{O}(n^{3/4})$ in worst case.*

Proof. We count the number of bit-entries and pointer-entries of the input matrix the algorithm probes. Up to logarithmic factor, that is asymptotically same as the number of bits queried.
The first *for* loop runs for $\sqrt{n}$ iteration and in each iteration samples T cells from a column. So the number of probes of the first *for* loop is $O(\sqrt{n} \times T) = \widetilde{O}(n^{3/4})$.

In **procedure A**, the number of probes needed to scan the column and to trace pointer from the column is $\widetilde{O}(n^{3/4})$. In **procedure B**, the algorithm has to check the span of two columns, which takes $\widetilde{O}(n^{3/4})$ probes. The number of iterations of the *for* loop of line 9 is at most $\tau = O(\log n)$. Hence the total number of probes made inside the *for* loop is $\widetilde{O}(n^{3/4})$.

Lastly, VerifyColumn takes $O(\sqrt{n})$ probes. So the total number of probes is bounded by $\widetilde{O}(n^{3/4})$. Thus the claim follows. ◀

The first *for* loop, i.e., line 3 to 8 is called *sparsification.* We have the following guarantee after sparsification.

▶ **Lemma 16.** *After the sparsification, with probability at least $99/100$, every column in $\mathcal{C}$ has at most $n^{1/4}$ cells with bit-entry 0.*

Proof. We will bound the probability that all the T probes in a column outputs 1 conditioned on the fact that the column has more than $n^{1/4}$ 0's. A single probe in such a column outputs 0 with probability at least $1/n^{1/4}$. Hence all the probes output 1 with probability $(1 - 1/n^{1/4})^T \leq \frac{1}{100\sqrt{n}}$. By union bound, this happens to some column in M with probability at most $1/100$. ◀

This implies that except with probability $1/100$, the *if* conditions of lines 20 and 32 are never satisfied.

▶ **Lemma 17.** *Either of the following is true in each iteration of the* for *loop of line 9:*
1. *for a random column $C \in \mathcal{C}$, $|\mathsf{Span}_C| > |\mathcal{C}|/100$ with probability at least $1/100$.*
2. *For two randomly picked columns C_i and C_j in $\mathcal{C}$, with probability at least $24/25$, $C_j \notin \mathsf{Span}_{C_i}$ and $C_i \notin \mathsf{Span}_{C_j}$.*

Proof. Suppose (1) does not hold. For two random columns C_i and C_j, Let L_i (L_j) be the event that $|\mathsf{Span}_{C_i}|$ ($|\mathsf{Span}_{C_j}|$)$| > |\mathcal{C}|/100$. Let $E_{i,j}$ ($E_{j,i}$) be the event that $C_j \in \mathsf{Span}_{C_i}$ ($C_i \in \mathsf{Span}_{C_j}$). Thus we have,

$$\mathbb{P}\{E_{i,j}\} = \mathbb{P}\{L_i\} \cdot \mathbb{P}\{E_{i,j}|L_i\} + \mathbb{P}\{\overline{L_i}\} \cdot \mathbb{P}\{E_{i,j}|\overline{L_i}\}$$

Algorithm 4

```
1: $\mathcal{C}$ := Set of columns in $M$;
2: $\tau$ := Least number such that $\sqrt{n} \cdot (\frac{99}{100})^{\tau} \leq 1$;
3: for each column $C$ in $\mathcal{C}$ do
4:     Sample $T = 10 \cdot n^{1/4} \log n$ cells uniformly at random;
5:     if any bit-entry of any cell is 0 then
6:         $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\}$;
7:     end if
8: end for
9: for $t = 1$ to $\tau$ do
10:     if $|\mathcal{C}| \leq 1$ then
11:         goto step 40
12:     end if
13:     repeat
14:         procedure A
15:             Sample a column $C$ from $\mathcal{C}$ uniformly at random;
16:             Read all entries of all cells of $C$;
17:             if All bit-entries are 1 then
18:                 VerifyColumn($M, C$);
19:             end if
20:             if Number of 0 bit-entries in $C > n^{1/4}$ then
21:                 Output 1 and abort;
22:             end if
23:             For each cell on $C$ with bit-entry 0, trace pointer and compute $\mathsf{Span}_C$;
24:             $\mathcal{C} \leftarrow \mathcal{C} \setminus \mathsf{Span}_C$;
25:         end procedure
26:     until $\alpha \log \log n$ times
27:     procedure B
28:         Pick two columns $C_1$ and $C_2$ uniformly at random from $\mathcal{C}$;
29:         if All bit-entries of $C_1$ ($C_2$) are 1 then
30:             VerifyColumn($M, C_1$) (VerifyColumn($M, C_2$));
31:         end if
32:         if Number of 0 bit-entries in $C_1$ or $C_2 > n^{1/4}$ then
33:             Output 1 and abort;
34:         end if
35:         if $C_2 \notin \mathsf{Span}_{C_1}$ and $C_1 \notin \mathsf{Span}_{C_2}$ then
36:             Output 0 and abort;
37:         end if
38:     end procedure
39: end for
40: if $\mathcal{C} = \emptyset$ then
41:     Output 0;
42: end if
43: if $|\mathcal{C}| = 1$ then
44:     Let $\mathcal{C} = \{C\}$.
45:     VerifyColumn($M, C$);
46: end if
47: Output 1.
```

$$\leq \mathbb{P}\{L_i\} + \mathbb{P}\{E_{i,j}|\overline{L_i}\}$$
$$\leq \frac{1}{100} + \frac{1}{100} = \frac{1}{50}$$

Similarly $\mathbb{P}\{E_{j,i}\} \leq \frac{1}{50}$. By union bound, (2) is true; ◀

Now we are ready prove the correctness of the algorithm.

▶ **Lemma 18.** *Given a* 0*-input, Algorithm 4 outputs* 0 *with probability at least* 19/20 .

Proof. We first note that after the execution of *for* loop in line 3, except with probability at most 1/100 there is no column in $\mathcal{C}$ having more than $n^{1/4}$ cells with bit-entries 0.

If the algorithm finds a column all of whose bit-entries are 1, it gives correct output by a run of VerifyColumn.

Next, we note that if in any iteration of the *for* loop (line 9), condition (2) of Claim 17 is satisfied, then we find a 0-certificate (i.e. a pair of columns, none of which lies in the span of the other) with probability at least 24/25.

Finally, assume that for each iteration of the *for* loop, condition (2) is not satisfied. This implies that for each iteration of the loop, condition (1) is satisfied (From Claim 17). As we run **procedure A** $\alpha \log\log n$ times, with probability at least $1 - (\frac{99}{100})^{\alpha \log\log n} \geq 1 - \frac{1}{100\tau}$ (for appropriate setting of the constant α) we land up on a column whose span is at least $|\mathcal{C}|/100$ and hence we eliminate 1/100 fraction of columns in $\mathcal{C}$, in one of the iterations of the inner *repeat* loop (line 13). By union bound, the probability that there is even one bad *repeat* loop where we do not eliminate $|\mathcal{C}|/100$ columns, is at most 1/100. Thus the probability that after the execution of *for* loop is over, $|\mathcal{C}| > 1$, is at most 1/100. So, the total error probability is bounded by $1/100 + \max\{1/25, 1/100\} = 1/20$ from which the claim follows. ◀

▶ **Lemma 19.** *Given a* 1*-input, Algorithm 4 outputs 1 with probability 1.*

Proof. The proof of this claim is straight-forward. As mentioned before, Algorithm 4 outputs 0 only if it finds a 0-certificate. As there is no 0-certificate for a 1-input , the algorithm outputs 1. ◀

Lemma 3 follows by combining Lemma 4, Lemma 18, Lemma 16 and Lemma 19.

▶ Remark. It is observed [1] that if we consider a slight variant of the function F, where the input matrix is a $n^{2/3} \times n^{1/3}$ matrix instead of $\sqrt{n} \times \sqrt{n}$ and modify Algorithm 4 accordingly, we get a $\widetilde{O}(n^{2/3})$ algorithm. It is to be noted that the query complexity of Algorithm 3 (modified accordingly) worsens to $\widetilde{O}(n^{2/3})$ for this function. This shows that for a minor variant of the function F, our algorithm can show a better separation between deterministic and zero-error randomized query complexity. However, the modified function cannot show the widest separation between deterministic and bounded error randomized query complexity.

Acknowledgments. We thank Arkadev Chattopadhyay, Prahladh Harsha and Srikanth Srinivasan for useful discussions.

References

1 Scott Aaronson. A query complexity breakthrough. shtetl-optimized.
2 Andris Ambainis, Kaspars Balodis, Aleksandrs Belovs, Troy Lee, Miklos Santha, and Juris Smotrovs. Separations in query complexity based on pointer functions. *CoRR*, abs/1506.04719, 2015.

3 Manuel Blum and Russell Impagliazzo. Generic oracles and oracle classes (extended abstract). In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 118–126, 1987.

4 Mika Göös, Toniann Pitassi, and Thomas Watson. Deterministic communication vs. partition number. *Electronic Colloquium on Computational Complexity (ECCC)*, 22:50, 2015.

5 Juris Hartmanis and Lane A. Hemachandra. One-way functions, robustness, and the non-isomorphism of np-complete sets. In *Proceedings of the Second Annual Conference on Structure in Complexity Theory, Cornell University, Ithaca, New York, USA, June 16-19, 1987*, 1987.

6 Noam Nisan. CREW prams and decision trees. *SIAM J. Comput.*, 20(6):999–1007, 1991.

7 Ronald L. Rivest and Jean Vuillemin. On recognizing graph properties from adjacency matrices. *Theor. Comput. Sci.*, 3(3):371–384, 1976.

8 Michael E. Saks and Avi Wigderson. Probabilistic boolean decision trees and the complexity of evaluating game trees. In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 29–38, 1986.

9 Miklos Santha. On the monte carlo boolean decision tree complexity of read-once formulae. In *Proceedings of the Sixth Annual Structure in Complexity Theory Conference, Chicago, Illinois, USA, June 30 - July 3, 1991*, pages 180–187, 1991.

10 Marc Snir. Lower bounds on probabilistic linear decision trees. *Theor. Comput. Sci.*, 38:69–82, 1985.

11 Gábor Tardos. Query complexity, or why is it difficult to seperate NP $^{\mathrm{a}}$ cap co NP$^{\mathrm{a}}$ from P$^{\mathrm{a}}$ by random oracles a? *Combinatorica*, 9(4):385–392, 1989.