Discrete Mathematics and Theoretical Computer Science . . . . . .
. . . . . **vol. 27:2 #18 (2025)**

Separations between Combinatorial Measures for Transitive Functions[*]

Sourav Chakraborty[1†] Chandrima Kayal[2‡] Manaswi Paraashar[3§]

[1] *Indian Statistical Institute, Kolkata, India.*
[2] *Université Paris Cité, CNRS, IRIF, Paris, France.*
[3] *University of Copenhagen, Copenhagen, Denmark.*

revisions 30th Mar. 2023, 13th Oct. 2024, 22nd May 2025; accepted 7th June 2025.

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 sub-group G of S_n such that f is invariant under the action of G. In other words, the value of a transitive 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 investigates relations and separations between various complexity measures for the class of transitive functions. A class of functions called "pointer functions" is well known for demonstrating several state-of-the-art separations for general Boolean functions. The main contribution of this work is to extend this technique to transitive functions, constructing new functions that preserve the structural properties of pointer functions while being transitive. In particular, this allows us to show separations between query complexity and other measures for the class of transitive functions.

Our results advance the understanding of the transitivity of Boolean functions and highlight the utility of certain transitive groups, which may be of independent interest in mathematics and theoretical computer science.

A comprehensive summary of the relationships between combinatorial measures for transitive functions is presented, modeled after the known table for general Boolean functions.

Keywords: Boolean function, Transitive function, Partial Symmetry

[*] A preliminary version of this work appeared in ICALP, 2022 Chakraborty et al. (2022).

[†] https://orcid.org/0000-0001-9518-6204
[‡] https://orcid.org/0009-0006-4827-3640
[§] https://orcid.org/0009-0005-3805-5095

ISSN 1365–8050

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 ($\mathsf{RC}(f)$), unambiguous certificate complexity ($\mathsf{UC}(f)$), degree ($\deg(f)$), approximate degree ($\widetilde{\deg}(f)$) and spectral sensitivity ($\lambda(f)$)[(i)]? 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 Huang (2019). We refer the reader to the survey Buhrman and de Wolf (2002) 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 $\mathsf{D}(f) \leq \mathsf{R}(f)^3$.
- Separations - constructing functions that demonstrate 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 Aaronson et al. (2021).

For special classes of Boolean functions, the relationships and the separations might be different than those of general Boolean functions. For example, while it is known that there exists f such that $\mathsf{bs}(f) = \Theta(\mathsf{s}(f)^2)$ Rubinstein (1995), for a symmetric function a tighter result is known, $\mathsf{bs}(f) = \Theta(\mathsf{s}(f))$. The best-known relationship of $\mathsf{bs}(f)$ for general Boolean functions is $\mathsf{s}(f)^4$ Huang (2019). 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) have been studied extensively in the context of various complexity measures. 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 Ben-David et al. (2020) 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 [(ii)]. Another example of transitive functions is the graph

[(i)] We provide the formal definitions of the various measures used in this paper in Section 2.1

[(ii)] There are still open problems on the tightness of the constants.

properties. The input is the adjacency matrix, and the transitive group is the graph isomorphism group acting on the bits of the adjacency matrix. Turán (1984); Sun (2011); Li and Sun (2017); Gao et al. (2013) tried to obtain tight bounds on the relationship between sensitivity and block sensitivity for graph properties. They also tried to answer the following question: how low can sensitivity and block sensitivity go for graph properties?

In Sun et al. (2004); Chakraborty (2011); Sun (2007); Drucker (2011) 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, 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[(iii)], it is known (from Sun (2007) and Huang (2019)) that if f is a transitive function on n effective variables then its sensitivity $s(f)$ is at least $\Omega(n^{1/12})$[(iv)]. 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 $O(\log n)$. In Table 3 we have summarized the best-known separations of the combinatorial measures for transitive functions.

A naturally 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 constructions like the pointer function in Ambainis et al. (2017), 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 Aaronson et al. (2016) used for separation between quantum query complexity and degree, or approximate degree, are not transitive. Constructing transitive functions that 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 Aaronson et al. (2021) 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 Snir (1985) 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 Ambainis et al. (2017) a new function, $\mathsf{A1}$, 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 Ambainis et al. (2017) was a

[(iii)] A variable is effective if the function is dependent on it.

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

variant of the pointer functions - a class of functions introduced by Göös et al. (2018b) that has found extensive usage in showing separations between various complexity measures of Boolean functions. Note that this classes of functions are defined on non-Boolean domain, we can make such function Boolean using the canonical Boolean encoding which is of size at most log of the size of the alphabets we are using for non-Booelan domain. 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 1.1. *There exists a transitive function $F_{1.1}$ such that*

$$\mathsf{D}(F_{1.1}) = \widetilde{\Omega}(\mathsf{Q}(F_{1.1})^4), \qquad \mathsf{D}(F_{1.1}) = \widetilde{\Omega}(\mathsf{R_0}(F_{1.1})^2), \qquad \mathsf{D}(F_{1.1}) = \widetilde{\Omega}(\deg(F_{1.1})^2).$$

The proof of Theorem 1.1 is presented in Section 4. In Ambainis et al. (2017); Ben-David et al. (2017) 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. Using similar techniques as Theorem 1.1 one can construct transitive versions of other variants of pointer functions(from Ambainis et al. (2017); Ben-David et al. (2017)) which give matching separations to the best-known separations of general functions. The construction of these functions, though more complicated and involved, are similar in flavor to that of $F_{1.1}$.

Using standard techniques, we can also obtain the following theorems as corollaries to Theorem 1.1.

Theorem 1.2. *There exists transitive functions $F_{1.2}$ such that* $\mathsf{D}(F_{1.2}) = \widetilde{\Omega}(\mathsf{s}(F_{1.2})^3)$.

Our proof techniques also help us make transitive versions of other functions like that used in Aaronson et al. (2016) to demonstrate the gap between Q and certificate complexity.

Theorem 1.3. *There exists a transitive function $F_{1.3}$ such that* $\mathsf{Q}(F_{1.3}) = \widetilde{\Omega}(\mathsf{C}(F_{1.3})^2)$.

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 [(v)]. 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. We leave those directions for future work.

2 Basic definitions and notations

2.1 Complexity measures of Boolean functions

In this work, the relation between various complexity measures of Boolean functions is extensively studied. We refer the reader to the survey Buhrman and de Wolf (2002) for an introduction to the complexity of

[(v)] In Ben-David et al. (2020) 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	$\widetilde{\mathsf{deg}}$
D		2 ; 2 T:1.1	2 ; 3 T:1.1	2 ; 2 $\wedge \circ \vee$	2 ; 3 $\wedge \circ \vee$	2 ; 3 $\wedge \circ \vee$	3 ; 6 T:1.2	4 ; 6	2 ; 3	2 ; 3 T:1.1	4 ; 4 T:1.1	4 ; 4
R_0	1,1 $\oplus$		2 ; 2	2 ; 2 $\wedge \circ \vee$	2 ; 3 $\wedge \circ \vee$	2 ; 3 $\wedge \circ \vee$	3 ; 6	4 ; 6	2 ; 3	2 ; 3	3 ; 4	4 ; 4
R	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 2 $\wedge \circ \vee$	2 ; 3 $\wedge \circ \vee$	2 ; 3 $\wedge \circ \vee$	3 ; 6	4 ; 6	1.5 ; 3	2 ; 3	2 ; 4 $\wedge$	4 ; 4
C	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 2 $\oplus$		2 ; 2 Gilmer et al. (2016)	2 ; 2 Gilmer et al. (2016)	2 ; 5 Rubinstein (1995)	2 ; 6 $\wedge$	1.15 ; 3 Ambainis (2016)	1.63 ; 3 Nisan and Wigderson (1995)	2 ; 4 $\wedge$	2 ; 4 $\wedge$
RC	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		1.5 ; 2 Gilmer et al. (2016)	2 ; 4 Rubinstein (1995)	2 ; 4 $\wedge$	1.15 ; 2 Ambainis (2016)	1.63 ; 2 Nisan and Wigderson (1995)	2 ; 2 $\wedge$	2 ; 2 $\wedge$
bs	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$	1 ; 1 $\oplus$		2 ; 4 Rubinstein (1995)	2 ; 4 $\wedge$	1.15 ; 2 Ambainis (2016)	1.63 ; 2 Nisan and Wigderson (1995)	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 Ambainis (2016)	1.63 ; 2 Nisan and Wigderson (1995)	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:1.3	2 ; 6 T:1.3		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:1.3	2 ; 3 T:1.3	2 ; 3 T:1.3	2 ; 6 T:1.3	2 ; 6 T:1.3	1, 1 $\oplus$	1 ; 3 $\oplus$		1 ; 4 $\oplus$
$\widetilde{\mathsf{deg}}$	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)] Cells with a gray background are those for which *transitive functions* can be constructed in a similar but generalized fashion of our construction to demonstrate separations that match the best-known separations for general functions.

[(4)] 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 is 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.

[(5)] 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.

Boolean functions and complexity measures. Several additional complexity measures and their relations among each other can also be found in Ben-David et al. (2017) and Aaronson et al. (2021). Similar to the above references, we define several complexity measures of Boolean functions that are relevant to us.

We refer to Buhrman and de Wolf (2002) and Ambainis et al. (2017) for definitions of the deterministic query model, randomized query model, and quantum query model for Boolean functions.

Definition 2.1 (Deterministic query complexity)**.** *The deterministic query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{D}(f)$*, is the worst-case cost of the best deterministic query algorithm computing* f*.*

Definition 2.2 (Randomized query complexity)**.** *The randomized query complexity of* $f : \{0,1\}^n \to \{0,1\}$*, denoted by* $\mathsf{R}(f)$*, is the worst-case cost of the best-randomized query algorithm computing* f *to error at most* $1/3$*.*

Definition 2.3 (Zero-error randomized query complexity)**.** *The zero-error randomized query complexity*

of $f : \{0,1\}^n \to \{0,1\}$, denoted by $\mathsf{R}_0(f)$, is the minimum worst-case expected cost of a (Las Vegas) randomized query algorithm that computes f to zero-error, that is, on every input x, if the algorithm gives output then it should give the correct answer $f(x)$ with probability 1. Otherwise, it returns 'undefined'.

Definition 2.4 (Quantum query complexity)**.** *The quantum query complexity of $f : \{0,1\}^n \to \{0,1\}$, denoted by $\mathsf{Q}(f)$, is the worst-case cost of the best quantum query algorithm computing f to error at most $1/3$.*

Definition 2.5 (Exact quantum query complexity)**.** *The exact quantum query complexity of $f : \{0,1\}^n \to \{0,1\}$, denoted by $\mathsf{Q}_\mathsf{E}(f)$, is the minimum number of queries made by a quantum algorithm that outputs $f(x)$ on every input $x \in \{0,1\}^n$ with probability 1.*

Next, we define the notion of partial assignment that will be used to define several other complexity measures.

Definition 2.6 (Partial assignment)**.** *A partial assignment is a function $p : S \to \{0,1\}$ where $S \subseteq [n]$ and the size of p is $|S|$. For $x \in \{0,1\}^n$ we say $p \subseteq x$ if x is an extension of p, that is the restriction of x to S denoted $x|_S = p$.*

Definition 2.7 (Certificate complexity)**.** *A 1-certificate is a partial assignment that forces the value of the function to 1 and similarly, a 0-certificate is a partial assignment that forces the value of the function to 0. The certificate complexity of a function f on x, denoted as $\mathsf{C}(x, f)$, is the size of the smallest $f(x)$-certificate that can be extended to x.*

Also, define 0-certificate of f as $\mathsf{C}^0(f) = \max\{\mathsf{C}(f, x) : x \in \{0,1\}^n, f(x) = 0\}$ and 1-certificate of f as $\mathsf{C}^1(f) = \max\{\mathsf{C}(f, x) : x \in \{0,1\}^n, f(x) = 1\}$. Finally, define the certificate complexity of $f : \{0,1\}^n \to \{0,1\}$, denoted by $\mathsf{C}(f)$, to be $\max\{\mathsf{C}^0(f), \mathsf{C}^1(f)\}$.

Definition 2.8 (Unambiguous certificate complexity)**.** *For any Boolean function $f : \{0,1\}^n \to \{0,1\}$, a set of partial assignments U is said to form an unambiguous collection of 0-certificates for f if*

1. *Each partial assignment in U is a 0-certificate (with respect to f)*
2. *For each $x \in f^{-1}(0)$, there is some $p \in U$ with $p \subseteq x$*
3. *No two partial assignments in U are consistent.*

We then define $\mathsf{UC}_0(f)$ to be the minimum value of $\max_{p \in U} |p|$ over all choices of such collections U. We define $\mathsf{UC}_1(f)$ analogously, and set $\mathsf{UC}(f) = \max\{\mathsf{UC}_0(f), \mathsf{UC}_1(f)\}$. We also define the one-sided version, $\mathsf{UC}_{\min}(f) = \min\{\mathsf{UC}_0(f), \mathsf{UC}_1(f)\}$.

Next, we define randomized certificate complexity (see Aaronson (2008)).

Definition 2.9 (Randomized certificate complexity)**.** *A randomized verifier for input x is a randomized algorithm that, on input y in the domain of f accepts with probability 1 if $y = x$, and rejects with probability at least $\frac{1}{2}$ if $f(y) \neq f(x)$. If $y \neq x$ but $f(y) = f(x)$, the acceptance probability can be arbitrary. The Randomized certificate complexity of f on input x is denoted by $\mathsf{RC}(f, x)$, is the minimum expected number of queries used by a randomized verifier for x. The randomized certificate complexity of f, denoted by $\mathsf{RC}(f)$, is defined as $\max\{\mathsf{RC}(f, x) : x \in \{0,1\}^n\}$.*

Definition 2.10 (Block sensitivity)**.** *The block sensitivity* $\mathsf{bs}(f,x)$ *of a function* $f:\{0,1\}^n\to\{0,1\}$ *on an input* x *is the maximum number of disjoint subsets* $B_1,B_2,\ldots,B_r$ *of* $[n]$ *such that for all* j*,* $f(x)\neq f(x^{B_j})$*, where* $x^{B_j}\in\{0,1\}^n$ *is the input obtained by flipping the bits of* x *in the coordinates in* B_j*. The block sensitivity of* f*, denoted by* $\mathsf{bs}(f)$*, is* $\max\{\mathsf{bs}\{(f,x)\}:x\in\{0,1\}^n\}$*.*

Definition 2.11 (Sensitivity)**.** *The sensitivity of* f *on an input* x *is defined as the number of bits on which the function is sensitive:* $\mathsf{s}(f,x)=|\{i:f(x)\neq f(x^i)\}|$*. We define the sensitivity of* f *as* $\mathsf{s}(f)=\max\{\mathsf{s}(f,x):x\in\{0,1\}^n\}$

We also define 0*-sensitivity of* f *as* $\mathsf{s}^0(f)=\max\{\mathsf{s}(f,x):x\in\{0,1\}^n,f(x)=0\}$*, and* 1*-sensitivity of* f *as* $\mathsf{s}^1(f)=\max\{\mathsf{s}(f,x):x\in\{0,1\}^n,f(x)=1\}$*.*

Here we are defining *Spectral sensitivity* from Aaronson et al. (2021). For more details about *Spectral sensitivity* and its updated relationship with other complexity measures, we refer Aaronson et al. (2021).

Definition 2.12 (Spectral sensitivity)**.** *Let* $f:\{0,1\}^n\to\{0,1\}$ *be a Boolean function. The* sensitivity graph *of* f*,* $G_f=(V,E)$ *is a subgraph of the Boolean hypercube, where* $V=\{0,1\}^n$*, and* $E=\{(x,x\oplus e_i)\in V\times V:i\in[n],f(x)\neq f(x\oplus e_i)\}$*, where* $x\oplus e_i\in V$ *is obtained by flipping the* i*th bit of* x*. That is,* E *is the set of edges between neighbors on the hypercube that have different* f *values. Let* A_f *be the adjacency matrix of the graph* G_f*. We define the spectral sensitivity of* f *as the largest eigenvalue of* A_f*.*

Definition 2.13 (Degree)**.** *A polynomial* $p:\mathbb{R}^n\to\mathbb{R}$ *represents* $f:\{0,1\}^n\to\{0,1\}$ *if for all* $x\in\{0,1\}^n$*,* $p(x)=f(x)$*. The degree of a Boolean function* f*, denoted by* $\mathsf{deg}(f)$*, is the degree of the unique multilinear polynomial that represents* f*.*

Definition 2.14 (Approximate degree)**.** *A polynomial* $p:\mathbb{R}^n\to\mathbb{R}$ *approximately represents a function* $f:\{0,1\}^n\to\{0,1\}$ *if for all* $x\in\{0,1\}^n$*,* $|p(x)-f(x)|\leq\frac{1}{3}$*. The approximate degree of a Boolean function* f*, denoted by* $\widetilde{\mathsf{deg}}(f)$*, is the minimum degree of a polynomial that approximately represents* f*.*

The following is a known relation between degree and one-sided unambiguous certificate complexity measure (Ben-David et al. (2017)).

Observation 2.15 (Ben-David et al. (2017))**.** *For any Boolean function* f*,* $\mathsf{UC}_{\mathsf{min}}(f)\geq\mathsf{deg}(f)$*.*

Next, we define the composition of two Boolean functions.

Definition 2.16 (Composition of functions)**.** *Let* $f:\{0,1\}^n\to\{0,1\}$ *and* $g:\{0,1\}^m\to\{0,1\}^k$ *be two functions. Then* composition of f and g*, denoted by* $f\circ g:\{0,1\}^{\frac{n}{k}\cdot m}\to\{0,1\}$*, is defined to be a function on* $\frac{n}{k}m$ *bits such that on input* $x=(x_1,\ldots,x_n)\in\{0,1\}^{\frac{n}{k}\cdot m}$*, 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*. Also, we will assume throughout that* k *divides* n *whenever* $\frac{n}{k}$ *appears in the input size of a function.*

One often tries to understand how the complexity measure of a composed function behaves concerning the measures of the individual functions. The following folklore theorem that we will be using multiple times in our paper.

Theorem 2.17. *Let* $f:\{0,1\}^n\to\{0,1\}$ *and* $g:\{0,1\}^m\to\{0,1\}^k$ *be two Boolean functions and the composition* $(f\circ g)$ *is defined as Definition 2.16 then*

1. $\mathsf{D}(f\circ g)=\Omega(\mathsf{D}(f)/k)$*, assuming* g *is an onto function.*
2. $\mathsf{Q}(f\circ g)=\Omega(\mathsf{Q}(f)/k)$*, assuming* g *is onto.*

3. $\mathsf{R}_0(f \circ g) = O(\mathsf{R}_0(f) \cdot m)$ *and if* g *is onto then* $\mathsf{R}_0(f \circ g) = \Omega(\mathsf{R}_0(f)/k)$
4. $\mathsf{R}(f \circ g) = O(\mathsf{R}(f) \cdot m)$ *and if* g *is onto then* $\mathsf{R}(f \circ g) = \Omega(\mathsf{R}(f)/k)$.
5. $\deg(f \circ g) = O(\deg(f) \cdot m)$
6. $\widetilde{\deg}(f \circ g) = O(\widetilde{\deg}(f) \cdot m)$.

Proof: Here we are giving a proof for (1). The proof of other lower bounds follow from a similar argument, while the upper-bounds are straight forward.

The input to f in $f \circ g$ can be seen as an n-bit string divided into $m = n/k$ blocks each of size k. Since g is onto, all possible strings in $\{0,1\}^n$ as possible inputs to f. In order to compute f correctly on all inputs, at least $\mathsf{D}(f)$ may inputs to f bits must be queried. Since one query to $f \circ g$ reveals at most k bits of the input to f, we get a lower bound of $\Omega(\mathsf{D}(f)/k)$.

□

If the inner function g is Boolean valued then we can obtain some tighter results for the composed functions.

Theorem 2.18. *Let* $f : \{0,1\}^n \to \{0,1\}$ *and* $g : \{0,1\}^m \to \{0,1\}$ *be two Boolean functions then*

1. *(Tal (2013); Montanaro (2014))* $\mathsf{D}(f \circ g) = \Theta(\mathsf{D}(f) \cdot \mathsf{D}(g))$.
2. *(Reichardt (2011); Lee et al. (2011); Kimmel (2013))* $\mathsf{Q}(f \circ g) = \Theta(\mathsf{Q}(f) \cdot \mathsf{Q}(g))$.
3. *(folklore)* $\deg(f \circ g) = \Theta(\deg(f) \cdot \deg(g))$.

Over the years a number of interesting Boolean functions have been constructed to demonstrate differences between various measures of Boolean functions. Some of the functions have been referred to in the Table 1. We describe the various functions in the Subsection 2.2.

2.2 Some Boolean functions and their properties

In this section, we define some standard functions that are either mentioned in Table 1 or used somewhere in the paper. We also state some of the properties that we need for our proofs. We start by defining some basic Boolean functions.

Definition 2.19. *Define* $\mathsf{PARITY} : \{0,1\}^n \to \{0,1\}$ *to be the* $\mathsf{PARITY}(x_1, \ldots, x_n) = \sum x_i mod\ 2$. *We use the notation* $\oplus$ *to denote* PARITY.

Definition 2.20. *Define* $\mathsf{AND} : \{0,1\}^n \to \{0,1\}$ *to be the* $\mathsf{AND}(x_1, \ldots, x_n) = 0$ *if and only if there exists an* $i \in [n]$ *such that* $x_i = 0$. *We use the notation* $\wedge$ *to denote* AND.

Definition 2.21. *Define* $\mathsf{OR} : \{0,1\}^n \to \{0,1\}$ *to be the* $\mathsf{OR}(x_1, \ldots, x_n) = 1$ *if and only if there exists an* $i \in [n]$ *such that* $x_i = 1$. *We use the notation* $\vee$ *to denote* OR.

Definition 2.22. *Define* $\mathsf{MAJORITY} : \{0,1\}^n \to \{0,1\}$ *as* $\mathsf{MAJORITY}(x) = 1$ *if and only if* $|x| > \frac{n}{2}$.

We need the following definition of composing iteratively with itself.

Definition 2.23 (Iterative composition of a function)**.** *Let $f : \{0,1\}^n \to \{0,1\}$ a Boolean function. For $d \in \mathbb{N}$ we define the function $f^d : \{0,1\}^{n^d} \to \{0,1\}$ as follows: if $d = 1$ then $f^d = f$, otherwise*

$$f^d(x_1, \ldots, x_{n^d}) = f\left(f^{d-1}(x_1, \ldots, x_{n^{d-1}}), \ldots, f^{d-1}(x_{n^d-n+1}, \ldots, x_{n^d})\right).$$

Definition 2.24. *For $d \in \mathbb{N}$ define* NAND*-tree of depth d as* NAND^d *where* $\mathsf{NAND} : \{0,1\}^2 \to \{0,1\}$ *is defined as:* $\mathsf{NAND}(x_1, x_2) = 0$ *if and only if $x_1 \neq x_2$. We use the notation $\widetilde{\wedge}$-tree to denote* NAND*-tree.*

Now we will define a function that gives a quadratic separation between sensitivity and block sensitivity.

Definition 2.25 (Rubinstein's function (Rubinstein (1995)))**.** *Let $g : \{0,1\}^k \to \{0,1\}$ be such that $g(x) = 1$ iff x contains two consecutive ones and the rest of the bits are 0. The Rubinstein's function, denoted by* $\mathsf{RUB} : \{0,1\}^{k^2} \to \{0,1\}$ *is defined to be* $\mathsf{RUB} = \mathsf{OR}_k \circ g$.

Theorem 2.26. *(Rubinstein (1995)) For the Rubinstein's function in Definition 2.25* $\mathsf{s}(\mathsf{RUB}) = k$ *and* $\mathsf{bs}(\mathsf{RUB}) = k^2/2$. *Thus* RUB *witnesses a quadratic gap between sensitivity and block sensitivity.*

Nisan and Szegedy (1994) first introduced a function whose $\deg$ is significantly smaller than s or bs. This appears in the footnote in Nisan and Wigderson (1995) that E. Kushilevitz also introduced a similar function with 6 variables which gives a slightly better gap between s and $\deg$. Later Ambainis computed $\mathsf{Q_E}$ of that function and gave a separation between $\mathsf{Q_E}$ and D Ambainis (2016). This function is fully sensitive at all zero input, consequently, this gives a separation between $\mathsf{Q_E}$ and s.

Definition 2.27 (Nisan and Szegedy (1994))**.** *Define* NW *as follows:*

$$\mathsf{NW}(x_1, x_2, x_3) = \begin{cases} 1 \text{ iff } x_i \neq x_j \text{ for some } i, j \in \{1,2,3\} \\ 0 \text{ otherwise}. \end{cases}$$

Now define the d-th iteration NW^d *on $(x_1, x_2, \ldots, x_{3^d})$ as Definition 2.23 where $d \in \mathbb{N}$.*

Definition 2.28 (Kushilevitz's function)**.** *Define* K *as follows:*

$$\mathsf{K}(x_1, x_2, x_3, x_4, x_5, x_6) = \Sigma x_i + \Sigma x_i y_i + x_1x_3x_4 + x_1x_4x_5 + x_1x_2x_5 + x_2x_3x_4 + x_2x_3x_5 + x_1x_2x_6 \\ + x_1x_3x_6 + x_2x_4x_6 + x_3x_5x_6 + x_4x_5x_6.$$

Now define the d-th iteration K^d *on $(x_1, x_2, \ldots, x_{6^d})$ as Definition 2.23 where $d \in \mathbb{N}$.*

Next, we will describe two examples that were introduced in Gilmer et al. (2016) and give the separation between C vs. RC and RC vs. bs respectively. They also have introduced some new complexity measures for the iterative version of a function and how to use them to get the critical measure between two complexity measures. For more details we refer to Gilmer et al. (2016).

Definition 2.29 (Gilmer et al. (2016))**.** *Let n be an even perfect square, let $k = 2\sqrt{n}$ and $d = \sqrt{n}$. Divide the n indices of the input into n/k disjoint blocks. Define* $\mathsf{GSS}_1 : \{0,1\}^n \to \{0,1\}$ *as follows:* $\mathsf{GSS}_1(x) = 1$ *if and only if $|x| \geq d$ and all the 1's in x are in a single block. Define* GSS_1^t *with* $\mathsf{GSS}_1^1 = \mathsf{GSS}_1$ *and* $\mathsf{GSS}_1^t = \mathsf{GSS}_1^{t-1} \circ \mathsf{GSS}_1^1$.

Definition 2.30 (Gilmer et al. (2016))**.** *Define* $\mathsf{GSS}_2 : \{0,1\}^n \to \{0,1\}$ *where n is of the form $\binom{t}{2}$ for some integer t. Identify the input bits of* GSS_2 *with the edges of the complete graph K_t. An input $x \in \{0,1\}^n$ induces a subgraph of K_t consisting of edges assigned 1 by x. The function* $\mathsf{GSS}_2(x)$ *is defined to be 1 iff the subgraph induces by x has a star graph.*

Definition 2.31. *For $\Sigma = [n^k]$ the function* k-sum $: \Sigma^n \to \{0,1\}$ *is defined as follows: on input $x_1, x_2, \ldots, x_n \in \Sigma$, if there exists k element $x_{i_1}, \ldots, x_{i_k}$, $i_1, \ldots, i_k \in [n]$, that sums to 0 (mod $|\Sigma|$) then output 1, otherwise output 0.*

Theorem 2.32 (Aaronson et al. (2016); Belovs and Spalek (2013))**.** *For the function* k-sum $: \Sigma^n \to \{0,1\}$, *if $|\Sigma| \geq 2\binom{n}{n}$ then*

$$\mathsf{Q}(\text{k-sum}) = \Omega(n^{k/(k+1)}/\sqrt{k}).$$

Next, we will define a refined version of k-sum function that was defined in Aaronson et al. (2016).

Definition 2.33. *Define* Block-k-Sum $: \{0,1\}^n \to \{0,1\}$ *as follows: split the input into blocks of size $10k \log n$ each and call a block balanced if it has an equal number of 0s and 1s. Let the balanced blocks represent numbers in an alphabet Σ of size $\Omega(n^k)$. The function* Block-k-Sum *evaluates to 1 if and only if there are k balanced blocks whose corresponding numbers sum to $0(\mod \Sigma)$ and all other blocks have at least as many 1s as 0s.*

Next, we define the cheat sheet version of a function from Aaronson et al. (2016).

Definition 2.34. *We define the cheat sheet version of f as follows: the input to f_{CS} consist of $\log n$ inputs to f , each of size n , followed by n blocks of bits of size $C(f) \times \log n$ each. Let us denote the input to f_{CS} as $X = (x_1, x_2, \ldots, x_{\log n}, Y_1, Y_2, ..., Y_n)$, where x_i is an input to f , and the Y_i are the aforementioned cells of size $C(f) \times \log n$. The first part $x_1, x_2, \ldots, x_{\log n}$ of the string, we call the input section, and the rest of the part we call as certificate section of the whole input. Define $f_{CS} : \{0,1\}^{n \times \log n + n \times (C(f) \log n)} \to \{0,1\}$ to be 1 if and only if the following conditions hold:*

- *For all i, x_i is in the domain of f. If this condition is satisfied, let l be the positive integer corresponding to the binary string $(f(x_1), f(x_2), ..., f(x_{\log n}))$.*
- *Y_l certifies that all x_i are in the domain of f and that l equals the binary string formed by their output values, $(f(x_1), f(x_2), ..., f(x_{\log n}))$.*

Finally, we present the pointer functions and their variants introduced in Ambainis et al. (2017). They are used to demonstrate the separation between several complexity measures like *deterministic query complexity*, *Randomized query complexity*, *Quantum query complexity* etc. These functions were originally motivated from Göös et al. (2018b) function. The functions that we construct for many of our theorems a composition functions whose outer function is these pointer functions, or a slight variant of these. In the next section, we present the formal definition of pointer functions.

2.2.1 Pointer function

For the sake of completeness first, we will describe the “pointer function" introduced in Ambainis et al. (2017) that achieves separation between several complexity measures like *Deterministic query complexity*, *Randomized query complexity*, *Quantum query complexity* etc. This function was originally motivated by a function in Göös et al. (2018b). There are 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 match the known separation between the general functions. Here we will define only the first variant of the pointer function.

Definition 2.35 (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[vi] 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.*

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

Definition 2.36 (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 2.37. *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 completely balanced tree with 2^k leaves, and add a pair of child nodes to each $n - 2^k$ leaves starting from the left. Define $T(j)$ as before.

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

Definition 2.38 (Variant 1, Ambainis et al. (2017))**.** *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*

[vi] In Ambainis et al. (2017) other variations of pointer functions were defined and have been used to give various separation results, we also give transitive pointer functions for such separations. A detailed proof can be found in the previous version of the paper Chakraborty et al. (2021).

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 to l_j as the leaves of the tree.

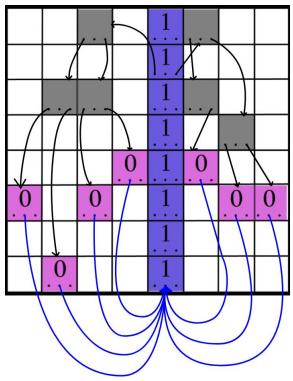

Figure 1: Example of 1-instance of A1 function on 8×8 grid

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 Ambainis et al. (2017) which we will use to prove the results for our function:

Theorem 2.39 (Ambainis et al. (2017))**.** *The function $\mathsf{A1}_{(m.n)}$ in Definition 2.38 satisfies*

$$\mathsf{D} = \Omega(n^2) \text{ for } m = 2n \text{ where } m, n \in \mathbb{N},$$
$$\mathsf{R}_0 = \widetilde{O}(m+n) \text{ for any } m, n \in \mathbb{N},$$
$$\mathsf{Q} = \widetilde{O}(\sqrt{m} + \sqrt{n}) \text{ for any } m, n \in \mathbb{N}.$$

Though Ambainis et al. (2017) gives the deterministic lower bound for the function 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 the sake of completeness, we give a proof for $n \times n$ matrices.

Theorem 2.40. $\mathsf{D}(\mathsf{A1}_{(n,n)}) = \Omega(n^2)$.

Adversary Strategy for $\mathsf{A1}_{(n,n)}$: We describe an adversary strategy that ensures that the value of the function is undetermined after $\Omega(n^2)$ queries. Assume that a deterministic query algorithm queries a cell (i, j). Let k be the number of queried cells in the column j. If $k \leq \frac{n}{2}$ adversary will return $(1, \perp, \perp, \perp)$. Otherwise adversary will return $(0, \perp, \perp, n-k)$.

Claim 2.41. *The value of the function $\mathsf{A1}_{(n,n)}$ will be undetermined if there is a column with at most $n/2$ queried cells in the first $\frac{n}{2}$ columns $\{1, 2, \ldots, \frac{n}{2}\}$ and at least $3n$ unqueried cells in total.*

Proof: The adversary can always set the value of the function to 0 if the conditions of the claim are satisfied.

Adversary can also set the value of the function to 1**:** If $s \in [\frac{n}{2}]$ be the column with at most $\frac{n}{2}$ queried cell, then all the queried cells of the column are of the form $(1, \perp, \perp, \perp)$. Assign $(1, \perp, \perp, \perp)$ to the other cell and leave one cell for the *special element* $a_{p,s}$(say).

For each non-marked column $j \in [n] \setminus \{s\}$ define l_j as follows: If column j has one unqueried cell then assign $(0, \perp, \perp, s)$ to that cell. If all the cells of the column j were already queried then the column contains a cell with $(0, \perp, \perp, s)$ by the adversary strategy. So, in either case, we can form a *leave* l_j in each of the non-marked columns.

Now using the cell of *special element* $a_{p,s}$ construct a rooted tree of pointers isomorphic to tree T as defined in Definition 2.38 such that the internal nodes we will use the other unqueried cells and assign pointers such that $l(j)$'s are the leaves of the tree and the *special element* $a_{p,s}$ is the $root$ of the tree. Finally, assign anything to the other cell. Now the function will evaluate to 1.

To carry out this construction we need at most $3n$ number of unqueried cells. Outside of the $marked\ column$ total $n-2$ cells for the internal nodes of the tree, at most $n-1$ unqueried cell for the $leaves$ and the *all* 1 *unique marked column* contains total n cell, so total $3n$ unqueried cell will be sufficient for our purpose.

Now there are a total n number of columns and to ensure that each of the columns in $\{1, 2, \ldots, \frac{n}{2}\}$ contains at least $\frac{n}{2}$ queried cell we need at least $\frac{n^2}{4}$ number of queries. Since $n^2 - 3n \geq \frac{n^2}{4}$ for all $n \geq 6$. Hence $D(\mathsf{A1}_{(n,n)}) = \Omega(n^2)$. □

Hence Theorem 2.40 follows.

Also Göös et al. (2018b)'s function realizes quadratic separation between D and $\deg$ and the proof goes via UC_{min} upper bound. The function $\mathsf{A1}_{(n,n)}$ 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 2.42. $\mathsf{UC}_{min}(\mathsf{A1}_{(n,n)}) = O(n)$ *which implies* $\deg(\mathsf{A1}_{(n,n)})$ *is also* $O(n)$ *for any* $n \in \mathbb{N}$.

Another important observation that we need is the following:

Observation 2.43 (Ambainis et al. (2017))**.** *For any input* $\Sigma^{n\times n}$ *to the function* $\mathsf{A1}_{(n,n)}$ *(in Definition 2.38) 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.*

2.3 Some useful notations

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 2.44. *For any* $n = 2k$ *the* flip *swaps* $(2i-1)$ *and* $2i$ *for all* $1 \leq i \leq k$*. The permutation* $\mathsf{Swap}_{\frac{1}{2}}$ *swaps* i *with* $(k+i)$*, for all* $1 \leq i \leq k$*. 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 the same for all $\ell \in [n]$. The following representation of $\ell \in [n]$ ensures that for all $\ell \in [n]$ the encoding has the same Hamming weight.

Definition 2.45 (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 the 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 2.46 (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 assignment 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 are 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 2.47. *If $A \leq \mathsf{S}_n$ and $B \leq \mathsf{S}_m$ are groups on $[n]$ and $[m]$ then the group $A \times B$ acts 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.*

2.4 Transitive groups and transitive functions

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

Definition 2.48. *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 2.49. *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 $\alpha_1, \ldots, \alpha_n \in A$.*

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

The following observation proves that the composition of transitive functions is also a transitive function.

Observation 2.50. *Let $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^m \to \{0,1\}$ be transitive functions. Then $f \circ g : \{0,1\}^{nm} \to \{0,1\}$ is also transitive.*

Proof: Let $T_f \subseteq \mathsf{S}_n$ and $T_g \subseteq \mathsf{S}_m$ be the transitive groups corresponding to f and g, respectively. On input $x = (X_1, \ldots, X_n)$, $X_i \in \{0,1\}^m$ for $i \in [n]$, the function $f \circ g$ is invariant under the action of the group $T_f \wr T_g$ - the wreath product of the T_f with T_g. The group $T_f \wr T_g$ acts on the input string through the following permutations:

1. any permutation $\pi \in T_f$ acting on indices $\{1, \ldots, n\}$ or

2. any permutations $(\sigma_1, \ldots, \sigma_n) \in (T_g)^n$ acting on $X_1, \ldots, X_n$ i.e. $(\sigma_1, \ldots, \sigma_n)$ sends $X_1, \ldots, X_n$ to $\sigma_1(X_1), \ldots, \sigma_n(X_n)$.

□

Observation 2.51. *If $A \leq \mathsf{S}_n$ and $B \leq \mathsf{S}_m$ are transitive 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 2.52. *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.*

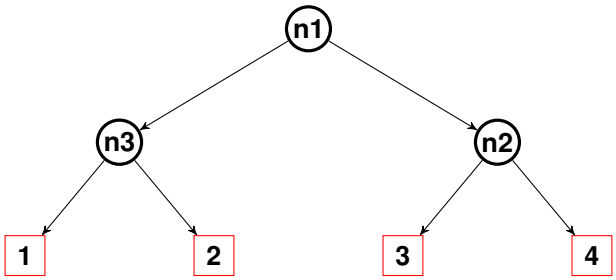

Figure 2: Induced group actions for Bt_4 group

Claim 2.53. *The group Bt_k is a transitive group.*

Proof: For any $i, j \in [k]$, we have to show that there exists a permutation $\pi \in \mathsf{Bt}_k$ such that $\pi(i) = j$. Let us form a complete binary tree of height $\log k$ in the following way:

- (Base case:) Start from the root node, and label the left and right child as 0 and 1 respectively.
- For every node x, label the left and right child as $x0$ and $x1$ respectively.

Note that our complete binary tree has k leaves, where each of the leaves is labeled by a binary string of the form $x_1 x_2 \ldots x_{\log k}$, which is the binary representation of numbers in $[k]$. Similarly, any node in the tree can be labeled by a binary string $x_1 x_2 \ldots x_t$, where $0 \leq t \leq \log k$ and t is the distance of the node from the root.

Now for any $i, j \in [k]$, let the binary representation of i be $(x_1 x_2 \dots x_{\log k})$ and that of j be $(y_1 y_2 \dots y_{\log k})$. Now we will construct the permutation $\pi \in \mathsf{Bt}_k$ such that $\pi(i) = j$. Without loss of generality, we can assume $i \neq j$.

Find the least positive integer $\ell \in [\log k]$ such that $x_\ell \neq y_\ell$, then go to the node labeled $x_1 x_2 \dots x_{\ell-1}$ and swap it's left and right child. Let $\pi_{x_1 \dots x_{\ell-1}} \in S_k$ be the corresponding permutation of the leaves of the tree, in other words on the set $[k]$. Note that, by definition, the permutation $\pi_{x_1 \dots x_{\ell-1}} \in S_k$ is in Bt_k. Also note that the permutation $\pi_{x_1 \dots x_{\ell-1}}$ acts of the set $[k]$ as follows:

- $\pi_{x_1 \dots x_{\ell-1}}(z_1 \dots z_{\log k}) = z_1 \dots z_{\log k}$ if $z_1 \dots z_{\ell-1} \neq x_1 \dots x_{\ell-1}$
- $\pi_{x_1 \dots x_{\ell-1}}(x_1 \dots x_{\ell-1} 0 z_{\ell+1} \dots z_{\log k}) = (x_1 \dots x_{\ell-1} 1 z_{\ell+1} \dots z_{\log k})$
- $\pi_{x_1 \dots x_{\ell-1}}(x_1 \dots x_{\ell-1} 1 z_{\ell+1} \dots z_{\log k}) = (x_1 \dots x_{\ell-1} 0 z_{\ell+1} \dots z_{\log k})$

Since $i = x_1 \dots x_{\ell-1} x_\ell \dots x_{\log k}$ and $j = y_1 \dots y_{\ell-1} y_\ell \dots y_{\log k}$ with $x_1 \dots x_{\ell-1} = y_1 \dots y_{\ell-1}$ and $x_\ell \neq y_\ell$, so

$$\pi_{x_1 \dots x_{\ell-1}}(i) = y_1 \dots y_{\ell-1} y_\ell x_{\ell+1} \dots y_{\log k}$$

So the binary representation of $\pi_{x_1 \dots x_{\ell-1}}(i)$ and j matching in the first ℓ positions which is one more than the number of positions where the binary representation of i and j matched. By doing this trick repeatedly, that is by applying different permutations from Bt_k one after another we can map i to j. □

Note 2.54. *Note that the group* Bt_k *is the same as an iterated wreath product* $\mathsf{S}_2 \wr \cdots \wr \mathsf{S}_2$ *of depth* $\log k$.

The following claim describes how the group Bt_k acts on various encodings of integers. Recall the balance-binary representation (Definition 2.45).

Claim 2.55. *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* $\gamma(i) = j$ *where* $2\log n$ *is a power of* 2.

Proof: Recall the group $\mathsf{Bt}_{2\log n}$: assuming that the elements of $[2\log n]$ are placed on the leaves of the binary tree of depth $\log(2\log n)$, the group $\mathsf{Bt}_{2\log n}$ is generated by the permutations of the form "pick a node in the binary tree of and swap the left and right sub-tree of the node". So it is enough to prove that for any elementary permutation $\widehat{\gamma}$ of the form "pick a node in the binary tree and swap the left and right sub-tree of the node" there is a $\gamma \in \mathsf{S}_n$ such that for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

Any node in the binary tree of depth $\log(2\log n)$ can be labeled by a $0/1$-string of length t, where $0 \leq t \leq \log(2\log n)$ is the distance of the node from the root. We split the proof of the claim into two cases depending on the value of the t - the distance from the root.

If $t = \log(2\log n)$**:** This is the case when the node is at the last level - just above the leaf level. Let the node be u and let s be the number whose binary representation is the label of the node u. Let the numbers in the leaves of the tree correspond to the $bb(i)$ - the balanced binary representation of $i \in [n]$. Note that because of the balanced binary representation, the children of u are

- 0 (left-child) and 1 (right-child) if the s-th bit in the binary representation of i is 0
- 1 (left-child) and 0 (right-child) if the s-th bit in the binary representation of i is 1

So the permutation (corresponding to swapping the left and right sub-trees of u) only changes the order of 0 and 1 - which corresponds to flipping the s-th bit of the binary representation of i. And so in this case the γ acting on the set $[n]$ is just collection transpositions swapping i and j iff the the binary representation of i and j are the same except for the s-th bit.

So in this case for all $i, j \in [n]$, $\widehat{\gamma}[bb(i)] = bb(j)$ iff $\gamma(i) = j$.

If $t < \log(2 \log n)$: Let the node be v. Note that in this case since the node keeps the order of the $2r - 1$ and $2r$ bits unchanged (for any $1 \leq r \leq \log n$), it is enough we can visualize the action by an action of swapping the left and right sub-trees of the node v on the binary representation of i (instead of the balance binary representation of i). And so we can see that the action of the permutation (corresponding to swapping the left and right sub-trees of v) automatically gives a permutation of the binary representations of numbers between 1 and n, as was discussed in the proof of Claim 2.53. And hence we have 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 be using for the set of rows and columns of a matrix.

Definition 2.56. *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:*

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

Note that Claim 2.57 directly follows from Claim 2.55.

Claim 2.57. *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 $(r_i, c_j) \in (R \times \{0\}) \cup (\{0\} \times C)$

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

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 [(vii)]. For more details, refer to Appendix 2.2.1. As described in Göös et al. (2018b), the high-level idea of pointer functions is the usage of pointers to make certificates unambiguous without increasing the input size significantly. This technique turns out to be very useful in giving separations between various complexity measures as we see in Mukhopadhyay and Sanyal (2015), Göös et al. (2018a), and Ambainis et al. (2017). Note that *Pointer function* contains non-Boolean inputs alphabets. Consequently, our transitive function also involves non-Boolean input alphabets.

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

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 [(viii)]. 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 main 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 change, the cells get 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 2.45) 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 an "entangled" manner. For this, we induce a group action 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 2.52,

[(viii)] 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.

Claim 2.55). This helps us to permute the rows (or columns) using a permutation while updating the encoding of the pointers accordingly.

By Claim 2.55, 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 2.56 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 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 2.38) transitive where there is the same number of rows and columns. The further variations need a more delicate handling of the encoding and the transitive groups - though the central idea is similar.

4 Separations between deterministic query complexity and some other complexity measures

4.1 Transitive pointer function $F_{1.1}$ for Theorem 1.1

Our function $F_{1.1} : \Gamma^{n\times n} \to \{0,1\}$ 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 Ambainis et al. (2017) (see Definition 2.38 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

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

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 2.39.

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

$$F_{1.1} := \big(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\big) : \Gamma^{n\times n} \to \{0,1\}.$$

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 $(\text{Value}, \text{LPointer}, \text{RPointer}, \text{BPointer})$ 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 Ambainis et al. (2017)) that the elements of Σ is encoded as a k-length[(ix)] binary string in a canonical way.

The main insight for our function $F_{1.1} := \left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right)$ 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[(x)] 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 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 cleverly. 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 $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \vdash$. Then we will extend it to describe the standard for the encoding of $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ where $T = \dashv$. Finally, we will explain all other valid encodings of a tuple $(V, (r_L, c_L), (r_R, c_R), (c_B), T)$ 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 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 a 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

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

(x) 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.

...	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 $\lvert\ell_2\rvert = 2\log n - 1 - V$	$4\log n$	$2\log n + 1$	$2\log n + 2$	$12\log n + 2 - V$
$P2$	$\mathcal{E}(r_L, 0)$	$2\log n + 3$	$2\log n + 1$	$2\log n + 2$	$7\log n + 6$
$P3$	$\mathcal{E}(0, c_L)$	$2\log n + 4$	$2\log n + 1$	$2\log n + 2$	$7\log n + 7$
$P4$	$\mathcal{E}(r_R, 0)$	$2\log n + 5$	$2\log n + 1$	$2\log n + 2$	$7\log n + 8$
$P5$	$\mathcal{E}(0, c_R)$	$2\log n + 6$	$2\log n + 1$	$2\log n + 2$	$7\log n + 9$
$P6$	$\mathcal{E}(0, c_B)$	$2\log n + 7$	$2\log n + 1$	$2\log n + 2$	$7\log n + 10$

Table 2: * An integer entry k in the table indicates it contains a Boolean string of length $4\log n$ with Hamming weight k. The 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.

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 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 emphasize 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 2.56). For part P_i (with $2 \leq i \leq 6$) 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 blocks 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 is classified 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 parts, the 4 blocks (say B_1, B_2, B_3, B_4) can be permuted in 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 $\mathsf{Dec} : \{0,1\}^{96\log n} \to \bar{\Sigma}$:

- 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 2.45).
- 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 $F_{1.1}$ 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 2.52) 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 $\mathcal{T}$ will 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 2.53.

Observation 4.1. *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 string 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 the $\mathcal{E}$-encoding of the column $(0, C_j)$ uses the balanced binary representation of j in the second half and all zero sting in the first half.*

Using Observation 4.1 we have the following claim.

Claim 4.2. *The group $\mathcal{T}$ acting on the cells 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 $\widehat{\sigma}[\mathcal{E}(0, j_1)] = (0, j_2)$. 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 4.2 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. The input to the function $F_{1.1}$ 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 all the bits of the input to the function $F_{1.1}$.

Recall that the input to the function $F_{1.1}$ 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)[xi]: $(\mathsf{flip}(B_1), \mathsf{flip}(B_2), B_4, B_3)$
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 cell untouched - it just changes the location of the cells.
   (b) (Pointer-update) For each of the 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 applies $\widehat{\sigma} \in \widehat{\mathcal{T}}$ corresponding to σ.

We now have the following theorems that would prove that the function $F_{1.1}$ is transitive.

Theorem 4.3. *G is a transitive group and the function $F_{1.1}$ is invariant under the action of the G.*

Proof of Theorem 4.3:

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 is broken into 4 blocks 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 group, 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 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 change 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 this might have once again changed the position of the index p in the block. Note that, even if the position in

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

the block changes it must be a flip operation away. In other words, since at 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 of the position in the block.

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. Depending on whether the Block Flip operation is applied the position in the block can again change. Note that, 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. This will not change the cell position, Part Number, or 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 function $F_{1.1}$ 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 the description of the decoding algorithm Dec in Section 4.1.1. So clearly the function $F_{1.1}$ 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 the description of the decoding algorithm Dec in Section 4.1.1. Thus $F_{1.1}$ is also invariant under any block permutation.

Case 3: π is a Cell-permutation From Observation 2.43 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 2.57, 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 4.4. *Deterministic query complexity of $F_{1.1}$ is $\Omega(n^2)$.*

Proof: The function $\mathsf{ModA1}_{(n,n)}$ is a "harder" function than $\mathsf{A1}_{(n,n)}$. Here by "harder" we mean $\mathsf{A1}_{(n,n)}$ is a sub-function of $\mathsf{ModA1}_{(n,n)}$. $\mathsf{ModA1}_{(n,n)}$ contains extra alphabets compared to $\mathsf{A1}_{(n,n)}$, upon restriction on those extra alphabets we can get back the function $\mathsf{A1}_{(n,n)}$, so $\mathsf{D}(\mathsf{ModA1}_{(n,n)})$ is at least that of $\mathsf{D}(\mathsf{A1}_{(n,n)})$. Now since, $F_{1.1}$ is $\left(\mathsf{ModA1}_{(n,n)} \circ \mathsf{Dec}\right)$ so clearly the $\mathsf{D}(F_{1.1})$ is at least $\mathsf{D}(\mathsf{A1}_{(n,n)})$. Theorem 2.40 proves that $\mathsf{D}(\mathsf{A1}_{(n,n)})$ is $\Omega(n^2)$. Hence $\mathsf{D}(F_{1.1}) = \Omega(n^2)$. □

The following Claim 4.5 follows from the definition of the function $\mathsf{ModA1}_{(n,n)}$.

Claim 4.5. *The following are some properties of the function* $\mathsf{ModA1}_{(n,n)}$

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

Finally, from Theorem 2.17 we see that the $\mathsf{R_0}(F_{1.1})$, $\mathsf{Q}(F_{1.1})$ and $\deg(F_{1.1})$ 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 4.4, Claim 4.5 and Theorem 2.39 (from Ambainis et al. (2017)) we have Theorem 1.1.

5 Separation between sensitivity and randomized query complexity

Ben-David et al. (2017) showed that functions that witness a gap between deterministic query complexity (or randomized query complexity), and $\mathsf{UC_{min}}$ can be transformed to give functions that witness separation between deterministic query complexity (or randomized query complexity) and sensitivity. We observe that transformation the Ben-David et al. (2017) described preserves transitivity. Our transitive functions from Theorem 1.1 along with the transformation from Ben-David et al. (2017) gives the cubic separations between R and s

We start by defining *desensitization transform* of Boolean functions as defined in Ben-David et al. (2017).

Definition 5.1 (Desensitized Transformation)**.** *Let $f : \{0,1\}^n \to \{0,1\}$. Let U be a collection of unambiguous 1-certificates for f, each of size at most $\mathsf{UC}_1(f)$. For each $x \in f^{-1}(1)$, let $p_x \in U$ be the unique certificate in U consistent with x. The desensitized version of f is the function $f_{\mathsf{DT}} : \{0,1\}^{3n} \to \{0,1\}$ defined by $f_{\mathsf{DT}}(x_1x_2x_3) = 1$ if and only if $f(x_1) = f(x_2) = f(x_3) = 1$ and $p_{x_1} = p_{x_2} = p_{x_3}$.*

Observation 5.2. *If $f : \{0,1\}^n \to \{0,1\}$ is transitive, then $f_{\mathsf{DT}} : \{0,1\}^{3n} \to \{0,1\}$ is also transitive.*

Proof: Let $T_f \subseteq \mathsf{S}_n$ be the transitive group corresponding to f and let $x_1x_2x_3 \in \{0,1\}^{3n}$ be the input to f_{DT}. Consider the following permutations acting on the input $x_1x_2x_3$ to f_{DT}:

1. S_3 acting on the indices $\{1,2,3\}$ and
2. $\{(\sigma,\sigma,\sigma) \in \mathsf{S}_{3n} | \sigma \in T_f\}$ acting on (x_1, x_2, x_3).

Observe that the above permutations act transitively on the inputs to f_{DT}. Also from the definition of f_{DT} the value of the function f_{DT} is invariant under these permutations. □

Next, we need the following theorem from Ben-David et al. (2017). The theorem is true for more general complexity measures. We refer the reader to Ben-David et al. (2017) for a more general statement.

Theorem 5.3 (Ben-David et al. (2017))**.** *For any $k \in \mathbb{R}^+$, if there is a family of function with $\mathsf{D}(f) = \widetilde{\Omega}(\mathsf{UC}_{min}(f)^{1+k})$, then for the family of functions defined by $\widetilde{f} = \mathsf{OR}_{3\mathsf{UC}_{\min}(f)} \circ f_{\mathsf{DT}}$ satisfies $\mathsf{D}(\widetilde{f}) = \widetilde{\Omega}(s(\widetilde{f})^{2+k})$. Also, if we replace $\mathsf{D}(f)$ by $\mathsf{R}(f)$, $\mathsf{Q}(f)$ or $\mathsf{C}(f)$, we will get the same result.*

Proof of Theorem 1.2: Let us begin with the transitive functions $F_{1.1}$ from Section 4.1 which will desensitize to get the desired claim. From Theorem 2.40 and Observation 2.42 we have $\mathsf{D}(F_{1.1}) \geq \widetilde{\Omega}(\mathsf{UC}_{min}(F_{1.1})^2)$. Note that for our function $\mathsf{UC}_{min} = \mathsf{UC}_1$.

Let F_{DT} be the desensitized version of $F_{1.1}$. Define $F_{1.2}$ to be $\mathsf{OR}_{3\mathsf{UC}_{\min}(F_{\mathsf{DT}})} \circ F_{\mathsf{DT}}$. From Theorem 5.3, Observation 5.2 we have the theorem $F_{1.2}$ being the required function. □

6 Separation between quantum query complexity and certificate complexity

Aaronson et al. (2016) constructed functions that demonstrated quadratic separation between quantum query complexity and certificate complexity. Their function was not transitive. We modify their function to obtain a transitive function that gives a similar separation.

We start this section by constructing an encoding scheme for the inputs to the k-sum function such that the resulting function ENC-k-Sum is transitive. We then, similar to Aaronson et al. (2016), define $\mathsf{ENC} - \mathsf{Block}$-k-Sum function. Composing ENC-k-Sum with $\mathsf{ENC} - \mathsf{Block}$-k-Sum as outer function gives us $F_{1.3}$.

6.1 Function definition

Recall that, from Definition 2.31, for $\Sigma = [n^k]$, the function k-sum : $\Sigma^n \to \{0,1\}$ is defined as follows: on input $x_1, x_2, \ldots, x_n \in \Sigma$, if there exists k element $x_{i_1}, \ldots, x_{i_k}$, $i_1, \ldots, i_k \in [n]$, that sums to $0 \pmod{|\Sigma|}$ then output 1, otherwise output 0. We first define an encoding scheme for Σ.

6.1.1 Encoding scheme

Similar to Section 4.1.1 we first define the standard form of the encoding of $x \in \Sigma$ and then extend it by action of suitable group action to define all encodings that represent $x \in \Sigma$ where Σ is of size n^k for some $k \in \mathbb{N}$.

Fix some $x \in \Sigma$ and let $x = x_1 x_2 \ldots x_{k \log n}$ be the binary representation of x. The standard form of encoding of x is defined as follows: For all $i \in [k \log n]$ we encode x_i with with $4(k \log n + 2)$ bit Boolean string satisfying the following three conditions:

1. $x_i = x_{i1}x_{i2}x_{i3}x_{i4}$ where each x_{ij}, for $j \in [4]$, is a $(k \log n + 2)$ bit string,
2. if $x_i = 1$ then $|x_{i1}| = 1, |x_{i2}| = 0, |x_{i3}| = 2, |x_{i4}| = i + 2$, and
3. if $x_i = 0$ then $|x_{i1}| = 0, |x_{i2}| = 1, |x_{i3}| = 2, |x_{i4}| = i + 2$.

Having defined the standard form, other valid encodings of $x_i = (x_{i1}x_{i2}x_{i3}x_{i4})$ are obtained by the action of permutations $(12)(34), (13)(24) \in \mathsf{S}_4$ on the indices $\{i1, i2, i3, i4\}$. Finally if $x = \{(x_{ij} | i \in [k \log n], j \in [4]\}$, then $\{(x_{\sigma(i)\gamma(j)}) | \sigma \in \mathsf{S}_{k \log n}, \gamma \in T \subset \mathsf{S}_4\}$ is the set of all valid encoding for $x \in \Sigma$.

The decoding scheme follows directly from the encoding scheme. Given $y \in \{0,1\}^{k \log n(4k \log n + 8)}$, first break y into $k \log n$ blocks each of size $4k \log n + 8$ bits. If each block is a valid encoding then output the decoded string else output that y is not a valid encoding for any element from Σ.

6.1.2 Definition of the encoded function

ENC-k-Sum is a Boolean function that defined on n-bit as follows: Split the n-bit input into block of size $4k\log n(k\log n+2)$. We say such a block is a valid block iff it follows the encoding scheme in Section 6.1.1 i.e. represents a number from the alphabet Σ. The output value of the function is 1 iff there exists k valid block such that the number represented by the block in Σ sums to $0 \pmod{|\Sigma|}$.

$\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$ is a special case of the ENC-k-Sum function. We define it next. $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$ is a Boolean function that is defined on n-bit as follows: The input string is split into blocks of size $4k\log n(k\log n+2)$ and we say such block is a valid block iff it follows the encoding scheme of Section 6.1.1 i.e. represents a number from the alphabet Σ. The output value of the function is 1 iff there exists k valid block such that the number represented by the block in Σ sums to $0 (\text{mod } |\Sigma|)$ and the number of 1 in the other block is at least $6\times(k\log n)$. Finally, similar to Aaronson et al. (2016) define $F_{1.3}:\{0,1\}^{n^2}\to\{0,1\}$ to be $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}\circ\mathsf{ENC\text{-}k\text{-}Sum}$, with $k=\log n$.

The proof of Theorem 1.3 is the same as that of Aaronson et al. (2016). We give the proof here for completeness.

Proof of Theorem 1.3: We first show that the certificate complexity of $F_{1.3}$ is $\mathsf{O}(4nk^2\log n(k\log n+2)$. For this we show that every input to $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$, the outer function of $F_{1.3}$, has a certificate with $\widetilde{O}(k\times(4k\log n(k\log n+2)))$ many 0's and $O(n)$ many 1's. Also the inner function of $F_{1.3}$, i.e. ENC-k-Sum, has 1-certificate of size $\mathsf{O}(4k^2\log n(k\log n+2)$ and 0-certificate of size $\mathsf{O}(n)$. Hence, the $F_{1.3}$ function has certificate of size $\mathsf{O}(4nk^2\log n(k\log n+2))$.

Every 1-input of $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$ has k valid encoded blocks such that the number represented by them sums to $(0 \text{ mod } |\Sigma|)$. This can be certified using at most $\widetilde{\mathsf{O}}(k\times(4k\log n(k\log n+2)))$ number of 0's and all the 1's from every other block.

There are two types of 0-inputs of $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$. The first type of 0-input has at least one block in which a number of 1 is less than $6\times(k\log n)$ and the zeros of that block is a 0-certificate of size $\widetilde{O}(4k\log n(k\log n+2))$. The other type of 0-input is such that every block contains at least $6\times(k\log n)$ number of 1's. This type of 0-input can be certified by providing all the 1's in every block, which is at most $\mathsf{O}(n)$. This is because using all the 1's we can certify that even if the blocks were valid, no k-blocks of them are such that the number represented by them sums to $0 \pmod{|\Sigma|}$.

Next, we prove $\Omega(n^2)$ lower bound on the quantum query complexity of $F_{1.3}$. From Theorem 2.18, $\mathsf{Q}(F_{1.3})=\Omega(\mathsf{Q}(\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum})\mathsf{Q}(\mathsf{ENC\text{-}k\text{-}Sum}))$. Note that the inputs to k-sum are coming from some alphabet set Σ where size of Σ is at most n^k. To represent any such $x\in\sigma$ we need at most $k\log n$ many bits. Now if we take the alphabet set a bit larger that is if we need $10logn$ bits to represent any element from the larger alphabet set, then k-sum a sub-function of Block-k-Sum. Now our encoding scheme being again of size $O(k\log n)$ it follows that ENC-k-Sum is a sub-function of $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$. Since quantum query complexity of $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$ is $\Omega(\mathsf{Q}(\mathsf{ENC\text{-}k\text{-}Sum}/k\log n))$, from Theorem 2.32 the quantum query complexity of the $\mathsf{ENC}-\mathsf{Block\text{-}k\text{-}Sum}$ function is $\Omega\left(\frac{n^{\frac{k}{k+1}}}{k^{\frac{3}{2}}\log n(k\log n+2)}\right)$. Thus

$$\mathsf{Q}(F_{1.3})=\Omega\left(\frac{n^{\frac{2k}{k+1}}}{k^3\log n(k\log n+2)}\right).$$

Hence, $\mathsf{Q}(F_{1.3})=\widetilde{\Omega}(n^2)$, taking $k=\log n$. □

7 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 Table 1. Unfortunately, a number of cells in the table is 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? Thus following is a natural question.

  Open Problem 7.1. *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 Bun and Thaler (2020) that demonstrates quadratic separations between approximate degrees with sensitivity and several other complexity measures. It is thus natural to investigate the following open problem.

  Open Problem 7.2. *Come up with transitive functions that achieve similar separations for those pair of measures whose best-known separations are shown by Bun and Thaler (2020).*

- Recently Ben-David et al. (2021), Balodis (2021) and Balodis et al. (2021) came up with new classes of Boolean functions, starting with the HEX (see Ben-David et al. (2021)) and EAH (see Balodis et al. (2021)) 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 an open problem is a natural starting point.

  Open Problem 7.3. *Can the HEX and EAH functions be modified to transitive functions, while preserving their desired complexity measures up to 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 7.4. *Bridge the gaps in Table 1 by coming up with better upper bounds on complexity measures for transitive functions.*

Even with the recent results of Huang (2019) and Aaronson et al. (2021), 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

S. Aaronson. Quantum certificate complexity. *Journal of Computer and System Sciences*, 74(3):313–322, 2008. doi: 10.1016/j.jcss.2007.06.020.

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

S. Aaronson, S. Ben-David, R. Kothari, and A. Tal. Quantum implications of Huang's sensitivity theorem. *CoRR*, abs/2004.13231, 2020. URL https://arxiv.org/abs/2004.13231.

S. Aaronson, S. Ben-David, R. Kothari, S. Rao, and A. Tal. Degree vs. approximate degree and quantum implications of Huang's sensitivity theorem. In *STOC*, pages 1330–1342, 2021. doi: 10.1145/3406325.3451047.

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

A. Ambainis, K. Balodis, A. Belovs, T. Lee, M. Santha, and J. Smotrovs. Separations in query complexity based on pointer functions. *Journal of the ACM*, 64(5):32:1–32:24, 2017. doi: 10.1145/3106234.

S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. ISBN 978-0-521-42426-4. URL http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264.

K. Balodis. Several separations based on a partial Boolean function. *arXiv:2103.05593*, 2021. URL https://arxiv.org/abs/2103.05593.

K. Balodis, S. Ben-David, M. Göös, S. Jain, and R. Kothari. Unambiguous DNFs and Alon-Saks-Seymour. In *FOCS*, pages 116–124, 2021. doi: 10.1109/FOCS52979.2021.00020.

R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf. Quantum lower bounds by polynomials. *J. ACM*, 48(4):778–797, 2001. URL https://doi.org/10.1145/502090.502097.

A. Belovs and R. Spalek. Adversary lower bound for the k-sum problem. In *ITCS*, pages 323–328, 2013. doi: 10.1145/2422436.2422474.

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

S. Ben-David, A. M. Childs, A. Gilyén, W. Kretschmer, S. Podder, and D. Wang. Symmetries, graph properties, and quantum speedups. In *FOCS*, pages 649–660, 2020. doi: 10.1109/FOCS46700.2020.00066.

S. Ben-David, M. Göös, S. Jain, and R. Kothari. Unambiguous DNFs from Hex. *Electronic Colloquium on Computational Complexity*, 28:16, 2021.

C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997. doi: 10.1137/S0097539796300933.

G. Brassard, P. Hoyer, M. Mosca, and A. Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

H. Buhrman and R. 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.

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

S. 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.

S. Chakraborty, C. Kayal, and M. Paraashar. Separations between combinatorial measures for transitive functions. *arXiv:2103.12355*, 2021. URL `https://arxiv.org/abs/2103.12355`.

S. Chakraborty, C. Kayal, and M. Paraashar. Separations between combinatorial measures for transitive functions. In *ICALP*, volume 229, pages 36:1–36:20, 2022. URL `https://doi.org/10.4230/LIPIcs.ICALP.2022.36`.

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

Y. Gao, J. Mao, X. Sun, and S. Zuo. On the sensitivity complexity of bipartite graph properties. *Theoretical Computer Science*, 468:83–91, 2013. doi: 10.1016/j.tcs.2012.11.006.

J. Gilmer, M. E. Saks, and S. 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.

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

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

H. 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.

S. Kimmel. Quantum adversary (upper) bound. *Chicago Journal of Theoritical Computer Science*, 2013. doi: 10.4086/cjtcs.2013.004.

R. Kulkarni and A. Tal. On fractional block sensitivity. *Chicago Journal of Theoritical Computer Science*, 2016. URL `http://cjtcs.cs.uchicago.edu/articles/2016/8/contents.html`.

T. Lee and J. Roland. A strong direct product theorem for quantum query complexity. *Computational Complexity*, 22(2):429–462, 2013. doi: https://doi.org/10.1007/s00037-013-0066-8.

T. Lee, R. Mittal, B. W. Reichardt, R. Spalek, and M. Szegedy. Quantum query complexity of state conversion. In *FOCS*, pages 344–353, 2011. doi: 10.1109/FOCS.2011.75.

Q. Li and X. 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.

A. Montanaro. A composition theorem for decision tree complexity. *Chicago Journal of Theoritical Computer Science*, 2014. doi: 10.4086/cjtcs.2014.006.

S. Mukhopadhyay and S. 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.

N. Nisan and M. Szegedy. On the degree of Boolean functions as real polynomials. *Computational Complexity*, 4:301–313, 1994. doi: 10.1007/BF01263419.

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

J. Radhakrishnan and S. Sanyal. The zero-error randomized query complexity of the pointer function. In *FSTTCS 2016*, pages 16:1–16:13, 2016. doi: 10.4230/LIPIcs.FSTTCS.2016.16.

B. Reichardt. Reflections for quantum query algorithms. In *SODA*, pages 560–569, 2011. doi: 10.1137/1.9781611973082.44.

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

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

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

X. 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.

X. Sun, A. C. Yao, and S. 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.

A. Tal. Properties and applications of Boolean function composition. In *ITCS*, pages 441–454, 2013. doi: 10.1145/2422436.2422485.

G. Tardos. Query complexity, or why is it difficult to seperate $NP^A \cap coNP^A$ from P^A by random oracles A? *Combinatorica*, 9(4):385–392, 1989. doi: https://link.springer.com/article/10.1007/BF02125350.

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

A Known lower bounds for complexity measures for the class of transitive function

The following table represents the individual known separations and the known example for different complexity measures for the class of transitive function:

Measure	**known lower bounds**	**Known example**
D	$\Omega(\sqrt{N})$ Sun et al. (2004)	$O(\sqrt{N})$ Sun et al. (2004)
$\mathsf{R_0}$	$\Omega(\sqrt{N})$ Sun et al. (2004)	$O(\sqrt{N})$ Sun et al. (2004)
R	$\Omega(N^{\frac{1}{3}})$ $\mathsf{bs(f)} = O(\mathsf{R}(f))$	$O(\sqrt{N})$ Sun et al. (2004)
C	$\Omega(\sqrt{N})$	$O(\sqrt{N})$ $Tribe(\sqrt{N}, \sqrt{N})$
RC	$\Omega(N^{\frac{1}{3}})$ $\mathsf{bs(f)} = O(\mathsf{RC}(f))$	$O(\sqrt{N})$ $Tribe(\sqrt{N}, \sqrt{N})$
bs	$\Omega(N^{\frac{1}{3}})$ Sun (2007)	$\widetilde{O}(N^{\frac{3}{7}})$ Sun (2007), Drucker (2011)
s	$\Omega(N^{\frac{1}{8}})$ $\deg(f) = O(\mathsf{s(f)})^2$	$\Theta(N^{\frac{1}{3}})$ Chakraborty (2011)
λ	$\Omega(N^{\frac{1}{12}})$ $\mathsf{C(f)} = O(\lambda(\mathsf{f}))^6$	$\Theta(N^{\frac{1}{3}})$ Chakraborty (2011)
$\mathsf{Q_E}$	$\Omega(N^{\frac{1}{4}})$ $\mathsf{Q}(f) = O(\mathsf{Q_E}(f))$	$O(\sqrt{N})$ Sun et al. (2004)
$\deg$	$\Omega(N^{\frac{1}{4}})$ $\deg(f) = \Omega(\widetilde{\deg}(f))$	$O(\sqrt{N})$ Sun et al. (2004)
Q	$\Omega(N^{\frac{1}{4}})$ Sun et al. (2004)	$\widetilde{O}(N^{\frac{1}{4}})$ Sun et al. (2004)
$\widetilde{\deg}$	$\Omega(N^{\frac{1}{4}})$ Kulkarni and Tal (2016)	$\widetilde{O}(N^{\frac{1}{4}})$ Sun et al. (2004)

Table 3: In each row, for the measure A, the two entries a, b represents: (1) (Known lower bound) for all transitive Boolean function f, $A(f) = \Omega(a)$, and (2) (Known example) there exists a *transitive* function g such that $A(g) = O(b)$, where a and b are some polynomial in N.