Separations Between Combinatorial Measures for Transitive Functions

Sourav Chakraborty
Indian Statistical Institute, Kolkata, India

Chandrima Kayal
Indian Statistical Institute, Kolkata, India

Manaswi Paraashar
Aarhus University, Denmark

Abstract

The role of symmetry in Boolean functions $f : \{0,1\}^n \to \{0,1\}$ has been extensively studied in complexity theory. For example, symmetric functions, that is, functions that are invariant under the action of S_n, is an important class of functions in the study of Boolean functions. A function $f : \{0,1\}^n \to \{0,1\}$ is called transitive (or weakly-symmetric) if there exists a transitive group G of S_n such that f is invariant under the action of G. In other words, the value of the function remains unchanged even after the input bits of f are moved around according to some permutation $\sigma \in \mathsf{G}$. Understanding various complexity measures of transitive functions has been a rich area of research for the past few decades.

This work studies transitive functions in light of several combinatorial measures. The question that we try to address in this paper is what are the maximum separations between various pairs of combinatorial measures for transitive functions. Such study for general Boolean functions has been going on for many years. Aaronson et al. (STOC, 2021) have nicely compiled the current best-known results for general Boolean functions. But before this paper, no such systematic study had been done on the case of transitive functions.

Separations between a pair of combinatorial measures are shown by constructing interesting functions that demonstrate the separation. Over the past three decades, various interesting classes of functions have been designed for this purpose. In this context, one of the celebrated classes of functions is the "pointer functions". Ambainis et al. (JACM, 2017) constructed several functions, which are modifications of the pointer function in Göös et al. (SICOMP, 2018 / FOCS, 2015), to demonstrate the separation between various pairs of measures. In the last few years, pointer functions have been used to show separation between various other pairs of measures (Eg: Mukhopadhyay et al. (FSTTCS, 2015), Ben-David et al. (ITCS, 2017), Göös et al. (ToCT, 2018 / ICALP, 2017)).

However, the pointer functions themselves are not transitive. Based on the various kinds of pointer functions, we construct new transitive functions, which we use to demonstrate similar separations between various pairs of combinatorial measures as demonstrated by the original pointer functions. Our construction of transitive functions depends crucially on the construction of particular classes of transitive groups whose actions, though involved, help to preserve certain structural features of the input strings. The transitive groups we construct may be of independent interest in other areas of mathematics and theoretical computer science.

We summarize the current knowledge of relations between various combinatorial measures of transitive functions in a table similar to the table compiled by Aaronson et al. (STOC, 2021) for general functions.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Transitive functions, Combinatorial complexity of Boolean functions

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.36

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2103.12355

49th International Colloquium on Automata, Languages, and Programming (ICALP 2022).
Editors: Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff;
Article No. 36; pp. 36:1–36:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

For a Boolean function $f : \{0,1\}^n \to \{0,1\}$ what is the relationship between its various combinatorial measures, like deterministic query complexity ($\mathsf{D}(f)$), bounded-error randomized and quantum query complexity ($\mathsf{R}(f)$ and $\mathsf{Q}(f)$ respectively), zero -randomized query complexity ($\mathsf{R}_0(f)$), exact quantum query complexity ($\mathsf{Q}_\mathsf{E}(f)$), sensitivity ($\mathsf{s}(f)$), block sensitivity ($\mathsf{bs}(f)$), certificate complexity ($\mathsf{C}(f)$), randomized certificate complexity ($\widetilde{\mathsf{RC}}(f)$), unambiguous certificate complexity ($\mathsf{UC}(f)$), degree ($\deg(f)$), approximate degree ($\widetilde{\deg}(f)$) and spectral sensitivity ($\lambda(f)$)[1]? For over three decades, understanding the relationships between these measures has been an active area of research in computational complexity theory. These combinatorial measures have applications in many other areas of theoretical computer science, and thus the above question takes a central position.

In the last couple of years, some of the more celebrated conjectures have been answered - like the quadratic relation between sensitivity and degree of Boolean functions [21]. We refer the reader to the survey [12] for an introduction to this area.

Understanding the relationship between various combinatorial measures involves two parts:

- Relationships – proving that one measure is upper bounded by a function of another measure. For example, for any Boolean function f, $\deg(f) \leq \mathsf{s}(f)^2$ and $\mathsf{D}(f) \leq \mathsf{R}(f)^2$.
- Separations – constructing functions that demonstrates separation between two measures. For example, there exists a Boolean function f with $\deg(f) \geq \mathsf{s}(f)^2$. Also there exists another Boolean function g with $\mathsf{D}(g) \geq \mathsf{R}(g)^2$.

Obtaining tight bounds between pairs of combinatorial measures - that is, when the relationship and the separation results match - is the holy grail of this area of research. The current best-known results for different pairs of functions have been nicely compiled in [2].

For special classes of Boolean functions the relationships and the separations might be different than that of general Boolean functions. For example, while it is known that there exists f such that $\mathsf{bs}(f) = \Theta(\mathsf{s}(f)^2)$ [26], for a symmetric function a more tighter result is known, $\mathsf{bs}(f) = \Theta(\mathsf{s}(f))$. The best-known relationship of $\mathsf{bs}(f)$ for a general Boolean functions is $\mathsf{s}(f)^4$ [21]. How the various measures behave for different classes of functions has been studied since the dawn of this area of research.

Transitive Functions. One of the most well-studied classes of Boolean functions is that of the transitive functions. A function $f : \{0,1\}^n \to \{0,1\}$ is transitive if there is a transitive group $G \leq \mathsf{S}_n$ such that the function value remains unchanged even after the indices of the input is acted upon by a permutation from G. Note that, when $G = \mathsf{S}_n$ then the function is symmetric. Transitive functions (also called "weakly symmetric" functions) has been studied extensively in the context of various complexity measure. This is because symmetry is a natural measure of the complexity of a Boolean function. It is expected that functions with more symmetry must have less variation among the various combinatorial measures. A recent work [7] has studied the functions under various types of symmetry in terms of quantum speedup. So, studying functions in terms of symmetry is important in various aspects.

For example, for symmetric functions, where the transitive group is S_n, most of the combinatorial measures become the same up to a constant [2]. Another example of transitive functions is the graph properties. The input is the adjacency matrix, and the transitive group

[1] For formal definitions of the various measures used in this paper please refer to the full version of this paper [15].

[2] There are still open problems on the tightness of the constants.

is the graph isomorphism group acting on the bits of the adjacency matrix. [31, 29, 23, 17] tried to obtain tight bounds on the relationship between sensitivity and block sensitivity for graph properties. They also tried to answer how low can sensitivity and block sensitivity go for graph properties?

In papers like [30, 14, 28, 16] it has been studied how low can the combinatorial measures go for transitive functions. The behavior of transitive functions can be very different from general Boolean functions. For example, while it is known that there are Boolean functions for which the sensitivity is as low as $\Theta(\log n)$ where n is the number of effective variables[3], it is known (from [28] and [21]) that if f is a transitive function on n effective variables then its sensitivity $s(f)$ is at least $\Omega(n^{1/12})$[4]. Similar behavior can be observed in other measures too. For example, it is easy to see that for a transitive function, the certificate complexity is $\Omega(\sqrt{n})$, while the certificate complexity for a general Boolean function can be as low as $O(\log n)$. Please see the full version of this paper [15] for a more detailed study.

A natural related question is:

What are tight relationships between various pairs of combinatorial measures for transitive functions?

By definition, the known relationship results for general functions hold for transitive functions. But tighter relationships may be obtained for transitive functions. On the other hand, the existing separations don't extend easily since the example used to demonstrate separation between certain pairs of measures may not be transitive. Some of the most celebrated examples are not transitive. For example some of the celebrated function construction like the pointer function in [4], used for demonstrating tight separations between various pairs like $\mathsf{D}(f)$ and $\mathsf{R}_0(f)$, are not transitive. Similarly, the functions constructed using the cheat sheet techniques [1] used for separation between quantum query complexity and degree, or approximate degree, are not transitive. Constructing transitive functions which demonstrate tight separations between various pairs of combinatorial measures is very challenging.

Our Results. We try to answer the above question for various pairs of measures. More precisely, our main contribution is to construct transitive functions that have similar complexity measures as the *pointer functions*. Hence for those pairs of measures where pointer functions can demonstrate separation for general functions, we prove that transitive functions can also demonstrate similar separation.

Our results and the current known relations between various pairs of complexity measures of transitive functions are compiled in Table 1. This table is along the lines of the table in [2] where the best-known relations between various complexity measures of general Boolean functions were presented.

Deterministic query complexity and zero-error randomized query complexity are two of the most basic measures and yet the tight relation between these measures was not known until recently. In [27] they showed that for the "balanced NAND-tree" function, $\widetilde{\wedge}$-tree, $\mathsf{D}(\widetilde{\wedge}\text{-tree}) \geq \mathsf{R}_0(\widetilde{\wedge}\text{-tree})^{1.33}$. Although the function $\widetilde{\wedge}$-tree is transitive, the best-known relationship was quadratic, that is for all Boolean function f, $\mathsf{D}(f) = O(\mathsf{R}_0(f)^2)$. In [4] a new function, A1, was constructed for which deterministic query complexity and zero-error randomized query complexity can have a quadratic separation between them, and this matched the known relationship results. The function in [4] was a variant of the pointer

[3] A variable is effective if the function is dependent on it.

[4] It is conjectured that the sensitivity of a transitive function is $\Omega(n^{1/3})$.

functions - a class of functions introduced by [20] that has found extensive usage in showing separations between various complexity measures of Boolean functions. The function, A1, also gave (the current best-known) separations between deterministic query complexity and other measures like quantum query complexity and degree. But the function A1 is not transitive. Using the A1 function we construct a transitive function that demonstrates a similar gap between deterministic query complexity and zero-error randomized query complexity, quantum query complexity, and degree.

▶ **Theorem 1.** *There exists a transitive function F_1 such that*

$$\mathsf{D}(F_1) = \widetilde{\Omega}(\mathsf{Q}(F_1)^4), \qquad \mathsf{D}(F_1) = \widetilde{\Omega}(\mathsf{R_0}(F_1)^2), \qquad \mathsf{D}(F_1) = \widetilde{\Omega}(\deg(F_1)^2).$$

The proof of Theorem 1 is presented in Section 4. In [4, 9] various variants of the pointer function have been used to show separation between other pairs of measures like $\mathsf{R_0}$ with R, $\mathsf{Q_E}$, deg, and Q, R with $\widetilde{\deg}$, deg, $\mathsf{Q_E}$ and sensitivity. Inspired by these functions, we construct transitive versions that demonstrate similar separation for transitive functions as general functions.

▶ **Theorem 2.** *There exists a transitive function F_2 such that*

$$\mathsf{R_0}(F_2) = \widetilde{\Omega}(\mathsf{R}(F_2)^2), \qquad \mathsf{R_0}(F_2) = \widetilde{\Omega}(\mathsf{Q_E}(F_2)^2), \qquad \mathsf{R_0}(F_2) = \widetilde{\Omega}(\deg(F_2)^2).$$

▶ **Theorem 3.** *There exists a transitive function F_3 such that*

$$\mathsf{R}(F_3) = \widetilde{\Omega}(\widetilde{\deg}(F_3)^4), \qquad \mathsf{R}(F_3) = \widetilde{\Omega}(\deg(F_3)^2).$$

The construction of these functions, though more complicated and involved, are similar in flavor to that of F_1. Due to lack of space, we skip the proofs of Theorem 2 and 3 in this conference version of this paper. The proofs are available in the full version of this paper [15]. Using standard techniques, we can also obtain the following theorems as corollaries to Theorem 3.

▶ **Theorem 4.** *There exists a transitive function F_4 such that $\mathsf{R_0}(F_4) = \widetilde{\Omega}(\mathsf{Q}(F_4)^3)$.*

▶ **Theorem 5.** *There exists a transitive function F_5 such that $\mathsf{R}(F_5) = \widetilde{\Omega}(\mathsf{Q_E}(F_5)^{1.5})$.*

▶ **Theorem 6.** *There exists transitive functions F_6 such that $\mathsf{R}(F_6) = \widetilde{\Omega}(\mathsf{s}(F_6)^3)$.*

Our proof techniques also help us make transitive versions of other functions like that used in[1] to demonstrate the gap between Q and certificate complexity.

▶ **Theorem 7.** *There exists a transitive function F_7 such that $\mathsf{Q}(F_7) = \widetilde{\Omega}(\mathsf{C}(F_7)^2)$.*

All our results are compiled (and marked in green) in Table 1.

One would naturally ask what stops us from constructing transitive functions analogous to the other functions, like cheat sheet-based functions. In fact, one could ask why to use ad-hoc techniques to construct transitive functions (as we have done in most of our proofs) and instead why not design a unifying technique for converting any function into a transitive function that would display similar properties in terms of combinatorial measures [5]. If one could do so, all the separation results for general functions (in terms of separation between pairs of measures) would translate to separation for transitive functions. In Section 5 we have discussed why such a task is challenging. We argue the challenges of making transitive versions of the cheat-sheet functions.

[5] In [7] they have demonstrated a technique that can be used for constructing a transitive partial function that demonstrates gaps (between certain combinatorial measures) similar to a given partial function that need not be transitive. But their construction need not construct a total function even when the given function is total.

Table 1 best-known separations between combinatorial measures for transitive functions.

	D	R_0	R	C	RC	bs	s	λ	Q_E	deg	Q	$\widetilde{\text{deg}}$
D		2 ; 2 T:1	2 ; 3 T:1	2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	3 ; 6 T:6	4 ; 6 T:3	2 ; 3 T:2	2 ; 3 T:1	4 ; 4 T:1	4 ; 4 T:3
R_0	1,1 $\oplus$		2 ; 2 T:2	2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	3 ; 6 T:6	4 ; 6 T:3	2 ; 3 T:2	2 ; 3 T:2	3 ; 4 T:4	4 ; 4 T:3
R	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	3 ; 6 T:6	4 ; 6 T:3	1.5 ; 3 T:5	2 ; 3 T:3	2 ; 4 $\wedge$	4 ; 4 T:3
C	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 2 $\oplus$		2 ; 2 [18]	2 ; 2 [18]	2 ; 5 [26]	2 ; 6 $\wedge$	1.15 ; 3 [3]	1.63 ; 3 [25]	2 ; 4 $\wedge$	2 ; 4 $\wedge$
RC	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		1.5 ; 2 [18]	2 ; 4 [26]	2 ; 4 $\wedge$	1.15 ; 2 [3]	1.63 ; 2 [25]	2 ; 2 $\wedge$	2 ; 2 $\wedge$
bs	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 4 [26]	2 ; 4 $\wedge$	1.15 ; 2 [3]	1.63 ; 2 [25]	2, 2 $\wedge$	2 ; 2 $\wedge$
s	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 2 $\wedge$	1.15 ; 2 [3]	1.63 ; 2 [25]	2, 2 $\wedge$	2 ; 2 $\wedge$
λ	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$
Q_E	1 ; 1 $\oplus$	1.33 ; 2 $\widetilde{\wedge}$-tree	1.33 ; 3 $\widetilde{\wedge}$-tree	2 ; 2 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 3 $\wedge\circ\vee$	2 ; 6 T:7	2 ; 6 T:7		1 ; 3 $\oplus$	2 ; 4 $\wedge$	1 ; 4 $\oplus$
deg	1 ; 1 $\oplus$	1.33 ; 2 $\widetilde{\wedge}$-tree	1.33 ; 2 $\widetilde{\wedge}$-tree	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge\circ\vee$	2 ; 2 $\wedge$	1 ; 1 $\oplus$		2 ; 2 $\wedge$	2 ; 2 $\wedge$
Q	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	2 ; 2 T:7	2 ; 3 T:7	2 ; 3 T:7	2 ; 6 T:7	2 ; 6 T:7	1, 1 $\oplus$	1 ; 3 $\oplus$		1 ; 4 $\oplus$
$\widetilde{\text{deg}}$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 2 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	

(1) Entry $a; b$ in row A and column B represents: for any *transitive* function f, $A(f) = O(B(f))^{b+o(1)}$, and there exists a *transitive* function g such that $A(g) = \Omega(B(g))^a$.
(2) Cells with a green background are those for which we constructed new *transitive functions* to demonstrate separations that match the best-known separations for general functions. The previously known functions that gave the strongest separations were not transitive. The second row (in each cell) gives the reference to the Theorems where the separation result is proved. Although for these green cells, the bounds match that of the general functions, for some cells (with a light green color), there is a gap between the known relationships and best-known separations.
(3) In the cells with a white background, the best-known examples for the corresponding separation were already *transitive functions*. For these cells, the second row either contains the function that demonstrates the separation or a reference to the paper where the separation was proved. So for these cells, the separations for transitive functions matched the current best-known separations for general functions. Note that for some of these cells, the bounds are not tight for general functions.
(4) Cells with a yellow background are those where the best-known separations for transitive functions do not match the best-known separations for general functions.

2 Notations and Background

2.1 Notations and basic definitions

We use $[n]$ to denote the set $\{1, \ldots, n\}$. $\{0,1\}^n$ denotes the set of all n-bit binary strings. For any $X \in \{0,1\}^n$ the Hamming Weight of X (denoted $|X|$) will refer to the number of 1 in X. 0^n and 1^n denotes all 0's string of n-bit and all 1's string of n-bit, respectively.

We denote by S_n the set of all permutations on $[n]$. Given an element $\sigma \in \mathsf{S}_n$ and a n-bit string $x_1, \ldots, x_n \in \{0,1\}^n$ we denote by $\sigma[x_1, \ldots, x_n]$ the string obtained by permuting the indices according to σ. That is $\sigma[x_1, \ldots, x_n] = x_{\sigma(1)}, \ldots, x_{\sigma(n)}$. This is also called the action of σ on the $x_1, \ldots, x_n$.

Following are a couple of interesting elements of S_n that will be used in this paper.

▶ **Definition 8.** *For any $n = 2k$ the* flip *swaps $(2i-1)$ and $2i$ for all $1 \le i \le k$. The permutation* $\mathsf{Swap}_{\frac{1}{2}}$ *swaps i with $(k+i)$, for all $1 \le i \le k$. That is,*

$$\mathsf{flip} = (1,2)(3,4)\ldots(n-1,n) \qquad \& \qquad \mathsf{Swap}_{\frac{1}{2}}[x_1, \ldots, x_{2k}] = x_{k+1}, \ldots, x_{2k}, x_1 \ldots, x_k.$$

Every integer $\ell \in [n]$ has the canonical $\log n$ bit string representation. However the number of 1's and 0's in such a representation is not same for all $\ell \in [n]$. The following representation of $\ell \in [n]$ ensures that for all $\ell \in [n]$ the encoding has same Hamming weight.

▶ **Definition 9** (Balanced binary representation)**.** *For any $\ell \in [n]$, let $\ell_1, \ldots, \ell_{\log n}$ be the binary representation of the number ℓ where $\ell_i \in \{0,1\}$ for all i. Replacing* 1 *by* 10 *and* 0 *by* 01 *in the binary representation of ℓ, we get a $2\log n$-bit unique representation, which we call* Balanced binary representation *of ℓ and denote as $bb(\ell)$.*

In this paper all the functions considered are of form $F : \{0,1\}^n \to \{0,1\}^k$. By Boolean functions we would mean a Boolean valued function that is of the form $f : \{0,1\}^n \to \{0,1\}$.

An input to a function $F : \{0,1\}^n \to \{0,1\}^k$ is a n-bit string but also the input can be thought of as different objects. For example, if the $n = NM$ then the input may be thought of as a $(N \times M)$-matrix with Boolean values. It may also be thought of as a $(M \times N)$-matrix.

If $\Sigma = \{0,1\}^k$ then $\Sigma^{(n\times m)}$ denotes an $(n \times m)$-matrix with an element of Σ (that is, a k-bit string) stored in each cell of the matrix. Note that $\Sigma^{(n\times m)}$ is actually $\{0,1\}^{mnk}$. Thus, a function $F : \Sigma^{(n\times m)} \to \{0,1\}$ is actually a Boolean function from a $\{0,1\}^{nmk}$ to $\{0,1\}$, where we think of the input as an $(n \times m)$-matrix over the alphabet Σ.

One particular nomenclature that we use in this paper is that of 1-cell certificate.

▶ **Definition 10** (1-cell certificate)**.** *Given a function $f : \Sigma^{(n\times m)} \to \{0,1\}$ (where $\Sigma = \{0,1\}^k$) the 1-cell certificate is a partial assignments to the cells which forces the value of the function to* 1*. So a 1-cell certificate is of the form $(\Sigma \cup \{*\})^{(n\times m)}$. Note that here we assume that the contents in any cell is either empty or a proper element of Σ (and not a partial k-bit string).*

Another notation that is often used is the following:

▶ **Notation 11.** *If $A \leq \mathsf{S}_n$ and $B \leq \mathsf{S}_m$ are groups on $[n]$ and $[m]$ then the group $A \times B$ act on the cells on the matrix. Thus for any $(\sigma, \sigma') \in A \times B$ and a $M \in \Sigma^{(n\times m)}$ by $(\sigma, \sigma')[M]$ we would mean the permutation on the cell of M according to (σ, σ') and move the contains in the cells accordingly. Note that the relative position of bits within the contents in each cell is not touched.*

Next, we define the composition of two Boolean functions.

▶ **Definition 12** (Composition of functions)**.** *Let $f : \{0,1\}^{nk} \to \{0,1\}$ and $g : \{0,1\}^m \to \{0,1\}^k$ be two functions. The* composition of f and g*, denoted by $f \circ g : \{0,1\}^{nm} \to \{0,1\}$, is defined to be a function on nm bits such that on input $x = (x_1, \ldots, x_n) \in \{0,1\}^{nm}$, where each $x_i \in \{0,1\}^m$, $f \circ g(x_1, \ldots, x_n) = f(g(x_1), \ldots, g(x_n))$. We will refer f as* outer function *and g as* inner function*.*

2.2 Transitive Groups and Transitive Functions

The central objects in this paper are transitive Boolean function. We first define transitive groups.

▶ **Definition 13.** *A group $G \leq \mathsf{S}_n$ is transitive if for all $i, j \in [n]$ there exists a $\sigma \in G$ such that $\sigma(i) = j$.*

▶ **Definition 14.** *For $f : A^n \to \{0,1\}$ and $G \leq \mathsf{S}_n$ we say f is invariant under the action of G, if for all $\alpha_1, \ldots, \alpha_n \in A$.*

$$f(\alpha_1, \ldots, \alpha_n) = f(\alpha_{\sigma(1)}, \ldots, \alpha_{\sigma(n)}).$$

▶ **Observation 15.** *If $A \leq \mathsf{S}_n$ and $B \leq \mathsf{S}_m$ are transitive groups groups on $[n]$ and $[m]$ then the group $A \times B$ is a transitive group acting on the cells on the matrix.*

There are many interesting transitive groups. The symmetric group is indeed transitive. The graph isomorphism group (that acts on the adjacency matrix - minus the diagonal - of a graph by changing the ordering on the vertices) is transitive. The cyclic permutation over all the points in the set is a transitive group. The following is another non-trivial transitive group on $[k]$ that we will use extensively in this paper.

▶ **Definition 16.** *For any k that is a power of 2, the Binary-tree-transitive group Bt_k is a subgroup of S_k. To describe its generating set we think of group Bt_k acting on the elements $\{1, \ldots, k\}$ and the elements are placed in the leaves of a balanced binary tree of depth $\log k$ - one element in each leaf. Each internal node (including the root) corresponds to an element in the generating set of Bt_k. The element corresponding to an internal node in the binary tree swaps the left and right sub-tree of the node. The permutation element corresponding to the root node is called the Root-swap as it swaps the left and right sub-tree to the root of the binary tree.*

We now state two claims whose proofs we skip in this version of the paper but are available in the full version of the paper [15].

▷ Claim 17. The group Bt_k is a transitive group.

The following claim describes how the group Bt_k acts on various encoding of integers. Recall the balance-binary representation (Definition 9).

▷ Claim 18. For all $\widehat{\gamma} \in \mathsf{Bt}_{2\log n}$ there is a $\gamma \in \mathsf{S}_n$ such that for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

Now let us consider another encoding that we will using for the set of rows and columns of a matrix.

▶ **Definition 19.** *Given a set R of n rows $r_1, \ldots, r_n$ and a set C of n columns $c_1, \ldots, c_n$ we define the balanced-pointer-encoding function $\mathcal{E} : (R \times \{0\}) \cup (\{0\} \times C) \to \{0,1\}^{4\log n}$, as follows:*

$$\mathcal{E}(r_i, 0) = bb(i) \cdot 0^{2\log n}, \text{ and, } \mathcal{E}(0, c_j) = 0^{2\log n} \cdot bb(j).$$

The following is a claim that is easy to verify.

▷ Claim 20. Let R be a set of n rows $r_1, \ldots, r_n$ and C be a set of n columns $c_1, \ldots, c_n$ and consider the *balanced-pointer-encoding function* $\mathcal{E} : (R \times \{0\}) \cup (\{0\} \times C) \to \{0,1\}^{4\log n}$. For any elementary permutation $\widehat{\sigma}$ in $\mathsf{Bt}_{4\log n}$ (other than the Root-swap) there is a $\sigma \in \mathsf{S}_n$ such that for any $(r_i, c_j) \in (R \times \{0\}) \cup (\{0\} \times C)$

$$\widehat{\sigma}[\mathcal{E}(r_i, c_j)] = \mathcal{E}(r_{\sigma(i)}, c_{\sigma(j)}),$$

where we assume $r_0 = c_0 = 0$ and any permutation of in S_n sends 0 to 0.

If $\widehat{\sigma}$ is the root-swap then for any $(r_i, c_j) \in (R \times \{0\}) \cup (\{0\} \times C)$

$$\widehat{\sigma}[\mathcal{E}(r_i, c_j)] = \mathsf{Swap}_{\frac{1}{2}}(\mathcal{E}(r_i, c_j)) = \mathcal{E}(c_j, r_i).$$

2.3 Pointer function

For the sake of completeness first we will describe the "pointer function" introduced in [4] that achieves separation between several complexity complexity measures like *Deterministic query complexity*, *Randomized query complexity*, *Quantum query complexity* etc. This function was originally motivated from a function in [20]. There are three three variants of the pointer function that have some special kind of non-Boolean domain, which we call *Pointer matrix*. Our function is a special "encoding" of that non-Boolean domain such that the resulting function becomes transitive and achieves the separation between complexity measures that matches the known separation between the general functions. Here we will define only the first variant of the pointer function.

▶ **Definition 21** (Pointer matrix over Σ)**.** *For $m, n \in \mathbb{N}$, let M be a $(m \times n)$ matrix with m rows and n columns. We refer to each of the $m \times n$ entries of M as* cells*. Each cell of the matrix is from a alphabet set Σ where $\Sigma = \{0,1\} \times \widetilde{P} \times \widetilde{P} \times \widetilde{P}$ and $\widetilde{P} = \{(i,j) | i \in [m], j \in [n]\} \cup \{\perp\}$. We call $\widetilde{P}$ as set of pointers where, pointers of the form $\{(i,j) | i \in [m], j \in [n]\}$ pointing to the cell (i,j) and $\perp$ is the null pointer. Hence,each entry $x_{(i,j)}$ of the matrix M is a 4-tuple from Σ. The elements of the 4-tuple we refer as* value*,* left pointer*,* right pointer *and* back pointer *respectively and denote by Value$(x_{(i,j)})$, LPointer$(x_{(i,j)})$, RPointer$(x_{(i,j)})$ and BPointer$(x_{(i,j)})$ respectively where Value $\in \{0,1\}$, LPointer, RPointer, BPointer $\in \widetilde{P}$. We call this type of matrix as* pointer matrix *and denote by $\Sigma^{n \times n}$.*

A special case of the pointer-matrix, which we call Type_1 pointer matrix over Σ*, is when for each cell of M, BPointer $\in \{[n] \cup \perp\}$ that is backpointers are pointing to the columns of the matrix.*

Also, in general when, BPointer $\in \{(i,j) | i \in [m], j \in [n]\} \cup \{\perp\}$, we call it a Type_2 pointer matrix over Σ.

Now we will define some additional properties of the domain that we need to define the pointer function.

▶ **Definition 22** (Pointer matrix with marked column)**.** *Let M be an $m \times n$* pointer-matrix *over Σ. A column $j \in [n]$ of M is defined to be a* marked column *if there exists exactly one cell (i,j), $i \in [m]$, in that column with entry $x_{(i,j)}$ such that $x_{(i,j)} \neq (1, \perp, \perp, \perp)$ and every other cell in that column is of the form $(1, \perp, \perp, \perp)$. The cell (i,j) is defined to be the* special element *of the* marked column *j.*

Let n be a power of 2. Let T be a rooted, directed and balanced binary tree with n-leaves and $(n-1)$ internal vertices. We will use the following notations that will be used in defining some functions formally.

▶ **Notation 23.** *Let n be a power of 2. Let T be a rooted, directed and balanced binary tree with n-leaves and $(n-1)$ internal vertices. Labels the edges of T as follows: the outgoing edges from each node are labeled by either $left$ or $right$. The leaves of the tree are labeled by the elements of $[n]$ from left to right, with each label used exactly once. For each leaf $j \in [n]$ of the tree, the path from the root to the leaf j defines a sequence of $left$ and $right$ of length $O(\log n)$, which we denote by $T(j)$.*

When n is not a power of 2, choose the largest $k \in \mathbb{N}$ such that $2^k \leq n$, consider a complete balanced tree with 2^k leaves and add a pair of child node to to each $n - 2^k$ leaves starting from left. Define $T(j)$ as before.

Now we are ready to describe the *Variant 1* of the pointer function.

▶ **Definition 24** (Variant 1 [4]). *Let* $\Sigma^{m\times n}$ *be a* Type_1 pointer matrix *where BPointer is a pointer of the form* $\{j|j \in [n]\}$ *that points to other column and LPointer, RPointer are as usual points to other cell. Define* $\mathsf{A1}_{(m,n)} : \Sigma^{m\times n} \to \{0,1\}$ *on a* Type_1 pointer matrix *such that for all* $x = (x_{i,j}) \in \Sigma^{m\times n}$*, the function* $\mathsf{A1}_{(m,n)}(x_{i,j})$ *evaluates to* 1 *if and only if it has a* 1- cell certificate *of the following form:*

1. *there exists exactly one* marked column $j^\star$ *in* M*,*
2. *There is a special cell, say* $(i^\star, j^\star)$ *which we call the special element in the the* marked column $j^\star$ *and there is a balanced binary tree* T *rooted at the special cell,*
3. *for each non-marked column* $j \in [n] \setminus \{j^\star\}$ *there exist a* cell l_j *such that* $Value(l_j) = 0$ *and* $BPointer(l_j) = j^\star$ *where* l_j *is the end of the path that starts at the* special element *and follows the pointers LPointer and RPointer as specified by the sequence* $T(j)$*.* l_j *exists for all* $j \in [n] \setminus \{j^\star\}$ *i.e. no pointer on the path is* $\perp$*. We refer* l_j *as the leaves of the tree.*

The above function achieves the separation between D vs. R_0 and D vs. Q for $m = 2n$. Here we will restate some of the results from [4] which we will use to prove the results for our function:

▶ **Theorem 25** ([4]). *The function* $\mathsf{A1}_{(m.n)}$ *in Definition 24 satisfies*

$\mathsf{D} = \Omega(n^2)$ *for* $m = 2n$ *where* $m, n \in \mathbb{N}$,

$\mathsf{R}_0 = \widetilde{O}(m+n)$ *for any* $m, n \in \mathbb{N}$,

$\mathsf{Q} = \widetilde{O}(\sqrt{m}+\sqrt{n})$ *for any* $m, n \in \mathbb{N}$.

Though [4] gives the deterministic lower bound for the function $\mathsf{A1}$ precisely for $2m \times m$ matrices following the same line of argument it can be proved that $\mathsf{D}(\Omega(n^2))$ holds for $n \times n$ matrices also. For sake of completeness we give a proof for $n \times n$ matrices in the full version of this paper [15].

▶ **Theorem 26.** $\mathsf{D}(\mathsf{A1}_{(n,n)}) = \Omega(n^2)$.

Also [20]'s function realises quadratic separation between D and deg and the proof goes via UC_{min} upper bound. But $\mathsf{A1}_{(n,n)}$ exhibits the same properties corresponding to UC_{min}. So, from the following observation it follows that $\mathsf{A1}_{(n,n)}$ also achieves quadratic separation between D and deg.

▶ **Observation 27.** $\deg(\mathsf{A1}_{(n,n)}) = O(n)$ *for any* $n \in \mathbb{N}$.

Another important observation that we need is the following:

▶ **Observation 28** ([4]). *For any input* $\Sigma^{n\times n}$ *to the function* $\mathsf{A1}_{(n,n)}$ *(in Definition 24) if we permute the rows of the matrix using a permutation* σ_r *and permute the columns of the matrix using a permutation* σ_c *and we update the pointers in each of the cells of the matrix accordingly then the function value does not change.*

3 High level description of our techniques

Pointer functions are defined over a special domain called *pointer matrix*, which is a $m \times n$ grid matrix. Each cell of the matrix contains some labels and some pointers that point either to some other cell or to a row or column [6]. As described in [20], the high level idea of pointer

[6] We naturally think of a pointer pointing to a cell as two pointers - one pointing to the row and the other to the column.

functions is the usage of pointers to make certificates unambiguous without increasing the input size significantly. This technique turns out to be very useful to give separations between various complexity measures as we see in [24], [19] and [4].

Now we want to produce a new function that possesses all the properties of pointer functions, along with the additional property of being transitive. To do so, first, we will encode the labels so that we can permute the bits (by a suitable transitive group) while keeping the structure of unambiguous certificates intact so that the function value remains invariant. One such natural technique would be to encode the contents of each cell in such a way that allows us to permute the bits of the contents of each cell using a transitive group and permute the cells among each other using another transitive group, and doing all of these while ensuring the unambiguous certificates remains intact [7]. This approach has a significant challenge: namely how to encode the pointers.

The information stored in each cell (other than the pointers) can be encoded using fixed logarithmic length strings of different Hamming weights - so that even if the strings are permuted and/or the bits in each string are permuted, the content can be "decoded". Unfortunately, this can only be done when the cell's contents have a constant amount of information - which is the case for pointer functions (except for the pointers). Since the pointers in the cell are strings of size $O(\log n)$ (as they are pointers to other columns or rows), if we want to use the similar Hamming weight trick, the size of the encoding string would need to be polynomial in $O(n)$. That would increase the size of the input compared to the unambiguous certificate. This would not give us tight separation results.

Also, there are three more issues concerning the encodings of pointers:

- As we permute the cells of the matrix according to some transitive group, the pointers within each cell need to be appropriately changed. In other words, when we move some cell's content to some other cell, the pointers pointing to the previous cell should point to the current cell now.
- If a pointer is encoded using a certain t-bit string, different permutations of bits of the encoded pointer can only generate a subset of all t-bit strings.
  For example: if we encode a pointer using a string of Hamming weight 10 then however we permute the bits of the string, the pointer can at most be modified to point to cells (or rows or columns) the encoding of whose pointers also have Hamming weight 10. (The issue is that permuting the bits of a string cannot change the Hamming weight of a string).
  The encoding of all the pointers should have the same Hamming weight.
- The encoding of the pointers has to be transitive. That is, we should be able to permute the bits of the encodings of the pointer using a transitive group in such a way that either the pointer value does not change or as soon as the pointer values changes, the cells gets permuted accordingly - kind of like an "entanglement".

The above three problems are somewhat connected. Our first innovative idea is to use *binary balance representation* (Definition 9) to represent the pointers. This way, we take care of the second issue. For the first and third issues, we define the transitive group – both the group acting on the contents of the cells (and hence on the encoding of the pointers) and the group acting on the cells itself – in a "entangled" manner. For this we induce a group action

[7] Here, we use the word "encode" since we can view the function defined only over codewords, and when the input is not a codeword, then it evaluates to 0. In our setting, since we are trying to preserve the one-certificates, the codewords are those strings where the unambiguous certificate is encoded correctly. At the same time, we must point out that the encoding of an unambiguous certificate is not necessarily unique.

acting on the nodes of a *balanced binary tree* and generate a transitive subgroup in S_n and $S_{2\log n}$ with the same action which will serve our purpose (Definition 16, Claim 18). This helps us to permute the rows (or columns) using a permutation while updating the encoding of the pointers accordingly.

By Claim 18, for every allowed permutation σ acting on the rows (or columns), there is a unique $\widehat{\sigma}$ acting on the encodings of the pointers in each of the cells such that the pointers are updated according to σ. This still has a delicate problem. Namely, each pointer is either pointing to a row or column. But the permutation $\widehat{\sigma}$ has no way to understand whether the encoding on which it is being applied points to a row or column. To tackle this problem, we think of the set of rows and columns as a single set. All of them are encoded by a string of size (say) $2t$, where for the rows, the second half of the encoding is all 0 while the columns have the first t bits all 0. This is the encoding described in Definition 19 using binary balanced representation. However, this adds another delicate issue about permuting between the first t bits of the encoding and the second t bits.

To tackle this problem, we modify the original function appropriately. We define a slightly modified version of existing pointer functions called $\mathsf{ModA1}$. This finally helps us obtain our "transitive pointer function," which has almost the same complexities as the original pointer function.

We have so far only described the high-level technique to make the 1st variation of pointer functions (Definition 24) transitive where there is the same number of rows and columns. The further variations need more delicate handling of the encoding and the transitive groups - though the central idea is similar.

4 Proof of Theorem 1

4.1 Transitive Pointer Function F_1 for Theorem 1

Our function $F_1 : \Gamma^{n\times n} \to \{0,1\}$ is a composition of two functions - an outer function $\mathsf{ModA1}_{(n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ and an inner function $\mathsf{Dec} : \Gamma \to \bar{\Sigma}$. We will set Γ to be $\{0,1\}^{96\log n}$.

The outer function is a modified version of the $\mathsf{A1}_{(n,n)}$ - pointer function described in [4] (see Definition 24 for a description). The function $\mathsf{A1}_{(n,n)}$ takes as input a $(n\times n)$-matrix whose entries are from a set Σ and the function evaluates to 1 if a certain kind of 1-cell-certificate exists. Let us define a slightly modified function $\mathsf{ModA1}_{(n,n)} : \bar{\Sigma}^{n\times n} \to \{0,1\}$ where $\bar{\Sigma} = \Sigma \times \{\vdash, \dashv\}$. We can think of an input $A \in \bar{\Sigma}^{n\times n}$ as a pair of matrices $B \in \Sigma^{n\times n}$ and $C \in \{\vdash, \dashv\}^{n\times n}$. The function $\mathsf{ModA1}_{(n,n)}$ is defined as

$$\mathsf{ModA1}_{(n,n)}(A) = 1 \text{ iff } \begin{cases} \text{Either,} & \text{(i) } \mathsf{A1}_{(n,n)}(B) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate have } \vdash \text{ in the corresponding cells in } C \\ \text{Or,} & \text{(ii) } \mathsf{A1}_{(n,n)}(B^T) = 1, \text{ and, all the cells in the} \\ & \text{1-cell-certificate have } \dashv \text{ in the corresponding cells in } C^T \end{cases}$$

Note that both the two conditions (i) and (ii) cannot be satisfied simultaneously. From this it is easy to verify that the function $\mathsf{ModA1}_{(n,n)}$ has all the properties as $\mathsf{A1}_{(n,n)}$ as described in Theorem 25.

The inner function Dec (we call it a decoding function) is function from Γ to $\bar{\Sigma}$, where $\Gamma = 96\log n$. Thus our final function is

$$F_1 := \left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right) : \Gamma^{n\times n} \to \{0,1\}.$$

4.1.1 Inner Function Dec

The input to $\mathsf{A1}_{(n,n)}$ is a Type_1 pointer matrix $\Sigma^{n\times n}$. Each cell of a Type_1 pointer matrix contains a 4-tuple of the form (Value, LPointer, RPointer, BPointer) where Value is either 0 or 1 and LPointer, RPointer are pointers to the other cells of the matrix and BPointer is a pointer to a column of the matrix (or can be a null pointer also). Hence, $\Sigma = \{0,1\} \times [n]^2 \times [n]^2 \times [n]$. For the function $\mathsf{A1}_{(n,n)}$ it was assumed (in [4]) that the elements of Σ is encoded as a k-length[8] binary string in a canonical way.

The main insight for our function $F_1 := \left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right)$ is that we want to maintain the basic structure of the function $\mathsf{A1}_{(n,n)}$ (or rather of $\mathsf{ModA1}_{(n,n)}$) but at the same time we want to encode the $\bar{\Sigma} = \Sigma \times \{\vdash, \dashv\}$ in such a way that the resulting function becomes transitive. To achieve this, instead of having a unique way of encoding an element in $\bar{\Sigma}$ we produce a number of possible encodings[9] for any element in $\bar{\Sigma}$. The inner function Dec is therefore a decoding algorithm that given any proper encoding of an element in $\bar{\Sigma}$ will be able to decode it back.

For the ease of understanding we start by describing the possible "encodings" of $\bar{\Sigma}$, that is by describing the pre-images of any element of $\bar{\Sigma}$ in the function Dec.

"Encodings" of the content of a cell in $\bar{\Sigma}^{n\times n}$. We will encode any element of $\bar{\Sigma}$ using a string of size $96\log n$ bits. Recall that, an element in $\bar{\Sigma}$ is of the form $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$, where V is the Boolean value, (r_L, c_L), (r_R, c_R) and c_B are the left pointer, right pointers and bottom pointer respectively and T take the value $\vdash$ or $\dashv$. The overall summary of the encoding is as follows:

- **Parts:** We will think of the tuple as 7 objects, namely V, r_L, c_L, r_R, c_R, c_B and T. We will use $16\log n$ bits to encode each of the first 6 objects. The value of T will be encoded in a clever way. So the encoding of any element of $\bar{\Sigma}$ contains 6 *parts* - each a binary string of length $16\log n$.
- **Blocks:** Each of 6 *parts* will be further broken into 4 *blocks* of equal length of $4\log n$. One of the blocks will be a special block called the "encoding block".

Now we explain, for a tuple $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ what is the 4 blocks in each part. We will start by describing a "standard-form" encoding of a tuple $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ when $T = \vdash$. Then, we will extend it to describe the standard for encoding the tuple when $T = \dashv$. And finally we will explain all other valid encoding of the tuple by describing all the allowed permutations on the bits of the encoding.

Standard-form encoding of $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \vdash$. For the standard form of encoding we will assume that the information of $V, r_L, c_L, r_R, c_R, c_B$ are stored in parts $P1, P2, P3, P4, P5$ and $P6$ respectively. For all $i \in [6]$, the part P_i with have blocks B_1, B_2, B_3 and B_4, of which the block B_1 will be the encoding-block. The encoding will ensure that every parts within a cell will have distinct Hamming weight. The description is also compiled in the Table 2.

- For part $P1$ (that is the encoding of V) the encoding block B_1 will store $\ell_1 \cdot \ell_2$ where ℓ_1 be the $2\log n$ bit binary string with Hamming weight $2\log n$ and ℓ_2 is any $2\log n$ bit binary string with Hamming weight $2\log n - 1 - V$. The blocks B_2, B_3 and B_4 will store a $4\log n$ bit string that has Hamming weight $4\log n, 2\log n + 1$ and $2\log n + 2$ respectively. Any

[8] For the canonical encoding $k = (1 + 5\log n)$ was sufficient

[9] We use the term "encoding" a bit loosely in this context as technically an encoding means a unique encoding. What we actually mean is the pre-images of the function Dec.

Table 2 Standard form of encoding of element $(V,(r_L,c_L),(r_R,c_R),c_B,\vdash)$ by a $96\log n$ bit string that is broken into 6 parts $P_1,\ldots,P_6$ of equal size and each Part is further broken into 4 Blocks B_1,B_2,B_3 and B_4. So all total there are 24 blocks each containing a $4\log n$-bit string. For the standard form of encoding of element $(V,(r_L,c_L),(r_R,c_R),c_B,\dashv)$ we encode $(V,(r_L,c_L),(r_R,c_R),c_B,\vdash)$ in the standard form as described in the table and then apply the $\mathsf{Swap}_{\frac{1}{2}}$ on each block. The last column of the table indicates the Hamming weight of each Part.

...	B_1 "encoding"-block	B_2	B_3	B_4	Hamming weight
$P1$	$\ell_1\ell_2$, where $\lvert\ell_1\rvert = 2\log n$, and $\lvert\ell_2\rvert = 2\log n - 1 - V$	$4\log n$	$2\log n+1$	$2\log n+2$	$12\log n+2-V$
$P2$	$\mathcal{E}(r_L,0)$	$2\log n+3$	$2\log n+1$	$2\log n+2$	$7\log n+6$
$P3$	$\mathcal{E}(0,c_L)$	$2\log n+4$	$2\log n+1$	$2\log n+2$	$7\log n+7$
$P4$	$\mathcal{E}(r_R,0)$	$2\log n+5$	$2\log n+1$	$2\log n+2$	$7\log n+8$
$P5$	$\mathcal{E}(0,c_R)$	$2\log n+6$	$2\log n+1$	$2\log n+2$	$7\log n+9$
$P6$	$\mathcal{E}(0,c_B)$	$2\log n+7$	$2\log n+1$	$2\log n+2$	$7\log n+10$

fixed string with the correct Hamming weight will do. We are not fixing any particular string for the blocks B_2, B_3 and B_4 to emphasise the fact that we will be only interested in the Hamming weights of these strings.

- The encoding block $B1$ for parts $P2,P3,P4,P5$ and $P6$ will store the string $\mathcal{E}(r_L,0)$, $\mathcal{E}(0,c_L)$, $\mathcal{E}(r_R,0)$, $\mathcal{E}(0,c_r)$ and $\mathcal{E}(0,C_B)$ respectively, where $\mathcal{E}$ is the Balanced-pointer-encoding function (Definition 19). For part P_i (with $2\le i\le 6$) block B_2,B_3 and B_4 will store any $4\log n$ bit string with Hamming weight $2\log n+1+i$, $2\log n+1$ and $2\log n+2$ respectively.

Standard form encoding of $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T=\dashv$. For obtaining a standard-form encoding of $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T=\dashv$, first we encode $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T=\vdash$ using the standard-form encoding. Let $(P1,P2,\ldots,P6)$ be the standard-form encoding of $(V,(r_L,c_L),(r_R,c_R),(c_B),T)$ where $T=\vdash$. Now for each of the block apply the $\mathsf{Swap}_{\frac{1}{2}}$ operator.

Valid permutation of the standard form. Now we will give a set of valid permutations to the bits of the encoding of any element of $\bar{\Sigma}$. The set of valid permutations are classified into into 3 categories:

1. Part-permutation: The 6 parts can be permuted using any permutation from S_6
2. Block-permutation: In each of the part, the 4 blocks (say B_1,B_2,B_3,B_4) can be permuted is two ways. (B_1,B_2,B_3,B_4) can be send to one of the following:

   (a) Simple Block Swap: (B_3,B_4,B_1,B_2) (b) Block Flip: $(B_2,B_1,\mathsf{flip}(B_3),\mathsf{flip}(B_4))$

The "decoding" function $\mathbf{Dec}:\{0,1\}^{96\log n}\to\bar{\Sigma}$.

- Identify the parts containing the encoding of V, r_L, c_L, r_R, c_R and c_B. This is possible because every part has a unique Hamming weight.
- For each part identify the blocks. This is also possible as in any part all the blocks have distinct Hamming weight. Recall, the valid Block-permutations, namely Simple Block Swap and Block Flip. By seeing the positions of the blocks one can understand if flip was applied and to what and using that one can revert the blocks back to the standard-form (recall Definition 9).

- In the part containing the encoding of V consider the encoding-block. If the block is of the form $\{(\ell_1\ell_2)$ such that $|\ell_1| = 2\log n, |\ell_2| \leq 2\log n - 1\}$ then $T = \{\vdash\}$. If the block is of the form $\{(\ell_2\ell_1)$ such that $|\ell_1| = 2\log n, |\ell_2| \leq 2\log n - 1\}$ then $T = \{\dashv\}$.
- By seeing the encoding block we can decipher the original values and the pointers.
- If the $96\log n$ bit string doesn't have the form of a valid encoding, then decode it as $(0, \perp, \perp, \perp)$.

4.2 Proof of Transitivity of the function

We start with describing the transitive group for which F_1 is transitive.

The Transitive Group. We start with describing a transitive group $\mathcal{T}$ acting on the cells of the matrix A. The matrix has rows $r_1, \ldots, r_n$ and columns $c_1, \ldots, c_n$. And we use the encoding function $\mathcal{E}$ to encode the rows and columns. So the index of the rows and columns are encoded using a $4\log n$ bit string. A permutation from $\mathsf{Bt}_{4\log n}$ (see Definition 16) on the indices of a $4\log n$ bit string will therefore induce a permutation on the set of rows and columns which will give us a permutation on the cells of the matrix. We will now describe the group $\mathcal{T}$ acting on the cells of the matrix by describing the permutation group $\widehat{\mathcal{T}}$ acting on the indices of a $4\log n$ bit string. The group $\widehat{\mathcal{T}}$ will be the group $\mathsf{Bt}_{4\log n}$ acting on the set $[4\log n]$. We will assume that $\log n$ is a power of 2. The group $\mathcal{T}$ with be the resulting group of permutations on the cells of the matrix induced by the group $\widehat{\mathcal{T}}$ acting on the indices on the balanced-pointer-encoding. Note that $\mathcal{T}$ is acting on the domain of $\mathcal{E}$ and $\widehat{\mathcal{T}}$ is acting on the image of $\mathcal{E}$. Also $\widehat{\mathcal{T}}$ is a transitive subgroup of $\mathsf{S}_{4\log n}$ from Claim 17.

▶ **Observation 29.** *For any $1 \leq i \leq 2\log n$ consider the permutation "ith-bit-flip" in $\widehat{\mathcal{T}}$ that applies the transposition $(2i-1, 2i)$ to the indices of the balanced-pointer-encoding. Since the $\mathcal{E}$-encoding of the row $(r_k, 0)$ uses the balanced binary representation of k in the first half and all zero sting in the second half, the jth bit in the binary representation of k is stored in the $2j-1$ and $2j$-th bit in the $\mathcal{E}$-encoding of r_i. So the j-th-bit-flip acts on the sets of rows by swapping all the rows with 1 in the j-th bit of their index with the corresponding rows with 0 in the j-th bit of their index. Also, if $i > \log n$ then there is no effect of the i-th-bit-flip operation on the set of rows. Similarly for the columns.*

Using Observation 29 we have the following claim.

▷ Claim 30. The group $\mathcal{T}$ acting on the cells of of the matrix is a transitive group. That is, for all $1 \leq i_1, j_1, i_2, j_2 \leq n$ there is a permutation $\widehat{\sigma} \in \widehat{\mathcal{T}}$ such that $\widehat{\sigma}[\mathcal{E}(i_1, 0)] = \mathcal{E}(i_2, 0)$ and $\widehat{\sigma}[\mathcal{E}(0, j_1)] = (0, j_2)$. Or in other words, there is a $\sigma \in \mathcal{T}$ acting on the cell of the matrix that would take the cell corresponding to row r_{i_1} and column c_{j_1} to the cell corresponding to row r_{i_2} and column c_{j_2}.

From the Claim 30 we see the group $\mathcal{T}$ acting on the cells of of the matrix is a transitive. But it does not touch the contents within the cells of the matrix. But the input to the function F_1 contains element of $\Gamma = \{0,1\}^{96\log n}$ in each cell. So we now need to extend the group $\mathcal{T}$ to a group G that acts on all the indices of the bits of the input to the function F_1.

Recall that the input to the function F_1 is a $(n \times n)$-matrix with each cell of matrix containing a binary string of length $96\log n$ which has 6 parts of size $16\log n$ each and each part has 4 blocks of size $4\log n$ each. We classify the generating elements of the group G into 4 categories:

1. Part-permutation: In each of the cells the 6 parts can be permuted using any permutation from S_6

2. Block-permutation: In each of the Parts the 4 blocks can be permuted in the following ways. (B_1, B_2, B_3, B_4) can be send to one of the following
   a. Simple Block Swap: (B_3, B_4, B_1, B_2)
   b. Block Flip (#1): $(B_2, B_1, \mathsf{flip}(B_3), \mathsf{flip}(B_4))$
   c. Block Flip (#2)[10]: $(\mathsf{flip}(B_1), \mathsf{flip}(B_2), B_4, B_3)$
3. Cell-permutation: for any $\sigma \in \mathcal{T}$ the following two action has to be done simultaneously:
   a. (Matrix-update) Permute the cells in the matrix according to the permutation σ. This keeps the contents within each cells untouched - it just changes the location of the cells.
   b. (Pointer-update) For each of blocks in each of the parts in each of the cells permute the indices of the $4 \log n$-bit strings according to σ, that is apply $\widehat{\sigma} \in \widehat{\mathcal{T}}$ corresponding to σ.

We now have the following theorems that would prove that the function F_1 is transitive.

▶ **Theorem 31.** *G is a transitive group and the function F_1 is invariant under the action of the G.*

Proof of Theorem 31. To prove that the group G is transitive we show that for any indices $p, q \in [96n^2 \log n]$ there is a permutation $\sigma \in \mathsf{G}$ that would take p to q. Recall that the string $\{0,1\}^{96n^2 \log n}$ is a matrix $\Gamma^{(n \times n)}$ with $\Gamma = \{0,1\}^{96 \log n}$ and every element in Γ is broken into 6 parts and each part being broken into 4 block of size $4 \log n$ each. So we can think of the index p as sitting in k_pth position $(1 \leq k_p \leq 4 \log n)$ in the block B_p of the part P_p in the (r_p, c_p)-th cell of the matrix. Similarly, we can think of q as sitting in k_qth position $(1 \leq k_q \leq 4 \log n)$ in the block B_q of the part P_q in the (r_q, c_q)-th cell of the matrix.

We will give a step by step technique in which permutations from G can be applied to move p to q.

- **Step 1: Get the positions in the block correct:** If $k_p \neq k_q$ then take a permutation $\widehat{\sigma}$ from $\widehat{\mathcal{T}}$ that takes k_p to k_q. Since $\widehat{\mathcal{T}}$ is a transitive so such a permutation exists. Apply the cell-permutation $\sigma \in \mathcal{T}$ corresponding to $\widehat{\sigma}$. As a result the index p can be moved to a different cell in the matrix but, by the choice of $\widehat{\sigma}$ its position in the block in which it is will be k_q. Without loss of generality, we assume the the cell location does not change.
- **Step 2: Get the cell correct:** Using a cell-permutation that corresponds to a series of "bit-flip" operations change r_p to r_q and c_p to c_q. Since one bit-flip operations basically changes one bit in the binary representation of the index of the row or column such a series of operations can be made.
  Since each bit-flip operation is executed by applying the bit-flips in each of the blocks so this might have once again changed the position of the index p in the block. But, even if the position in the block changes it must be a flip operation away. Or in other word, since in the beginning of this step $k_p = k_q$, so if k_q is even (or odd) then after the series bit-flip operations the position of p in the block is either k_q or $(k_q - 1)$ (or $(k_q + 1)$).
- **Step 3: Align the Part:** Apply a suitable permutation to ensure that the part P_p moves to part P_q. Note this does not change the cell or the block within the part or the position in the block.

[10] Actually this Block flip can be generated by a combination of Simple Block Swap and Block Flip (#1)

- **Step 4: Align the Block:** Using a suitable combination of Simple Block Swap and Block Flip ensures the Block number gets matched, that is B_p goes to B_q. In this case the cell or the Part does not change. But depending on whether the Block Flip operation is applied the position in the block can again change. But, the current position in the block k_p is at most one flip away from k_q.
- **Step 5: Apply the final flip:** It might so happen that already we a done after the last step. If not we know that the current position in the block k_p is at most one flip away from k_q. So we apply the suitable Block-flip operation. Thus will not change the cell position, Part number, Block number and the position in the block will match.

Hence we have proved that the group G is transitive. Now we show that the the function F_1 is invariant under the action of G, i.e., for any elementary operations π from the group G and for any input $\Gamma^{(n\times n)}$ the function value does not change even if after the input is acted upon by the permutation π.

Case 1: π is a Part-permutation: It is easy to see that the decoding algorithm Dec is invariant under Part-permutation. This was observed in description of the decoding algorithm Dec in Section 4.1.1. So clearly that the function F_1 is invariant under any Part-permutation.

Case 2: π is a Block-permutation: Here also it is easy to see that the decoding algorithm Dec is invariant under Block-permutation. This was observed in description of the decoding algorithm Dec in Section 4.1.1. Thus F_1 is also invariant under any Block-permutation.

Case 3: π is a Cell-permutation From Observation 28 it is enough to prove that when we permute the cells of the matrix we update the points in the cells accordingly.

Let $\pi \in \mathcal{T}$ be a permutation that permutes only the rows of the matrix. By Claim 20, we see that the contents of the cells will be updated accordingly. Similarly if π only permute the columns of the matrix we will be fine.

Finally, if π swaps the row set and the column set (that is if π makes a transpose of the matrix) then for all i row i is swapped with column i and it is easy to see that $\widehat{\pi}[\mathcal{E}(i,0)] = \mathcal{E}(0,i)$. In that case the encoding block of the value part in a cell also gets swapped. This will thus be encoding the T value as $\dashv$. And so the function value will not be affected as the $T = \dashv$ will ensure that one should apply the π that swaps the row set and the column set to the input before evaluating the function. ◀

4.3 Properties of the Function

▷ Claim 32. Deterministic query complexity of F_1 is $\Omega(n^2)$.

Proof. The function $\mathsf{ModA1}_{(n,n)}$ is a "harder" function than $\mathsf{A1}_{(n,n)}$. So $\mathsf{D}(\mathsf{ModA1}_{(n,n)})$ is at least that of $\mathsf{D}(\mathsf{A1}_{(n,n)})$. Now since, F_1 is $\left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right)$ so clearly the $\mathsf{D}(F_1)$ is at least $\mathsf{D}(\mathsf{A1}_{(n,n)})$. Theorem 26 proves that $\mathsf{D}(\mathsf{A1}_{(n,n)})$ is $\Omega(n^2)$. Hence $\mathsf{D}(F_1) = \Omega(n^2)$. ◁

The following Claim 33 follows from the definition of the function $\mathsf{ModA1}_{(n,n)}$.

▷ Claim 33. The following are some properties of the function $\mathsf{ModA1}_{(n,n)}$

1. $\mathsf{R_0}(\mathsf{ModA1}_{(n,n)}) \leq 2\mathsf{R_0}(\mathsf{A1}_{(n,n)}) + O(n \log n)$
2. $\mathsf{Q}(\mathsf{ModA1}_{(n,n)}) \leq 2\mathsf{Q}(\mathsf{A1}_{(n,n)}) + O(n \log n)$
3. $\deg(\mathsf{ModA1}_{(n,n)}) \leq 2\deg(\mathsf{A1}_{(n,n)}) + O(n \log n)$

Finally, from "composition theorem" (formal proof of which is presented in the full version of the paper [15]) we see that the $\mathsf{R_0}(F_1)$, $\mathsf{Q}(F_1)$ and $\deg(F_1)$ are at most $O(\mathsf{R_0}(\mathsf{ModA1}_{(n,n)} \cdot \log n)$, $O(\mathsf{Q}(\mathsf{ModA1}_{(n,n)} \cdot \log n)$ and $O(\deg(\mathsf{ModA1}_{(n,n)} \cdot \log n)$, respectively. So combining this fact with Claim 32, Claim 33 and Theorem 25 (from [4]) we have Theorem 1.

5 Challenges in transitive versions of "cheat sheet" based functions

In this section we show that it is not possible to give a quadratic separation between degree and quantum query complexity for transitive functions by modifying the cheat sheet function using the techniques in [1] which go via unambiguous certificate complexity.

Let us start by recalling the cheat sheet framework from [1]. Let $f : \{0,1\}^n \to \{0,1\}$ be a total Boolean function. Let $C(f)$ be its certificate complexity and $Q(f)$ be its bounded-error quantum query complexity. We consider the following cheat sheet function, which we denote by $f_{CS,t} : \{0,1\}^{n\times\log t+t\times\log t\times C(f)\times\log n} \to \{0,1\}$:

- There are $\log t$ copies of f on disjoint sets on inputs denoted by $f_1, \ldots, f_{\log t}$.
- There are t cheat sheets: each cheat sheet is a block of $(\log t \times C(f) \times \log n)$ many bits
- Let $x_1, \ldots, x_{\log t} \in \{0,1\}^n$ denote the input to the $\log t$ copies of f and let $Y_1, \ldots, Y_t$ denote the t cheat sheets.
- Let $\ell = (f(x_1), \ldots, f(x_{\log t}))$. $f_{CS,t}$ evaluates to 1 if and only if Y_ℓ is a valid cheat sheet.

Separations between various complexity measures was shown in [1] using the cheat sheet framework. In [1], the separations that lower bound bounded-error quantum query complexity in terms of other complexity measures, for example degree, are obtained as follows:

1. Start with a total function $f : \{0,1\}^n \to \{0,1\}$ that has quadratic separation between quantum query complexity and certificate complexity: $Q(f) = \widetilde{\Omega}(n)$ and $C(f) = \widetilde{O}(\sqrt{n})$. Consider the cheat sheet version of this function $f_{CS,t}$, with $t = n^{10}$.
2. Lower bound $Q(f_{CS,t})$, for $t = n^{10}$, by $Q(f)$. This uses the hybrid method ([10]) and strong direct product theorem ([22]).
3. Upper bound degree of $f_{CS,t}$ by using the upper bound on the unambiguous certificate complexity of $f_{CS,t}$.

Instead of degree, one might use approximate degree in the third step above for a suitable choice of f (see [1] for details).

A natural approach to obtain a transitive function with gap between a pair of complexity measures is to modify the cheat sheet framework to make it transitive. One possible modification is to allow a poly-logarithmic blowup in the input size of the resulting transitive function while preserving complexity measures of the cheat sheet function that are of interest (upto poly-logarithmic factors).

We show, however, that it is not possible to obtain a quadratic separation between degree and quantum query complexity for transitive functions by modifying the cheat sheet function using the techniques in [1] which go via unambiguous certificate complexity. The reason for this is that the unambiguous certificate complexity of a transitive cheat sheet function on N-bits is $\Omega(\sqrt{N})$ (see Observation 34) whereas we show (see Lemma 35) that the quantum query complexity of such a function is $o(N)$.

Note that this does not mean that cheat sheet framework can not be made transitive to show such a quadratic gap. If the cheat sheet version of a function that is being made transitive has a better degree upper bound than that given by unambiguous certificate complexity then a better gap might be possible.

To formalize the above discussion we first need the following observation that lower bounds the certificate complexity of any transitive function.

▶ **Observation 34** ([30]). *Let $f : \{0,1\}^N \to \{0,1\}$ be a transitive function, then $C(f) \geq \sqrt{N}$.*

Next, we upper bound on quantum query complexity of cheat sheet function using quantum amplitude amplification ([11]). The details of proof of the following lemma can be found in the full version of this paper [15].

▶ **Lemma 35.** *The quantum query complexity of $f_{CS,t}$ is $O(\sqrt{t} \times \log t \times \sqrt{n} \times \log n)$.*

The cheat sheet version of f, $f_{CS,t}$, is a function on $\widetilde{\Theta}(n + C(f)t)$ many variables, where t is polynomial in n. From the cheat sheet property the unambiguous certificate complexity of $f_{CS,t}$, denoted by $\mathsf{UC}(f_{CS,t})$, is $\widetilde{\Theta}(C(f))$.

Let $\widetilde{f_{CS,t}}$ be a modified transitive version of $f_{CS,t}$ that preserves the quantum query complexity and certificate complexity of $f_{CS,t}$ upto poly-logarithmic factors, respectively. From Observation 34 it follows that $\mathsf{UC}(\widetilde{f_{CS,t}}) = \widetilde{\Omega}(\sqrt{n + C(f)t})$. On the other hand, since $\widetilde{f_{CS,t}}$ preserves the certificate complexity upto poly-logarithmic factors, $\mathsf{UC}(\widetilde{f_{CS,t}}) = \widetilde{O}(C(f))$. This implies that $t = \widetilde{O}(C(f))$. Lemma 35 that $Q(f_{CS,t})$ is at most $\widetilde{O}(C(f)\sqrt{t})$. Thus in order to achieve quadratic separation between UC and Q, t has to be $\widetilde{\Omega}(C(f)^2)$.

We end this section by giving a concrete approach towards showing separation between degree and quantum query complexity for a transitive functions using the cheat sheet method. We believe the it is possible to start with $f_{CS,t}$, for transitive function f and $t = \sqrt{n}$ and convert it to a transitive function that preserves the unambiguous certificate complexity and quantum query complexity upto poly-logarithmic factors, while incurring a poly-logarithmic blowup in the input size. However, we do not know how to prove quantum query complexity lower bound matching our upper bound from Lemma 35 for $t = \sqrt{n}$. We make the following conjecture towards this end, which, if true, implies that for a transitive function f, $Q(f) = \widetilde{\Omega}(\deg(f)^{4/3})$.

▶ **Conjecture 36.** *There exists a transitive function $f : \{0,1\}^n \to \{0,1\}$ with $C(f) = \widetilde{O}(\sqrt{n})$ and $Q(f) = \widetilde{\Omega}(n)$. Let $f_{CS,\sqrt{n}}$ be the cheat sheet version of f with $\sqrt{n}$ cheat sheets. Then $Q(f_{CS,\sqrt{n}}) = \Omega(n^{3/4})$.*

It was showed in [1] that the quantum query complexity of the cheat function $f_{CS,t}$, i.e. $Q(f_{CS,t})$, is lower bounded by $Q(f)$, when $t = n^{10}$. Their proof goes via he hybrid method ([10]) and strong direct product theorem ([22]). Is is interesting to find the the constant smallest c such that $Q(f_{CS,n^c}) = \Omega(Q(f))$. We know that such a c must be at least than 1 (from Lemma 35) and is at most 10 (from [1]). We state this formally below:

▶ **Question 37.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a non-constant Boolean function and let f_{CS,n^c} be its cheat sheet version with n^c cheat sheets. What is the smallest c such that the following is true $Q(f_{CS,n^c}) = \Omega(Q(f))$.*

6 Conclusion

As far as we know, this is the first paper that presents a thorough investigation on the relationships between various pairs of complexity measures for transitive function.

The current best-known relationships and best-known separations between various pairs of measures for transitive functions are summarized in the Table 1. Unfortunately, a number of cells in the table are not tight. In this context, we would like to point out some important directions:

- For some of these cells, the separation results for transitive functions are weaker than that of the general functions. A natural question is the following: why can't we design a transitive version of the general functions that achieve the same separation? For some cases, like the cheat sheet-based functions, we discuss the difficulties and possible directions in Section 5. Thus following is a natural question.

  ▶ **Open Problem 38.** *For a pair of complexity measures for Boolean functions whose best-known separations are achieved via cheat sheets, obtain similar separations for transitive Boolean functions.*

- A total function was constructed in [13] that demonstrates quadratic separations between approximate degree with sensitivity and several other complexity measures. It is thus natural to investigate the following open problem.

  ▶ **Open Problem 39.** *Come up with transitive functions that achieve similar separations for those pair of measures whose best-known separations are shown by [13].*

- Recently [8], [5] and [6] came up with new classes of Boolean functions, starting with the HEX (see [8]) and EAH (see [6]) functions, that exhibit improved separations between certificate complexity and other complexity measures using the.
  In light of these recent developments is important to ask whether similar separations can be shown for transitive functions. Following open problem is a natural starting point.

  ▶ **Open Problem 40.** *Can the HEX and EAH functions be modified to a transitive functions, while preserving its desired complexity measures upto poly-logarithmic factors?*

- While we have been concerned only with lower bounds in this paper, it is an exciting research direction to bridge the gap between complexity measures of transitive Boolean functions by providing improved upper bounds.

  ▶ **Open Problem 41.** *Bridge the gaps in Table 1 by coming up with better upper bounds on complexity measures for transitive functions.*

In the full version of this paper, [15], we summarize the results on how low can individual complexity measures go for transitive function. Even with the recent results of [21] and [2], there are significant gaps between the best-known lower and upper bounds in this case which gives another set of open problems to investigate in the study of combinatorial measures of transitive Boolean functions.

References

1 Scott Aaronson, Shalev Ben-David, and Robin Kothari. Separations in query complexity using cheat sheets. In *STOC*, pages 863–876, 2016. doi:10.1145/2897518.2897644.

2 Scott Aaronson, Shalev Ben-David, Robin Kothari, Shravas Rao, and Avishay Tal. Degree vs. approximate degree and quantum implications of Huang's sensitivity theorem. In *STOC*, pages 1330–1342, 2021. doi:10.1145/3406325.3451047.

3 Andris Ambainis. Superlinear advantage for exact quantum algorithms. *SIAM Journal on Computing*, pages 617–631, 2016. doi:10.1137/130939043.

4 Andris Ambainis, Kaspars Balodis, Aleksandrs Belovs, Troy Lee, Miklos Santha, and Juris Smotrovs. Separations in query complexity based on pointer functions. *Journal of the ACM*, 64(5):32:1–32:24, 2017. doi:10.1145/3106234.

5 Kaspars Balodis. Several separations based on a partial Boolean function. *CoRR*, 2021. arXiv:2103.05593.

6 Kaspars Balodis, Shalev Ben-David, Mika Göös, Siddhartha Jain, and Robin Kothari. Unambiguous DNFs and Alon-Saks-Seymour. In *FOCS*, pages 116–124, 2022. doi:10.1109/FOCS52979.2021.00020.

7 Shalev Ben-David, Andrew M. Childs, András Gilyén, William Kretschmer, Supartha Podder, and Daochen Wang. Symmetries, graph properties, and quantum speedups. In *FOCS*, pages 649–660, 2020. doi:10.1109/FOCS46700.2020.00066.

8 Shalev Ben-David, Mika Göös, Siddhartha Jain, and Robin Kothari. Unambiguous DNFs from Hex. *Electronic Colloquium on Computational Complexity*, 28:16, 2021.

9 Shalev Ben-David, Pooya Hatami, and Avishay Tal. Low-sensitivity functions from unambiguous certificates. In *ITCS*, volume 67, pages 28:1–28:23, 2017. doi:10.4230/LIPIcs.ITCS.2017.28.

10 Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997. doi:10.1137/S0097539796300933.

11 Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

12 Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, 2002. doi:10.1016/S0304-3975(01)00144-X.

13 Mark Bun and Justin Thaler. A nearly optimal lower bound on the approximate degree of AC^0. *SIAM Journal on Computing*, 49(4), 2020. doi:10.1137/17M1161737.

14 Sourav Chakraborty. On the sensitivity of cyclically-invariant Boolean functions. *Discrete Mathematics and Theoretical Computer Science*, 13(4):51–60, 2011. doi:10.46298/dmtcs.552.

15 Sourav Chakraborty, Chandrima Kayal, and Manaswi Paraashar. Separations between combinatorial measures for transitive functions. *ArXiv*, 2021. arXiv:2103.12355.

16 Andrew Drucker. Block sensitivity of minterm-transitive functions. *Theoretical Computer Science*, 412(41):5796–5801, 2011. doi:10.1016/j.tcs.2011.06.025.

17 Yihan Gao, Jieming Mao, Xiaoming Sun, and Song Zuo. On the sensitivity complexity of bipartite graph properties. *Theoretical Computer Science*, 468:83–91, 2013. doi:10.1016/j.tcs.2012.11.006.

18 Justin Gilmer, Michael E. Saks, and Srikanth Srinivasan. Composition limits and separating examples for some Boolean function complexity measures. *Combinatorica*, 36(3):265–311, 2016. doi:10.1007/s00493-014-3189-x.

19 Mika Göös, T. S. Jayram, Toniann Pitassi, and Thomas Watson. Randomized communication versus partition number. *ACM Transactions on Computation Theory*, 10(1):4:1–4:20, 2018. doi:10.1145/3170711.

20 Mika Göös, Toniann Pitassi, and Thomas Watson. Deterministic communication vs. partition number. *SIAM Journal on Computing*, 47(6):2435–2450, 2018. doi:10.1137/16M1059369.

21 Hao Huang. Induced subgraphs of hypercubes and a proof of the sensitivity conjecture. *Annals of Mathematics*, 190(3):949–955, 2019. doi:10.4007/annals.2019.190.3.6.

22 Troy Lee and Jérémie Roland. A strong direct product theorem for quantum query complexity. *Computational Complexity*, 22(2):429–462, 2013. doi:10.1109/CCC.2012.17.

23 Qian Li and Xiaoming Sun. On the sensitivity complexity of k-uniform hypergraph properties. In *STACS*, volume 66, pages 51:1–51:12, 2017. doi:10.4230/LIPIcs.STACS.2017.51.

24 Sagnik Mukhopadhyay and Swagato Sanyal. Towards better separation between deterministic and randomized query complexity. In *FSTTCS*, volume 45, pages 206–220, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.206.

25 Noam Nisan and Avi Wigderson. On rank vs. communication complexity. *Combinatorica*, 15(4):557–565, 1995. doi:10.1007/BF01192527.

26 David Rubinstein. Sensitivity vs. block sensitivity of Boolean functions. *Combinatorica*, 15(2):297–299, 1995. doi:10.1007/BF01200762.

27 Marc Snir. Lower bounds on probabilistic linear decision trees. *Theoretical Computer Science*, 38:69–82, 1985. doi:10.1016/0304-3975(85)90210-5.

28 Xiaoming Sun. Block sensitivity of weakly symmetric functions. *Theoretical Computer Science*, 384(1):87–91, 2007. doi:10.1016/j.tcs.2007.05.020.

29 Xiaoming Sun. An improved lower bound on the sensitivity complexity of graph properties. *Theoretical Computer Science*, 412(29):3524–3529, 2011. doi:10.1016/j.tcs.2011.02.042.

30 Xiaoming Sun, Andrew Chi-Chih Yao, and Shengyu Zhang. Graph properties and circular functions: How low can quantum query complexity go? In *CCC*, pages 286–293, 2004. doi:10.1109/CCC.2004.1313851.

31 György Turán. The critical complexity of graph properties. *Information Processing Letters*, 18(3):151–153, 1984. doi:10.1016/0020-0190(84)90019-X.