k^+ Decision Trees

James Aspnes Eric Blais Murat Demirbas Ryan O'Donnell* Atri Rudra†
Steve Uurtamo

November 23, 2009

Abstract

Consider a wireless sensor network in which each node possesses a bit of information. Suppose all sensors with the bit 1 broadcast this fact to a central processor. If zero or one sensors broadcast, the central processor can detect this fact. If two or more sensors broadcast, the central processor can only detect that there is a "collision." Although collisions may seem to be a nuisance, they can in some cases help the central processor compute an aggregate function of the sensors' data.

Motivated by this scenario, we study a new model of computation for boolean functions: the 2^+ *decision tree*. This model is an augmentation of the standard decision tree model: now each internal node queries an arbitrary *set* of literals and branches on whether 0, 1, or at least 2 of the literals are true. This model was suggested in a work of Ben-Asher and Newman but does not seem to have been studied previously.

Our main result shows that 2^+ decision trees can "count" rather effectively. Specifically, we show that zero-error 2^+ decision trees can compute the threshold-of-t symmetric function with $O(t)$ expected queries (and that $\Omega(t)$ is a lower bound even for two-sided error 2^+ decision trees). Interestingly, this feature is not shared by 1^+ decision trees, demonstrating that "collisions can help." Our result implies that the natural generalization to k^+ decision trees does not give much more power than 2^+ decision trees. We also prove a lower bound of $\tilde{\Omega}(t) \cdot \log(n/t)$ for the *deterministic* 2^+ complexity of the threshold-of-t function, demonstrating that the randomized 2^+ complexity can in some cases be unboundedly better than deterministic 2^+ complexity.

Finally, we generalize the above results to arbitrary symmetric functions, and we discuss the relationship between k^+ decision trees and other complexity notions such as decision tree rank and communication complexity.

1 Introduction

Decision trees provide an elegant framework for studying the complexity of boolean functions. The internal nodes of a decision tree are associated with *tests* on the input; the branches leaving a node correspond to the outcomes of the associated test; and, the leaves of the tree are labeled with output values. The main parameter of interest is the *depth* of the decision tree; i.e., the maximum

*Supported in part by NSF grants CCF-0747250 and CCF-0915893, a Sloan fellowship, and an Okawa fellowship.
†A.R. and S.U. are supported in part by NSF CAREER grant CCF-0844796.

number of tests made over all inputs. The decision tree complexity of a particular boolean function is defined to be the minimum of this parameter over all decision trees computing the function.

We can define different decision tree models by restricting the set of tests that can be performed at each internal node. In the *simple* decision tree model, each test queries the value of a single bit of the input. This standard model is extremely well-studied in theoretical computer science; see e.g. the survey of Buhrman and de Wolf [5]. Other models include *linear* decision trees, where the tests are signs of linear forms on the bits (see, e.g., [8]); *algebraic* decision trees, the generalization to signs of low-degree polynomials (see, e.g., [3]); k-AND decision trees [4], where the tests are ANDs of up to k literals; k-bounded decision trees [15], the generalization to arbitrary functions of up to k bits; and $\mathbb{F}_2$-linear decision trees [13], where the tests are parities of sets of input bits.

Closer to the interests of the present article are models where the tests are *threshold functions* of the input bits. When the tests can be ORs of input variables, the model is known as *combinatorial group testing* (see the book [9]). When the tests can count and branch on the number of inputs in any subset of variables, the model is connected to that of *combinatorial search* (see the book [1]). Finally, when the tests allowed are ORs of any subset of input *literals*, we have the decision tree model studied by Ben-Asher and Newman [2]. We call this last model the *1*$^+$ *decision tree* model.

In this article, we initiate the study of the *2*$^+$ *decision tree* model. The tests in this model are on arbitrary subsets of the n input literals, and the branches correspond to the cases of either 0, 1, or at least 2 literals in the subset being true. (We will give a formal definition in Section 1.2.) We also introduce and examine the k^+ *decision tree* model, a natural generalization of the 2^+ model in which the branches correspond to 0, 1, 2, ..., $k-1$, or at least k literals being true.

1.1 Motivation

Our original motivation for considering 2^+ decision trees came from an application in wireless sensor networks. Consider the scenario where n sensor nodes can communicate directly with a central node (i.e., a "single-hop" network). Further, each node contains one bit of information (e.g., "Is the temperature more than 70°F?") and the central node wants to compute some aggregate function over this information (e.g., "Is the temperature more than 70°F for at least 10 of the sensors?"). How efficiently can the central node compute the aggregate function? The naive way to compute the aggregate function is to query each sensor node individually; with this approach, the number of queries required for the central node to compute the aggregate function is bounded below by the simple decision tree complexity of the function.

A better solution is to use the broadcast primitive available in wireless networks. With a single broadcast message, the central node may simultaneously ask all the nodes if their bit is 1. If the central node does not hear back any reply, it learns that all the sensor nodes' bits are 0. If it does hear a reply, then it knows that at least one of the sensors' bits is a 1. The complexity of computing the aggregate function with this approach is now determined by the 1^+ decision tree complexity of the function.

It is possible to make even better use of the broadcast primitive. When the central node has made a query and is listening for replies, there are three possible scenarios: either 0, 1, or at least 2 sensor nodes reply. In the first two cases, the outcome is clear. In the latter case, there will be

a collision in the sensors' replies. Conventional wisdom says that collisions are bad, and in fact protocols in wireless networks try to disambiguate collisions. In this scenario, however, collisions provide useful information: if the algorithm at the central node does not try to avoid collisions but instead uses them to determine when at least 2 sensors have replied, the complexity of computing the aggregate function is determined by the 2^+ decision tree complexity of the function. Indeed, there has been recent work in using existing wireless sensor nodes to detect collisions and using this capability to design more efficient protocols [6, 7].

A similar motivation for 2^+ decision trees appears in the work of Ben-Asher and Newman [2]. They were concerned with the PRAM model of parallel computation with n processors and a one-bit CRCW memory cell. This led naturally to the 1^+ model of computation, which Ben-Asher and Newman studied. The authors also mentioned that an Ethernet channel scenario — say, a single bus Ethernet network where the controller can detect any collisions in the network — yields a computational model equivalent to our 2^+ decision trees, but left the study of this model as an open problem.

1.2 Definitions

Before presenting our results, we introduce the necessary definitions and notation.

In this article, we are concerned with boolean functions; i.e., functions of the form $f : \{0,1\}^n \to \{0,1\}$. We write a typical input as $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, and write $|x|$ for its Hamming weight, namely $\sum_{i=1}^n x_i$. We also use the notation $[n] = \{1, \ldots, n\}$ and $\log(m) = \max\{\log_2(m), 1\}$.

A boolean function $f : \{0,1\}^n \to \{0,1\}$ is *monotone* if $f(x) \geqslant f(y)$ whenever $x \geqslant y$ coordinate-wise. A function f is *(totally) symmetric* if the value of $f(x)$ is determined by $|x|$. When f is symmetric, we write $f_0, f_1, \ldots, f_n \in \{0,1\}$ for the values of the function on inputs of Hamming weight $0, 1, \ldots n$, respectively. The functions which are both monotone and symmetric are the *threshold* functions. Given $0 \leqslant t \leqslant n+1$, the *$t$-threshold* function $T_n^t : \{0,1\}^n \to \{0,1\}$ is defined by $T_n^t(x) = 1$ iff $|x| \geqslant t$.

For an arbitrary symmetric function f we recall the integer parameter $\Gamma(f)$, first introduced by Paturi [17], and related to the longest interval centered around $n/2$ on which f is constant:

$$\Gamma(f) = \min_{0 \leqslant \ell \leqslant \lceil n/2 \rceil} \{\ell : f_\ell = f_{\ell+1} = \cdots = f_{n-\ell}\}.$$

E.g., for the threshold functions we have $\Gamma(T_n^t) = \min\{t, n+1-t\}$.

k^+ decision trees. Let $1 \leqslant k \leqslant n$ be integers. A *k^+ decision tree* T over n-bit inputs is a tree in which every leaf has a label from $\{0,1\}$, every internal node is labeled with two disjoint subsets $Q_{\text{pos}}, Q_{\text{neg}} \subseteq [n]$, and the internal nodes have $k+1$ outgoing edges labeled $0, 1, \ldots, k-1$, and k^+. Every internal node is also called a *query*; the corresponding sets Q_{pos} and Q_{neg} are called the *positive query set* and *negative query set*. Given a boolean input $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, the computation of T on x begins at the root of T. If that node is labeled by $(Q_{\text{pos}}, Q_{\text{neg}})$, computation proceeds along the edge labeled $0, 1, \ldots, k-1$, or k^+ according to Hamming weight of the literal set $\{x_i : i \in Q_{\text{pos}}\} \cup \{\overline{x}_j : j \in Q_{\text{neg}}\}$; i.e., $\sum_{i \in Q_{\text{pos}}} x_i + \sum_{j \in Q_{\text{neg}}} \overline{x}_j$. The label k^+ has the interpretation "at least k." The computation of T on x then proceeds recursively at the resulting child node. When

a leaf node is reached, the tree's output on x, denoted by $T(x)$, is the label of the leaf node. T is said to *compute* (or *decide*) a function $f : \{0,1\}^n \to \{0,1\}$ if and only if $T(x) = f(x)$ for all $x \in \{0,1\}^n$. The *cost* of T on x, denoted $\text{cost}(T, x)$, is the length of the path traced by the computation of T on x. The *depth* of the tree T is the maximum cost over all inputs. The *deterministic k^+ decision tree complexity* of a boolean function f, denoted $D^{(k^+)}(f)$, is the minimum depth of any k^+ decision tree that computes it.

As usual, we also introduce *randomized k^+* decision trees. Formally, these are probability distributions $\mathcal{P}$ over deterministic k^+ decision trees. The *expected cost* of $\mathcal{P}$ on input x is $\mathbf{E}_{T\sim\mathcal{P}}[\text{cost}(T, x)]$. The expected cost of $\mathcal{P}$ itself is the maximum expected cost over all inputs x. Given a boolean function $f : \{0,1\}^n \to \{0,1\}$, the *error* of $\mathcal{P}$ on input x is $\mathbf{Pr}_{T\sim\mathcal{P}}[T(x) \neq f(x)]$. We say that $\mathcal{P}$ computes f with *zero error* if this error is 0 for all inputs x (in particular, each deterministic T in the support of $\mathcal{P}$ must compute f). We say that $\mathcal{P}$ computes f with *two-sided error* if the error is at most $1/3$ for all inputs x. Note that both the expected cost measure and the error measure are worst-case over all inputs; we do not consider distributional complexity in this article. The *zero (respectively, two-sided) error randomized k^+ decision tree complexity* of a boolean function f, denoted $R_0^{(k^+)}(f)$ (respectively, $R_2^{(k^+)}(f)$), is the minimum expected cost over all distributions $\mathcal{P}$ which compute f with zero (respectively, two-sided) error. In this work, our randomized upper bounds will be for zero error k^+ computation and our randomized lower bounds for two-sided error.

Simple decision trees. The (simple) decision tree model can be thought of as the 1^+ model with the extra restriction that the query sets Q_{pos} and Q_{neg} satisfy $|Q_{\text{pos}} \cup Q_{\text{neg}}| = 1$ at each node. As is standard, we use the notation $D(f)$, $R_0(f)$, and $R_2(f)$ for the associated deterministic, zero error, and two-sided error complexities.

Another measure of complexity of functions associated with simple decision trees is the (simple decision tree) *rank* of a function. The notion of rank was first introduced by Ehrenfeucht and Haussler [11] in the context of learning theory, and has the following recursive definition. If T has a single (leaf) node we define $\text{RANK}(T) = 0$. Otherwise, supposing the two subtrees of T's root node are T_1 and T_2, we define $\text{RANK}(T) = \max\{\text{RANK}(T_1), \text{RANK}(T_2)\}$ if $\text{RANK}(T_1) \neq \text{RANK}(T_2)$, and $\text{RANK}(T) = \text{RANK}(T_1) + 1$ if $\text{RANK}(T_1) = \text{RANK}(T_2)$. For a boolean function f, we define $\text{RANK}(f)$ to be the minimum rank among simple decision trees computing f.

1.3 Results

We begin our analysis of k^+ decision trees with some simple properties that demonstrate the structural differences between k^+ decision trees and simple decision trees, and also give a flavor of the types of techniques that are useful for analyzing those trees. We outline those preliminary results in Section 1.3.1.

In Section 1.3.2, we present our first main result: a general characterization of the deterministic k^+ complexity of an arbitrary boolean function. One might expect this characterization to relate the k^+ complexity of a function to its (simple) decision tree complexity; instead, we show that the deterministic k^+ complexity of a function is closely related to its simple decision tree *rank*.

Our second main result is a more refined analysis of the 2^+ complexity of *symmetric* functions. As we show in Section 1.3.3, the deterministic and randomized 2^+ complexity of a symmetric function is determined by Paturi's Γ parameter.

1.3.1 Warm-up

We begin[1] with a simple question: what is the maximal deterministic k^+ complexity of any boolean function $f : \{0,1\}^n \to \{0,1\}$? It is well-known that most functions $f : \{0,1\}^n \to \{0,1\}$ have (trivial) deterministic decision tree complexity n. Our first result shows that, in contrast, the deterministic k^+ complexity of *every* function is less than n.

Theorem 1.1. [2] *For any $k \geqslant 2$ and every function $f : \{0,1\}^n \to \{0,1\}$,*

$$D^{(k^+)}(f) \leqslant O(n/\log k).$$

The bound in Theorem 1.1 is tight, and in fact as the following result shows, most functions have deterministic k^+ complexity $\Omega(n/\log k)$.

Theorem 1.2. *At least a $1 - 2^{-2^{n-1}}$ fraction of functions $f : \{0,1\}^n \to \{0,1\}$ satisfy the inequality*

$$D^{(k^+)}(f) \geqslant \Omega(n/\log k). \tag{1}$$

The proof of Theorem 1.2 uses a probabilistic argument. By establishing a connection between the deterministic k^+ complexity of a function and its deterministic 2-party communiction complexity, we identify an explicit function for which the inequality (1) holds.

Theorem 1.3. *Let $\textsc{eq} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be defined by $\textsc{eq}(x,y) = 1$ iff $x = y$. Then*

$$D^{(k^+)}(\textsc{eq}) \geqslant \Omega(n/\log k).$$

1.3.2 Connections to rank

Our first main result shows that, interestingly, the deterministic k^+ complexity of a function is closely related to its simple decision tree *rank*.[3]

Theorem 1.4. *For all $f : \{0,1\}^n \to \{0,1\}$,*

$$\textsc{rank}(f)/k \leqslant D^{(k^+)}(f) \leqslant O\big(\textsc{rank}(f)\log(n/\textsc{rank}(f))\big).$$

[1] **Eric:** Any suggestions for a better title for this sub-section?

[2] **Eric:** Should the Theorems in this sub-section be changed to Propositions?

[3] **Eric:** We should add more discussion of this result. In particular, we should discuss some of the implications of the Theorem. Any ideas?

1.3.3 Complexity of symmetric functions

The complexity of totally symmetric functions is not interesting in the simple decision tree model; it's easy to see that for nonconstant totally symmetric f we have $D(f) = n$ (and it's also known [16] that even $R_2(f) \geqslant \Omega(n)$). But the question becomes interesting in the 1^+ decision tree model. For example, we of course have $D^{(1^+)}(T_n^1) = 1$, but the value of even $D^{(1^+)}(T_n^2)$ is not immediately obvious. Ben-Asher and Newman point out that it is not hard to show that $D^{(1^+)}(T_n^t) \leqslant O(t \log(n/t))$, and their main theorem shows that this bound is tight:

Ben-Asher–Newman Theorem [2]. $D^{(1^+)}(T_n^t) = \Theta(t \log(n/t))$.

Ben-Asher and Newman also consider randomized complexity. They provide an incomplete proof of the fact that $R_0^{(1^+)}(T_n^2) \geqslant \Omega(\log n)$, and also observe that $R_2^{(1^+)}(T_n^2) = O(1)$.

Ben-Asher and Newman leave the study of the 2^+ model as an open problem. In particular, they ask if their main theorem can be extended to $D^{(2^+)}(T_n^t) \geqslant \Omega(t \log(n/t))$, observing only a trivial $\Omega(t)$ lower bound. We answer this open question up to a $\log t$ factor, and in fact do more: we characterize exactly (up to constants) the zero and two-sided error 2^+ complexities of all symmetric function, and provide a nearly-tight characterization of their deterministic 2^+ complexity.

Theorem 1.5. *For any symmetric boolean function $f : \{0,1\}^n \to \{0,1\}$, write $\Gamma = \Gamma(f)$. Then*[4]

$$\Omega\big((\Gamma/\log\Gamma) \cdot \log(n/\Gamma)\big) \leqslant D^{(2^+)}(f) \leqslant O\big(\Gamma \cdot \log(n/\Gamma)\big),$$

$$R_0^{(2^+)}(f) = \Theta(\Gamma),$$

$$R_2^{(2^+)}(f) = \Theta(\Gamma).$$

In particular, the above bounds hold with $\Gamma = \min(t, n+1-t)$ for threshold functions $f = T_n^t$.

An immediate corollary of Theorem 1.5 is that there is no polynomial relationship between deterministic and zero-error randomized 2^+ decision tree complexity; indeed, no bounded relationship at all. This is because for $t = O(1)$ we have $D^{(2^+)}(T_n^t) \geqslant \Omega(\log n)$, yet $R_0^{(2^+)}(T_n^t) = O(1)$. This latter result shows that the zero-error 2^+ decision tree model is quite powerful, being able to compute $T_n^{O(1)}$ with a number of queries *independent* of n.

In the proof of our upper bound $R_0^{(2^+)}(T_n^t) \leqslant O(t)$, we actually prove a stronger statement: any "k^+ query" can be exactly simulated with an expected $O(k)$ many 2^+ queries. Consequently we deduce that the zero error randomized k^+ decision tree complexity of *any* boolean function is at best $O(k)$ times smaller than its 2^+ decision tree complexity.

Corollary 1.6. *For any boolean function $f : \{0,1\}^n \to \{0,1\}$,*

$$R_0^{(k^+)}(f) \geqslant \Omega(R_0^{(2^+)}(f)/k).$$

The inequality in this corollary is best possible. Indeed, we show that for every symmetric function f it holds that $R_0^{(k^+)}(f) = \Theta(\Gamma(f)/k)$.

[4] **Eric:** We should generalize this result to characterize the k^+ complexity of symmetric functions?

1.3.4 Evasiveness and Yao-Karp conjectures

As noted above, the computation of totally symmetric functions is not interesting in the simple decision tree model. But some of the most interesting open problems in the theory of simple decision trees concern highly symmetric functions. Recall that a *graph property* for v-vertex graphs is a decision problem $f : \{0,1\}^{\binom{v}{2}} \to \{0,1\}$ which is invariant under all permutations of the vertices. Let f be a nonconstant *monotone* graph property. Two famous open problem in simple decision tree complexity are the evasiveness conjecture [18], that $D(f)$ must be equal $\binom{v}{2}$, and the Yao-Karp conjecture [19], that $R(f)$ must be $\Omega(v^2)$.

Guided by the evasiveness conjecture, one may be tempted to ask whether the "evasiveness 1^+ conjecture" might hold; that is, whether it is true that for every graph property f, $D^{(1^+)}(f) = \binom{v}{2}$. Our next result shows that this proposed conjecture is false.

Theorem 1.7. *For the connectivity graph property* $\textsc{Conn}_v : \{0,1\}^{\binom{v}{2}} \to \{0,1\}$ *it holds that* $D^{(1^+)}(\textsc{Conn}_v) \leqslant v(\lceil \log v \rceil + 1)$.

Since $R_0^{(1^+)}(f) \leqslant D^{(1^+)}(f)$ for every function f, Theorem 1.7 also implies that the Yao-Karp conjecture cannot be extended to the 1^+ decision tree model either.

2 Preliminary results

In this section, we prove the Theorems presented in Section 1.3.1.

2.1 A general upper bound

We begin by establishing an upper bound on the k^+ complexity of all boolean functions. The key to this upper bound is the following well-known result from combinatorial search and group testing.

Theorem 2.1 ([9][5])**.** *Any n-bit vector can be identified exactly with $O(n/\log n)$ many n^+ queries.*[6]

The proof of our first theorem, establishing an upper bound of $O(n/\log k)$ on the deterministic k^+ complexity of any boolean function $f : \{0,1\}^n \to \{0,1\}$ now follows directly from Theorem 2.1.

Theorem 1.1 (Restated)**.** For all $f : \{0,1\}^n \to \{0,1\}$, $D^{(k^+)}(f) \leqslant O(n/\log k)$.

Proof. Consider the following method for determining any given n-bit string with k^+ queries. First, we divide the set $[n]$ of indices into $\lceil n/k \rceil$ subsets size at most k. By Theorem 2.1, we can use $O(k/\log k)$ queries to determine each of the associated substrings. So in total, $O(\lceil n/k \rceil \cdot k/\log k) = O(n/\log k)$ queries are required to determine the input string exactly, which also lets us determine any function f of this input string. □

[5] **Eric:** Can we include a reference to the number of the theorem in the book?

[6] **Eric:** Out of interest, do we have an exact bound on the constant involved in this $O(\cdot)$?

2.2 A matching lower bound

A counting argument on k^+ decision trees with bounded depth establishes Theorem 1.2 and shows that the upper bound in Theorem 1.1 is sharp.

Theorem 1.2 (Restated and sharpened)**.** At least a $1 - 2^{-2^{n-1}}$ fraction of functions $f : \{0,1\}^n \to \{0,1\}$ satisfy the inequality $D^{(k^+)}(f) \geqslant (n/\log(k+1))(1 - o_n(1))$.

Proof. Let $T_k(d)$ be the number of k^+ decision trees of depth at most d. Recall that the total number of n-variable boolean functions is 2^{2^n}. Thus, if we show that for d strictly less than the claimed bound

$$T_k(d) < 2^{2^{n-1}}, \tag{2}$$

we will be done (as the $T_k(d)$ k^+ decision trees can compute at most $T_k(d)$ distinct functions).

To prove (2), we first estimate $T_k(d)$. Consider the root node of any k^+ decision tree of depth at most d. There are 3^n different choices for the query at the root (each index in $[n]$ can either be present in the positive query set, or in the negative query set, or in neither). Now each of the subtrees for each of the $k+1$ possible outcomes is in turn a k^+ decision tree of depth at most $d-1$. Thus, we get the recurrence relation:

$$T_k(d) = 3^n \cdot (T(d-1))^{k+1},$$

with $T(0) = 2$ (as a decision tree of depth 0, which does not make any queries can only compute the constant functions). One can check (e.g., by induction) that the recurrence is satisfied by

$$\log(T_k(d)) \leqslant n \log 3(1 + (k+1) + \cdots + (k+1)^{d-1}) + 2(k+1)^{d-1} \leqslant 2(k+1)^{d-1}(n \log 3 + 1),$$

where the second inequality follows from the fact that $k \geqslant 1$. The bound above implies (2) provided

$$1 + (d-1)\log(k+1) + \log(n \log 3 + 1) < n - 1,$$

which is implied by

$$d < \frac{n}{\log(k+1)} - \frac{\log n}{\log(k+1)} - \frac{4}{\log(k+1)} + 1 = \frac{n}{\log(k+1)}\left(1 - o_n(1)\right),$$

as desired. □

2.3 An explicit function with high k^+ complexity

Theorem 1.2 showed that most functions have a high deterministic decision tree complexity. We can furthermore simple explicit functions with this property by exploiting a connection between k^+ decision tree complexity and communication complexity.

Let $\mathrm{CC}(f)$ denote the deterministic 2-party communication complexity of f (for details, see [14]). We establish the following connection between the deterministic k^+ complexity of a function and its 2-party communication complexity.

Theorem 2.2. *For any $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, $D^{(k^+)}(f) \geqslant \Omega(\mathrm{CC}(f)/\log k)$.*

Proof. Let T be a deterministic k^+ decision tree of depth d that decides g. We will use T to design a protocol to decide g with at most $2\lceil\log(k+1)\rceil \cdot d$ bits of communication. Note that this proves the claimed result.

Let $x_1, \ldots, x_{2n}$ denote the input bits to g and assume Alice has $x_1, \ldots, x_n$ and Bob has $x_{n+1}, \ldots, x_{2n}$. Alice and Bob will essentially simulate the input on T. First let us assume that Alice and Bob are at the same node in T in their simulation. Let the node correspond to querying the sets $(Q_{\mathrm{pos}}, Q_{\mathrm{neg}}) \subseteq [2n] \times [2n]$. Define $(A_{\mathrm{pos}}, A_{\mathrm{neg}}) = (Q_{\mathrm{pos}}, Q_{\mathrm{neg}}) \cap [n] \times [n]$ and $(B_{\mathrm{pos}}, B_{\mathrm{neg}}) = (Q_{\mathrm{pos}}, Q_{\mathrm{neg}}) \setminus (A_{\mathrm{pos}}, A_{\mathrm{neg}})$. Now Alice evaluates the query $(A_{\mathrm{pos}}, A_{\mathrm{neg}})$ on the input $(x_1, \ldots, x_n)$ and communicates the answer to Bob (this takes at most $\lceil\log(k+1)\rceil$ bits as there are $k+1$ possible answers). Bob evaluates the query $(B_{\mathrm{pos}}, B_{\mathrm{neg}})$ on $(x_{n+1}, \ldots, x_{2n})$ and using the value he received from Alice can compute the answer of the query $(Q_{\mathrm{pos}}, Q_{\mathrm{neg}})$ on $(x_1, \ldots, x_{2n})$, which he communicates back to Alice using at most $\lceil\log(k+1)\rceil$ bits. Note that after Bob communicates back to Alice, both know which node in T to move to next. Finally, the proof is complete by noting that Alice and Bob can be "synchronized" at the beginning by making them both start at the root of T. □

Theorem 1.3 follows directly from Theorem 2.2.

Theorem 1.3 (Restated)**.** Let $\textsc{eq} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be defined by $\textsc{eq}(x,y) = 1$ iff $x = y$. Then $D^{(k^+)}(\textsc{eq}) \geqslant \Omega(n/\log k)$.

Proof. It is well-known that $\mathrm{CC}(\textsc{eq}) = n + 1$ (see, for example, [14, §1.3]). The proof is then completed by applying Theorem 2.2. □

3 Connection to rank

In this section, we prove Theorem 1.4 and show that the deterministic k^+ complexity of a boolean function is closely related to its (simple) decision tree rank. The proof of the theorem relies on the following properties of the rank function.

Proposition 3.1. *Let $r \geqslant 1$ be an integer and T be a decision tree of rank r. Then:*

(a) The roots of all the subtrees of T with rank r lie on a unique path in T (with the root as one of its end points).

(b) Let T have ℓ leaves and let $T_1, \ldots, T_\ell$ be decision trees of rank at most r'. Then replacing the ith leaf in T by T_i results in a decision tree of rank at most $r + r'$.

Proof. Both the properties can be proved by induction. For part (a), we induct on the depth of T. First, note that by the definition of rank, any subtree T' of T has rank at most r. Let T_0 and T_1 be the subtrees rooted at the children of the root of T. Again, by the definition of rank note that it cannot be the case that both T_0 and T_1 have rank exactly r. Now if both T_0 and T_1 have rank $r-1$, then we are done (as neither of them will have a rank r node). Otherwise, w.l.o.g. assume

that T_0 has rank r and T_1 has rank at most $r-1$. Then we recurse on T_0 to obtain the required path.

If T_i has rank exactly r' for every $1 \leqslant i \leqslant \ell$, then part (b) follows from the definition of rank. Further, due to the following claim, for the general case, one can assume that all the T_i's have rank exactly r'. Let T' be a subtree of T and let T'' be an arbitrary decision tree with higher rank than T'. If one replaces T' by T'' in T, then the rank of T does not decrease. The claim follows from a simple induction on the depth of the root of T' in T. If the depth is 1, then by the definition of rank after the substitution of T' with T'', all proper subtrees of T not contained in T' have the same rank as before. Again by the definition of rank, the rank of the root of T does not decrease after the substitution. The general case follows from a similar argument. □

The lower bound of Theorem 1.4 follows from a simulation argument that uses the fact that a single k^+ query can be simulated by a rank k decision tree.

Lemma 3.2. *Let $k \geqslant 1$ and $d \geqslant 0$ be integers. If a k^+ decision tree of depth d decides a boolean function f, then there is a decision tree of rank at most $k \cdot d$ that decides f.*

Proof. We begin by proving that any k^+ query on n variables can be simulated by a decision tree of rank at most k. Let the k^+ query be $(Q_{\text{pos}}, Q_{\text{neg}}) \subseteq [n] \times [n]$. The proof of the claim follows by induction on $k + |Q_{\text{pos}} \cup Q_{\text{neg}}|$.

For the base case of $k = 1$ and $|Q_{\text{pos}} \cup Q_{\text{neg}}| = 1$, it is easy to check that the query $(Q_{\text{pos}}, Q_{\text{neg}})$ can be simulated by a decision tree with one node that queries the variable in $Q_{\text{pos}} \cup Q_{\text{neg}}$ and takes a decision depending on whether the variable is a 1 or not. Note that this decision tree has a rank of 1.

Assume as the inductive hypothesis that any $(k')^+$ query $(Q'_{\text{pos}}, Q'_{\text{neg}})$ with $k' + |Q'_{\text{pos}} \cup Q'_{\text{neg}}| = N > 2$ can be simulated by a decision tree of rank at most k'. Let $(Q_{\text{pos}}, Q_{\text{neg}})$ be a k^+ query with $k + |Q_{\text{pos}} \cup Q_{\text{neg}}| = N + 1$. First assume that Q_{pos} is not empty and let $x_i \in Q_{\text{pos}}$. Consider the following decision tree (which we call $\mathcal{T}$) that simulates $(Q_{\text{pos}}, Q_{\text{neg}})$. Query x_i. If the answer is 1 then we need to simulate the $(k-1)^+$ query $(Q_{\text{pos}} \setminus \{x_i\}, Q_{\text{neg}})$, otherwise we need to simulate the k^+ query $(Q_{\text{pos}} \setminus \{x_i\}, Q_{\text{neg}})$. For each of these, by induction, we have decision trees of rank $k-1$ and k that can simulate the respective queries. Thus, by the definition of rank $\mathcal{T}$ has rank k, as desired. If $Q_{\text{pos}} = \emptyset$, then let $x_i \in Q_{\text{neg}}$. Again the required rank k decision tree is similar to the previous case (except the "roles" of 0 (as well as Q_{pos}) and 1 (and Q_{neg} respectively) are swapped). Finally if $Q_{\text{pos}} \cup Q_{\text{neg}} = \emptyset$, then the claim follows trivially.

To complete the proof, let $\mathcal{T}$ be a k^+ decision tree of depth d. We will now construct a decision tree $\mathcal{T}'$ of rank at most kd that simulates $\mathcal{T}$. The construction is pretty simple: replace each k^+ query in $\mathcal{T}$ by the corresponding decision tree of rank at most k that is guaranteed by the claim above. (In particular, consider the following recursive construction: replace the root in $\mathcal{T}$ by its corresponding decision tree τ and replicate the children of the root in $\mathcal{T}$ at the appropriate leaves in τ. Then recurse on these replicated k^+ decision trees of depth at most $d-1$.) Now one can prove that $\mathcal{T}'$ has rank at most kd by a simple induction on d and using part (b) of Proposition 3.1. □

The upper bound of Theorem 1.4 is implied by the following result.

Lemma 3.3. *Let $r, d \geqslant 1$ be integers. If a boolean function f has a decision tree with rank r and depth d, then there is a 1^+ decision tree of depth $O(r \log(d/r))$ that decides f.*

Proof. For the proof, we will need to solve the following problem: Given literals $\ell_1, \ldots, \ell_n$ (where each ℓ_i is either x_i or $\bar{x_i}$) find the minimum i^* such that $\ell_{i^*} = 1$ (if no such index exists, output fail). We now give an algorithm that finds such an i^* with $O(\log(i^*))$ 1^+ queries to the input bits $x_1, \ldots, x_n$. First the algorithm queries geometrically increasing sized sets to first get an estimate on i^*. More precisely in "round" $j \geqslant 0$ we do a 1^+ query[7] on the set $\{\ell_1, \ldots, \ell_{2^j}\}$ and we stop at the first round j^* such that the answer is 1^+ (if no such j^* exists, the algorithm stops and outputs fail). Note that $i^* \leqslant 2^{j^*} \leqslant 2i^*$. In the second step, the algorithm runs a binary search on the set $\{\ell_1, \ldots, \ell_{2^{j^*}}\}$ to find out the exact value of i^*. In other words, in the first step the algorithm runs a 1^+ query on $\{\ell_1, \ldots, \ell_{2^{j^*-1}}\}$. If the answer is 1^+, then the algorithm recurses on $\{\ell_1, \ldots, \ell_{2^{j^*-1}}\}$ else it recurses on $\{\ell_{2^{j^*-1}+1}, \ldots, \ell_{2^{j^*}}\}$. (The base case is when there is only one literal is left, whose index is i^*.) Both these steps take $O(\log(i^*))$ many 1^+ queries, as desired.

Let T be the given decision tree of rank r and depth d that decides f. We will show how to simulate T (i.e. given an input determine which leaf of T the input reaches) with $O(r \log(d/r))$ 1^+ queries. By part (a) of Lemma 3.1, T has exactly one path (call it P) that consists of roots of all the rank r subtrees in T. W.l.o.g. assume that the variables in P are (in order) $x_1, \ldots, x_t$. Further, define the literal corresponding to the t nodes as $\ell_1, \ldots, \ell_t$ such that to "get off" P at the ith node, $\bar{\ell_1} \wedge \cdots \wedge \bar{\ell}_{i-1} \wedge \ell_i$ needs to be satisfied[8]. Now given the procedure in the para above, we find out the minimum i such that given the values of $\ell_1, \ldots, \ell_t$, we will move off P after querying x_i. Note that by definition of P once we query x_i, we will move onto a decision tree of rank at most $r - 1$ and depth at most $d - i$. We now recurse on this subtree till we are left with a tree of depth 1, in which case we can find the correct leaf with a single 1^+ query. Note that since the rank decreases by at least one in each step, there can be at most $r' \leqslant r$ such steps. Let y_j $(1 \leqslant j \leqslant r')$ denote the amount by which the depth of the decision tree decreases in jth step. Note that the number of queries made by the algorithm in this step is $O(\log y_j)$. Also note that

$$\sum_{j=1}^{r'} y_j = d. \tag{3}$$

Thus, the total number of 1^+ queries made by our algorithm is

$$O\left(\log\left(\prod_{j=1}^{r'} y_j\right)\right) \leqslant O\left(r \log(d/r)\right),$$

where the inequality follows from the fact that given (3), the product $\prod_{j=1}^{r} d_j$ is maximized when $d_j = d/r$ for every j (and the fact that $r' \leqslant r$). □

[7] By running a 1^+ query on the set $\{\ell_1, \ldots, \ell_t\}$, we mean a 1^+ query $(Q_{\text{pos}}, Q_{\text{neg}})$ on $\{x_1, \ldots, x_t\}$, where $Q_{\text{pos}} = \{i | \ell_i = x_i\}$ and $Q_{\text{neg}} = \{j | \ell_j = \bar{x_j}\}$.

[8] There is a small technicality that for $i = t$, when either $\bar{\ell_1} \wedge \cdots \wedge \bar{\ell}_{t-1} \wedge \ell_t$ or $\bar{\ell_1} \wedge \cdots \wedge \bar{\ell}_{t-1} \wedge \bar{\ell_t}$ is satisfied, then we end up in a subtree of rank $r - 1$. However, the algorithm mentioned later on can handle this special case.

4 Lower bounds for symmetric functions

We prove the lower bounds of Theorem 1.5 in this section. The proofs for the upper bounds of the same theorem follow in the next section.

4.1 Deterministic lower bound

Lemma 4.1. *For any symmetric function $f : \{0,1\}^n \to \{0,1\}$ such that $\Gamma = \Gamma(f) > 2$,*

$$D^{(2^+)}f \geqslant \Omega\left((\Gamma/\log\Gamma)\cdot\log(n/\Gamma)\right).$$

(When $\Gamma \leqslant 2$, $D^{(2^+)}f \leqslant 1$.)

Proof. The statement about the $\Gamma \leqslant 2$ case is trivial (see the observation in Appendix A). For $\Gamma > 2$, assume without loss of generality that $f_{\Gamma-1} \neq f_\Gamma$. (If the inequality does not hold, then $f_{n-\Gamma} \neq f_{n-\Gamma+1}$ and we exchange the roles of the 0 and 1 labels in the rest of the proof.) Assume also for now that $\Gamma/3$ and $3n/\Gamma$ are integers. We describe an adversary that constructs inputs of weight $\Gamma-1$ or Γ while answering the 2^+ queries of the algorithm consistently.

The adversary maintains two pieces of information: a list of $m = \Gamma/3$ sets $U_1, \ldots, U_m$ of "undefined" variables and a set $I \subseteq [m]$ of the sets of undefined variables that are "active." Initially, $U_\ell = \{\frac{n}{m}(\ell-1)+1, \ldots, \frac{n}{m}\cdot\ell\}$ and $I = [m]$. For each query $(Q_{\text{pos}}, Q_{\text{neg}})$, the adversary proceeds as follows:

1. If there is an index $\ell \in I$ such that $|Q_{\text{pos}} \cap U_\ell| > |U_\ell|/m$, then the adversary answers "2^+", assigns the variables in $U_\ell \setminus Q_{\text{pos}}$ the value 0, and updates $U_\ell = U_\ell \cap Q_{\text{pos}}$. We refer to a query handled in this manner as an ℓ-query.

2. Otherwise, let $Q' \subseteq Q_{\text{neg}}$ be a set of size $|Q'| = \min\{2, |Q_{\text{neg}}|\}$. The adversary sets the variables in $U_\ell \cap (Q_{\text{pos}} \cup Q')$ to 0 and updates $U_\ell = U_\ell \setminus (Q_{\text{pos}} \cup Q')$ for each $\ell \in I$. It then returns the answer "0", "1", or "2^+", depending on the size of Q'. We refer to the query as a 0-query in this case.

After answering the query, each set U_ℓ of size $|U_\ell| < 3m$ is considered "defined." When the set U_ℓ is defined, the adversary updates $I = I \setminus \{\ell\}$. If I is still not empty, the adversary also sets 3 of the variables in U_ℓ to one. When the last set U_ℓ is defined, the adversary sets either 2 or 3 of its variables to one.

While not all the sets are defined, the answers of the adversary are consistent with inputs of weight $\Gamma-1$ and Γ. Therefore, the algorithm must make enough queries to reduce the sizes of $U_1, \ldots, U_m$ to less than $3m$ each. Let $q = q_0 + q_1 + \cdots + q_m$ be the number of queries made by the algorithm, where q_0 represents the number of 0-queries and q_ℓ represents the number of ℓ queries, for $\ell = 1, \ldots, m$.

Consider now a fixed $\ell \in [m]$. Each ℓ-query removes at most a $1 - 1/m$ fraction of the elements in U_ℓ, and each 0-query removes at most $|U_\ell|/m + 2 \leqslant 2|U_\ell|/m$ elements. So $|U_\ell| < 3m$ holds only when

$$\binom{n}{m}\left(\frac{1}{m}\right)^{q_\ell}\left(1-\frac{2}{m}\right)^{q_0} < 3m.$$

The inequality holds for each of $\ell = 1, \ldots, m$; taking the product of the m inequalities, we obtain

$$\left(\frac{n}{m}\right)^m \left(\frac{1}{m}\right)^{q_1+\cdots+q_m} \left(1-\frac{2}{m}\right)^{m\cdot q_0} < (3m)^m,$$

which implies

$$\left(\frac{n}{3m^2}\right)^m < m^{q_1+\cdots+q_m} \left(1-\frac{2}{m}\right)^{m\cdot q_0} \leqslant m^{q_1+\cdots+q_m} e^{2q_0} \leqslant m^{2(q_0+q_1+\cdots+q_m)}.$$

Taking the logarithm on both sides and dividing by $2\log m$, we get

$$\frac{m}{2\log m} \log \frac{n}{3m^2} < q_0 + q_1 + \cdots + q_m = q.$$

Recalling that $m = \Gamma/3$, we get the desired lower bound.

To complete the proof, we now consider the case where $\Gamma/3$ or n/m is not an integer. In this case, let Γ' be the largest multiple of 3 that is no greater than Γ, let $n' = n - (\Gamma - \Gamma')$, and let n'' be the largest multiple of m no greater than n'. Let the adversary fix the value of the last $n - n'$ variables to one and the previous $n' - n''$ variables to zero. We can now repeat the above argument with Γ' and n'' replacing Γ and n. □

4.2 Randomized lower bound

Lemma 4.2. *For any symmetric function $f : \{0,1\}^n \to \{0,1\}$ such that $\Gamma = \Gamma(f) > k$ and integer $k \geqslant \Gamma$,*

$$R_2^{(k^+)}(f) \geqslant \Omega\left(\Gamma/k\right).$$

When $\Gamma \leqslant k$, even $D^{(2^+)}f \leqslant 1$.

Proof. As in the proof of Lemma 4.1, the remark about $\Gamma \leqslant k$ is trivial; and, we may assume without loss of generality that $f_{\Gamma-1} \neq f_\Gamma$.

We prove the theorem using Yao's minimax principle. For $a \in \{\Gamma - 1, \Gamma\}$, let $\mathcal{D}_a$ be the uniform distribution over all the inputs with exactly a of the first $2\Gamma - 1$ variables set to 1 and all other variables set to 0. Let $\mathcal{D}$ be the distribution on inputs obtained by randomly selecting $\mathcal{D}_\Gamma$ or $\mathcal{D}_{\Gamma-1}$ with equal probability and then drawing an input from the selected distribution. We will show that any deterministic k^+ decision tree algorithm $\mathcal{A}$ that tries to determine the value of an input drawn from $\mathcal{D}$ with $q < (2\Gamma - 1)/(72k)$ queries must err with probability greater than $1/3$.

Let $\mathcal{P}$ be a random process for answering queries from $\mathcal{A}$ while constructing an input from $\mathcal{D}$. The random process $\mathcal{P}$ starts by selecting an index α uniformly at random from the range $[2\Gamma - 1]$, sets the value of the variables $x_{2\Gamma}, \ldots, x_n$ to 0, and leaves the values of the variables $x_1, \ldots, x_{2\Gamma-1}$ undefined for now.

When the algorithm makes a query $(Q_{\text{pos}}, Q_{\text{neg}})$, the process $\mathcal{P}$ begins by randomly ordering the literals in $Q_{\text{pos}} \cup Q_{\text{neg}}$. It then processes the literals one by one — either using the value already assigned to the associated variable or randomly assigning a value to the variable if it is currently undefined — until either: (a) k literals have been satisfied; or (b) all the literals in $Q_{\text{pos}} \cup Q_{\text{neg}}$

have been processed. In the first case, the answer "k^+" is returned; otherwise the answer returned corresponds to the number of satisfied literals.

After all the queries have been handled, the process $\mathcal{P}$ assigns random values to the variables in $[2\Gamma - 1] \setminus \{\alpha\}$ that are still undefined such that exactly $\Gamma - 1$ of them have the value 1. There are two possibilities: either x_α has been defined while answering one of the queries, or it is the last undefined variable. If the latter occurs, the process ends by flipping a fair coin to determine the value of x_α and the resulting Hamming weight of the input.

When x_α is the last undefined variable, its value is established by $\mathcal{P}$ *after* all the interaction with the algorithm is completed. Therefore, in this case the algorithm cannot predict the value of x_α — and by consequence the value of the function on the input — with probability greater than $1/2$. So to complete the proof, it suffices to show that the probability that x_α is defined while answering the queries is less than $1/3$.

Let m be the total number of variables whose values are defined while answering the queries and ζ correspond to the event where α is defined while answering the queries. Then

$$\begin{aligned}\mathop{\mathbf{Pr}}_{\mathcal{P}}[\zeta] &= \sum_{t \geqslant 0} \mathop{\mathbf{Pr}}_{\mathcal{P}}[m = t] \cdot \mathop{\mathbf{Pr}}_{\mathcal{P}}[\zeta \mid m = t] \\ &\leqslant \mathop{\mathbf{Pr}}_{\mathcal{P}}[m \geqslant (2\Gamma - 1)/6] + \mathop{\mathbf{Pr}}_{\mathcal{P}}[\zeta \mid m < (2\Gamma - 1)/6].\end{aligned}$$

We now show that both terms and the right-hand side are less than $1/6$.

First, let's bound the probability that $m \geqslant (2\Gamma - 1)/6$. Let m_j be the number of variables that are assigned a value by the process while answering the jth query. Each variable assigned a random value satisfies its corresponding literal in the query with probability $1/2$, so $\mathbf{E}[m_j] \leqslant 2k$ for $j = 1, \ldots, q$. When $q < (2\Gamma - 1)/(72k)$,

$$\mathbf{E}[m] = \mathbf{E}\left[\sum_{j=1}^{q} m_j\right] \leqslant 2kq < \frac{1}{6} \cdot \left(\frac{2\Gamma - 1}{6}\right).$$

Therefore, by Markov's inequality, $\mathbf{Pr}[m \geqslant (2\Gamma - 1)/6] < 1/6$.

Finally, let us consider $\mathbf{Pr}_{\mathcal{P}}[\zeta \mid m < (2\Gamma - 1)/6]$. Note that α is chosen uniformly at random from $[2\Gamma - 1]$ by $\mathcal{P}$, so when $m < (2\Gamma - 1)/6$ variables have been defined while answering the queries, the probability that α is one of those variables is less than $1/6$. □

5 Upper bounds for symmetric functions

The upper bound for the deterministic model follows almost immediately the Ben-Asher–Newman Theorem (see Appendix A for the details).

To complete the proof of Theorem 1.5, we now analyze the zero-error randomized complexity of symmetric functions. The key to establishing the upper bound is the following theorem.

Theorem 5.1. *Let $2 \leqslant k \leqslant t \leqslant n$ be an integers. Then $R_0^{(k^+)}(T_n^t) \leqslant O(t/k)$. Indeed, there is a randomized k^+ query algorithm which, given $x \in \{0,1\}^n$, correctly decides whether $|x|$ is $0, 1, \ldots, t-1$, or at least t, using an expected $O(t/k)$ queries.*

(Note that the assumption $t \geqslant k$ is not restrictive since the problem can be solved with one k^+ query if $t \leqslant k$.) Corollary 1.6 follows directly from this theorem (using linearity of expectation). See Appendix A for the details on why Theorem 5.1 implies the upper bounds in Theorem 1.5.

The key to proving Theorem 5.1 is the following "COUNT" algorithm:

Theorem 5.2. *Let $k \geqslant 2$. There is an algorithm* COUNT *which on input x, outputs $|x|$ in an expected $O(1 + |x|/k)$ many k^+ queries.*

We will also need the following easier result.

Proposition 5.3. *For each $k \geqslant 1$ and each real t satisfying $k \leqslant t \leqslant n$, there is an $O(t/k)$-query zero-error randomized k^+ query algorithm which on input $x \in \{0,1\}^n$ certifies that $|x| \geqslant t$, assuming $|x| \geqslant 4t$. More precisely, the algorithm has the following properties:*

(i) It makes (at most) $4t/k$ queries, with probability 1.

(ii) It outputs either "$|x| \geqslant t$" or "don't know."

(iii) If $|x| \geqslant 4t$, it outputs "$|x| \geqslant t$" with probability at least $1/4$.

(iv) If it ever outputs "$|x| \geqslant t$", then indeed $|x| \geqslant t$.

Proof. Let $m = \lfloor 4t/k \rfloor$. If $m \geqslant n$ we can solve the problem trivially by querying every bit. Otherwise, $1 \leqslant m \leqslant n$; we now partition the input coordinates randomly into m bins and perform a k^+ query on each. If at least t/k of the responses are "k^+" then we output "$|x| \geqslant t$" (which is certainly correct); otherwise we output "don't know."

Assume now that $|x| \geqslant 4t$; i.e., there are at least $4t$ "balls." The number of balls in a particular bin has distribution Binomial$(|x|, 1/m)$ and hence its mean $|x|/m$ is at least $4t/m \geqslant k$. It known that the median of a binomial random variable is at least the floor of its mean [12]; hence the probability that a bin has fewer than k balls is at most $1/2$. The expected fraction of bins with fewer than k balls is therefore at most $(1/2)m$; hence by Markov's inequality, the probability that there are more than $(2/3)m$ bins with fewer than k balls is at most $(1/2)/(2/3) = 3/4$. Thus with probability at least $1/4$ we see at least $(1/3)m$ bins with k^+ balls, and $(1/3)m = (1/3)\lfloor 4t/k \rfloor \geqslant t/k$. This completes the proof. □

We remark that Proposition 5.3 works even if our k^+ queries only return the response "k^+" or "$< k$"; in particular, it holds even when $k = 1$. Theorem 5.2, however, needs the full power of k^+ queries. We now show how Theorem 5.2 and Proposition 5.3 imply Theorem 5.1:

Proof of Theorem 5.1. The idea is to construct a dovetailing algorithm, as follows: First, run COUNT till $4t/k$ queries are made, or it halts. If it halts with $|x|$ then we are able to output the correct value for $T_n^t(x)$. Otherwise we "stop temporarily" and run the algorithm from Proposition 5.3. If that algorithm returns with "$|x| \geqslant t$", then stop and output $T_n^t(x) = 1$; otherwise resume executing COUNT for another $4t/k$ many queries. Repeat this process till either COUNT terminates or the algorithm of Proposition 5.3 returns with an answer of "$|x| \geqslant t$."

The fact that this dovetailing algorithm's output is always correct follows from the correctness in Theorem 5.2 and Proposition 5.3. As for the expected number of queries used, if the input x satisfies $|x| \geqslant 4t$, then by the third property in Proposition 5.3, the dovetailing algorithm stops after at most an expected $32t/k$ queries. On the other hand, if $|x| < 4t$, by Theorem 5.2, the dovetailing algorithm stops after an expected $O(t/k)$ queries. Thus for any input the dovetailing algorithm makes at most an expected $O(t/k)$ many queries. □

In the next section, we describe how to prove Theorem 5.2 using balls and bins analysis.

5.1 Balls and bins framework

The COUNT algorithm involves randomly partitioning the coordinates $[n]$ into some number of "bins." We think of the "balls" as being the indices for which the input x has a 1. Suppose we toss the balls (1-coordinates) into bins and then make a k^+ query on each bin. Recall that this tells us whether the number of balls is 0, 1, 2, ..., $k-1$, or $\geqslant k$. If a bin contains fewer than k balls, we say it *isolates* these balls.

Whenever a bin isolates balls, we have made progress: we know exactly how many 1's are in x in the bin's coordinates. We can thenceforth "throw away" these coordinates, remembering only the 1-count in them, and continue processing x on the substring corresponding to those coordinates in bins with at least k many 1's. Thus in terms of balls and bins, we can think of the task of counting $|x|$ as the task of isolating all of the balls. We note that the ability to isolate/count and throw away is the crucial tool that 2^+ queries gain us over 1^+ queries.

We now give a brief intuition behind the COUNT algorithm, with formal analysis in the next section. Although the algorithm doesn't actually know $|x|$, the number of balls, if it could partition using $2|x|/k$ bins then that would likely isolate a constant fraction of the balls. If we could do this repeatedly while only using $O(\#\text{ balls remaining}/k)$ many queries, we will be able to construct the desired COUNT algorithm. Since we don't know the number of balls remaining, we can try using 2, 4, 8, etc., many bins. If we get up to around the "correct number" $2|x|/k$, we're likely to isolate a good fraction of balls; we can then reset back to 2 bins and repeat. Although we pay a query for each bin, we don't have to worry too much about doubling the number of bins too far; resetting becomes highly likely once the number of bins is at least the correct number. More worrisome is the possibility of resetting too early; we don't want to just keep making tiny "nibbles." However, we will show that if the number of bins is too small, we are very unlikely to get many isolating bins; hence we *won't* reset.

5.2 The COUNT algorithm and proof of Theorem 5.2

Let's start with a simple warmup (recall that k^+ queries can simulate 2^+ queries for $k \geqslant 2$):

Proposition 5.4. *For any $A \geqslant 1$, there is an algorithm A-COUNT which on input x, outputs $|x|$ (or runs forever). If $|x| \leqslant A$, then the algorithm halts after an expected $O(A^2)$ many 2^+ queries.*

The proof is by a Birthday Paradox argument.

Proof. Toss the coordinates into A^2 bins and perform a 2^+ query on each. If all of the bins contain either 0 or 1 balls, we can report $|x|$. Otherwise, if we get at least one 2^+ response, we repeat. Assuming there are at most A balls, a standard "Birthday Paradox" analysis implies that the probability we get any collisions — i.e., any 2^+ responses — is at most $1/A^2 + 2/A^2 + \cdots + (A-1)/A^2 \leqslant A(A-1)/2A^2 \leqslant 1/2$. Since each trial uses A^2 queries, and we succeed in identifying $|x|$ with probability $1/2$ in each trial, the expected number of queries is at most $2A^2$. □

Using a "doubling-and-dovetailing" strategy, we could use this algorithm to output $|x|$ using an expected $O(|x|^2)$ 2^+ queries. Our goal is to reduce this to $O(|x|)$ for 2^+ queries, and in general, $O(1 + |x|/k)$ for k^+ queries. As we will see next, Proposition 5.4 is still a useful tool; it will let us assume in our analysis that $|x| \geqslant A$ for any fixed absolute constant A.

5.2.1 Reduction to the SHAVE algorithm

Let $1 \leqslant A < \infty$ and $0 < \varepsilon, \delta \leqslant 1$ be universal constants to be chosen later. The main work goes into proving the following:

Theorem 5.5. *There is an algorithm* SHAVE *which, on input x of weight $|x| \geqslant A$, halts in an expected $O(|x|/k)$ many k^+ queries and with probability at least ε isolates at least $\delta|x|$ balls.*

Assuming Theorem 5.5, we can complete the proof of Theorem 5.2 by repeatedly running SHAVE and dovetailing it with the A-COUNT algorithm.

Proof of Theorem 5.2. The idea is to repeatedly run the SHAVE algorithm, isolating balls. We also dovetail this repeated SHAVING with the A-COUNT algorithm. This allows us to freely assume, in analysis using the SHAVE algorithm, that $|x| \geqslant A$; for as soon as this fails to hold, Proposition 5.4 implies we halt after an expected $2 \cdot O(A^2) = O(1)$ more k^+ queries (recall that A is a universal constant). It remains to show that SHAVE will reduce the ball count below A in an expected $O(|x|/k)$ queries.

The analysis is straightforward. Suppose that at some stage we have at most w balls. We analyze the expected number of SHAVES until the number of balls is at most $(1-\delta/2)w$. So long as there are at least $(1-\delta/2)w$ balls, each time we run SHAVE we have at least an ε chance of "successfully" isolating at least $\delta(1-\delta/2)w \geqslant (\delta/2)w$ balls. As soon as we have a success, we drop to at most $(1-\delta/2)w$ balls. The expected number of runs of SHAVE to get a success is at most $1/\varepsilon$, and each uses at most an expected $O(w/k)$ queries. By Wald's Theorem, the expected number of queries to get a success is at most $(1/\varepsilon) \cdot O(w/k) = O(w/k)$ (since ε is a positive absolute constant). Finally, since we start with $|x|$ balls, the expected number of queries to get below A balls is certainly at most

$$O(\tfrac{|x|}{k}) + O(\tfrac{(1-\delta/2)|x|}{k}) + O(\tfrac{(1-\delta/2)^2|x|}{k}) + O(\tfrac{(1-\delta/2)^3|x|}{k}) + \cdots = (2/\delta)O(\tfrac{|x|}{k}) = O(\tfrac{|x|}{k})$$

(since δ is a positive absolute constant). □

5.2.2 The PARTITION algorithm

The basic building block for the SHAVE algorithm is the PARTITION(t) algorithm, which for $t \in \mathbb{N} = \{0, 1, 2, \dots\}$ makes exactly 2^t many k^+ queries.

PARTITION(t):

1. Toss the indices into 2^t bins and do a k^+ query on each bin.
2. Call a bin "good" if the number of balls in it is in the range $[\frac{1}{4}k, \frac{3}{4}k]$.
3. If the fraction of good bins is at least $\frac{1}{20}$, declare "accept" and isolate the balls.
4. Otherwise declare "reject" and *do not* isolate any balls.

Remark 5.6. In practice, one would always isolate balls, regardless of accepting or rejecting. However we do not isolate any balls on a reject for analysis purposes.

Note that in a run of PARTITION(t), the number of balls in a particular bin is distributed as Binomial($|x|, 1/2^t$). In particular, it has mean $|x|/2^t$.

Definition 5.7. We denote by $t^* \in \mathbb{N}$ the critical value of t: the unique number such that

$$\frac{1}{3}k < \frac{|x|}{2^{t^*}} \leqslant \frac{2}{3}k. \tag{4}$$

(We assume $|x| > 0$, which is fine since in proving Theorem 5.5 we have $|x| \geqslant A \geqslant 1$.)

We would really like to run PARTITION(t^*) repeatedly (with t^* changing as the number of remaining balls changes); doing so uses $\Theta(|x|/k)$ queries and, as we will see, isolates $\Omega(|x|)$ balls in expectation. Unfortunately, we do not in general know what t^* is, so we have to analyze what happens for other values of t. When $t \geqslant t^*$, the analysis of PARTITION is straightforward and follows. The other cases are handled in the next two sections.

Proposition 5.8. *Suppose we run* PARTITION*(t) with $t \geqslant t^* - B$ and it accepts. Then we isolate at least $\delta|x|$ balls, for some $\delta = \delta(B) > 0$.*

Proof. Since we accept, at least $\frac{1}{20} \cdot 2^t$ bins are good. Note that a good bin isolates all of the balls in it, since $\frac{3}{4}k < k$. Since good bins have at least $\frac{1}{4}k$ balls, we isolate at least

$$\frac{1}{4}k \cdot \tfrac{1}{20}2^t \geqslant \frac{1}{80} \cdot (k2^{t^*}) \cdot 2^{-B} \geqslant \frac{1}{80} \cdot \frac{3}{2}|x| \cdot 2^{-B}$$

balls, using (4). Taking $\delta = (3/160)2^{-B}$ completes the proof. □

5.2.3 PARTITION(t^*)'s acceptance probability

Remark 5.9. It will be convenient for analysis to assume that either $k = 2$ or $k \geqslant C$, where C is a large universal constant to be chosen later. Note that we can freely make this assumption: if $2 < k < C$, we can simply pretend (for analysis purposes) that $k = 2$. This causes an overall loss of a factor of $C/2$ in the query complexity, but this is just $O(1)$ since C will be a universal constant. And as noted before, k^+ queries can simulate 2^+ queries for $k > 2$.

Theorem 5.10. *If we run* PARTITION(t^*)*, it accepts with probability at least* $\frac{1}{20}$.

Proof. If $t^* = 0$, then by (4), we have $\frac{1}{4}k < |x| < \frac{3}{4}k < k$, which implies that a k^+ query will return the value of $|x|$. Further, since the indices are tossed in a single bin, PARTITION(0), will accept with probability 1. For the rest of the proof, we will assume that $t^* \geqslant 1$.

Our goal will be to show that when we partition into 2^{t^*} bins, the probability a particular bin is good is at least $\frac{1}{10}$. In that case, the expected fraction of good bins is at least $\frac{1}{10}$. It then follows that there is at least a $\frac{1}{20}$ fraction of good bins with probability at least $\frac{1}{20}$, and this completes the proof.

Letting X denote the number of balls in a particular bin, we have $X \sim \text{Binomial}(|x|, 1/2^{t^*})$, and we need to show that

$$\mathbf{Pr}[\tfrac{1}{4}k \leqslant X \leqslant \tfrac{3}{4}k] \geqslant \tfrac{1}{10}, \tag{5}$$

using the fact that $\mathbf{E}[X] = |x|/2^{t^*}$ is in the range $(\frac{1}{3}k, \frac{2}{3}k]$ (by (4)).

If k is large enough, this follows by Chernoff bounds. Specifically, the probability of $X < \frac{1}{4}k$ is no more than what it would be if $\mathbf{E}[X] = \frac{1}{3}k$, in which case the Chernoff bound gives an upper bound of $\exp(-\frac{1}{96}k)$. The probability of $X > \frac{3}{4}k$ is no more than what it would be if $\mathbf{E}[X] = \frac{2}{3}k$, in which case the Chernoff bound gives an upper bound of $\exp(-\frac{1}{204}k)$. By taking C to be a sufficiently large absolute constant (say, $C = 120$), we ensure that $\exp(-\frac{1}{96}k) + \exp(-\frac{1}{204}k) \leqslant \frac{9}{10}$ for all $k \geqslant C$. I.e., a particular bin is indeed good with probability at least $\frac{1}{10}$, assuming $k \geqslant C$.

By Remark 5.9, it suffices to further handle just the $k = 2$ case. In this case, showing (5) amounts to showing $\mathbf{Pr}[X = 1] \geqslant \frac{1}{10}$ using the fact that X is a binomial random variable with mean in $(\frac{2}{3}, \frac{4}{3}]$. Writing $X \sim \text{Binomial}(N, p)$ for simplicity ($N = |x|$, $p = 1/2^{t^*} \leqslant 1/2$), we have

$$\mathbf{Pr}[X = 1] = Np(1-p)^{N-1} \geqslant Np(1-p)^N \geqslant Np \cdot \exp(-(3/2)p)^N = (Np) \cdot \exp(-(3/2)Np).$$

Since the function $\lambda \cdot \exp(-(3/2)\lambda)$ is decreasing for $\lambda \geqslant \frac{2}{3}$, and $Np \leqslant \frac{4}{3}$ in our case, we conclude $\mathbf{Pr}[X = 1] \geqslant \frac{4}{3} \cdot \exp(-2) \geqslant \frac{1}{10}$, as desired. □

5.2.4 PARTITION(t)'s acceptance probability, $t < t^*$

Theorem 5.11. *There is an absolute constant* $B < \infty$ *such that if we run* PARTITION(t) *with* $t \leqslant t^* - B$*, it accepts with probability at most* $\sqrt{|x|}\exp(-\frac{1}{3}\sqrt{|x|})$.

Proof. Define a bin to be "semi-good" if the number of balls in it is at most $\frac{3}{4}k$. We will show that PARTITION(t) does not even see at least $\frac{1}{20}$ fraction of semi-good bins with probability more than $\sqrt{|x|}\exp(-\frac{1}{3}\sqrt{|x|})$.

The number of balls in a bin is binomially distributed with mean

$$\frac{|x|}{2^t} \geqslant \frac{|x|}{2^{t^*-B}} \geqslant \frac{2^B}{3}k, \tag{6}$$

using (4). Now by taking B large enough, a Chernoff bound will imply that the probability a bin has at most $\frac{3}{4}k$ balls is at most

$$\exp(-\frac{1}{3} \cdot |x|/2^t) \tag{7}$$

(indeed, $B = 5$ suffices). We now divide into two cases: $2^t < \sqrt{|x|}$ and $2^t \geqslant \sqrt{|x|}$.

Case 1: $2^t < \sqrt{|x|}$. In this case, from (7) we see that the probability of a semi-good bin is at most $\exp(-\frac{1}{3}\sqrt{|x|})$. By a union bound, the probability of getting even 1 semi-good bin is at most $2^t \exp(-\frac{1}{3}\sqrt{|x|}) \leqslant \sqrt{|x|}\exp(-\frac{1}{3}\sqrt{|x|})$. Thus we are done in this case, since accepting requires at least $\frac{1}{20}2^t$ good bins, and hence certainly at least 1 semi-good bin.

Case 2: $2^t \geqslant \sqrt{|x|}$. In this case, we simply combine (6) and (7) to conclude that the probability a particular bin is semi-good is at most $\exp(-\frac{2^B}{9}k)$, which is at most $\frac{1}{40}$ if we make B large enough (again, $B = 5$ suffices). We would now like to use a Chernoff bound to claim that the probability of at least a $\frac{1}{20}$ fraction of the 2^t bins being even semi-good is at most $\exp(-\frac{1}{3}\cdot 2^t) \leqslant \exp(-\frac{1}{3}\sqrt{|x|})$. This would complete the proof. The complication is that the events of the bins being semi-good are not independent. However, the (indicators of the) events are negatively associated (see, e.g., [10, Prop. 7.2, Theorem 13]), and hence the Chernoff bound is valid (see [10, Prop. 5]). □

5.2.5 Boosting slightly the success probability

The success probability of 1/20 that PARTITION(t^*) achieves is not quite high enough for us; we would prefer it to be, say, 9/10. This is straightforwardly achieved by repeating PARTITION, say, 50 times. In particular, define PARTITION$^+(t)$ to be the algorithm which runs PARTITION(t) 50 times, accepting if any of the runs accept (and using, say, the first isolation), rejecting if all runs reject. Using Proposition 5.8, Theorem 5.10, and Theorem 5.11, we easily conclude the following:

Theorem 5.12. *The algorithm* PARTITION$^+(t)$*, when run on input x, has the following properties:*

1. *It has k^+ query complexity $50 \cdot 2^t$.*
2. *If $t \geqslant t^* - B$ and the algorithm accepts, it isolates at least $\delta|x|$ balls.*
3. *If $t = t^*$, the algorithm accepts with probability at least 9/10.*
4. *If $t \leqslant t^* - B$, the algorithm accepts with probability at most $50\sqrt{|x|}\exp(-\frac{1}{3}\sqrt{|x|})$.*

5.2.6 Constructing SHAVE

In this section, we show how to use "doubling-and-dovetailing" with PARTITION$^+$ to construct the SHAVE algorithm, proving Theorem 5.5 and thus Theorem 5.2.

Proof of Theorem 5.5. The SHAVE algorithm is the following: Run PARTITION$^+(t)$ with "segments" of t's as follows:

$$0; \quad 0,1; \quad 0,1,2; \quad 0,1,2,3; \quad \cdots$$

Halt and isolate as soon as one of the runs accepts.

Proving the first property of SHAVE, that it halts in an expected $O(|x|/k)$ queries, is straightforward. The number of queries used in the "segment" $0, 1, 2, \ldots, t$ above is

$$50 \cdot (2^0 + 2^1 + 2^2 + \cdots + 2^t) \leqslant 100 \cdot 2^t.$$

We know that every segment ending in a $t \geqslant t^*$ includes a call to $\textsc{Partition}^+(t^*)$, which has probability at least $9/10$ of accepting. Thus the probability of executing the segment ending in $t^* + j$ is at most $1/10^j$ for $j \geqslant 1$. We conclude that the expected number of queries is at most

$$100 \cdot (2^0 + 2^1 + \cdots + 2^{t^*} + \frac{1}{10} 2^{t^*+1} + \frac{1}{10^2} 2^{t^*+2} + \frac{1}{10^3} 2^{t^*+3} + \cdots) \leqslant 200 \cdot 2^{t^*} + 100 \cdot \frac{2/10}{(1 - 2/10)} \cdot 2^{t^*} = 225 \cdot 2^{t^*}.$$

But $2^{t^*} < 3\frac{|x|}{k}$, by (4), so the expected number of queries is at most $775\frac{|x|}{k}$, as needed.

It remains to show the second property of Shave, that it has probability at least ε of isolating at least $\delta|x|$ balls. To do this, we will forget about all the segments ending in a $t > t^*$; we will still be able to show the desired result. We divide up the remaining invocations of $\textsc{Partition}^+(t)$ (from the segments up to and including t^*) into two parts: The "low" t's, those with $t \leqslant t^* - B$; and the "high" t's, those with $t > t^* - B$. Our goal is to show that with some positive probability ε, there is at least one accepting call to $\textsc{Partition}^+(t)$ with a high t. In that case, we know that at least $\delta|x|$ balls are isolated, as required.

Note that there are at most $t^*(t^* + 1)/2$ calls to $\textsc{Partition}^+$ (in the segments ending with $t \leqslant t^*$) and hence at most this many calls with with low t's. By a union bound, the probability that *any* call to $\textsc{Partition}^+$ with a low t succeeds is at most

$$\frac{t^*(t^*+1)}{2} \cdot 50\sqrt{|x|} \exp(-\tfrac{1}{3}\sqrt{|x|}) \leqslant O(\sqrt{|x|} \lg^2 |x|) \exp(-\tfrac{1}{3}\sqrt{|x|}),$$

where we used $2^{t^*} < 3\frac{|x|}{k}$ and hence $t^* \leqslant O(\log |x|)$. We have the liberty of assuming that $|x| \geqslant A$ for any large universal constant A we like; by taking A large enough (it can be checked that $A = 1500$ suffices), the above probability can be made smaller than $\frac{1}{3}$. We then have that with probability at least $\frac{2}{3}$, *all* calls to $\textsc{Partition}^+$ with low t's reject.

Certainly, the probability that at least one of the calls to $\textsc{Partition}^+$ with a high t accepts is at least $9/10$; this is because the call with $t = t^*$ alone accepts with at least this probability. Therefore, we conclude that we get at least one call with a high t which accepts with probability at least $(2/3) \cdot (9/10) = 3/5$. We can take this $3/5$ to be our value of ε. □

6 Evasiveness and Yao-Karp conjectures

In contrast to the evasiveness conjecture, we show that the basic monotone graph property of *connectivity* has $o(v^2)$ deterministic 1^+ complexity:

Theorem 1.7. For the connectivity graph property $\textsc{Conn}_v : \{0,1\}^{\binom{v}{2}} \to \{0,1\}$ it holds that $D^{(1^+)}(\textsc{Conn}_v) \leqslant v(\lceil \log v \rceil + 1)$.

Proof. We will prove the theorem by presenting a 1^+ decision tree protocol that runs the DFS algorithm to compute a spanning forest of the underlying graph. Thus, the algorithm can in fact even "count" the number of connected components (and in particular can compute $\textsc{Conn}_v$). For notational convenience let the vertex set of the underlying graph be denoted by $[v]$. Recall that every input bit corresponds to whether some edge (i, j) is present in the graph or not.

Before we show how to implement the DFS algorithm, we first present a sub-routine that solves the following problem. Given a vertex $i \in [v]$ and a subset $S \subseteq [v] \setminus \{i\}$, find out a vertex $j \in S$ such that the edge (i, j) is present (if such a vertex does not exist then output "fail"). This procedure is implemented using a simple divide and conquer strategy. First execute a 1^+ (positive) query on $(S, \emptyset)$. If the answer is 0 then output "fail." Otherwise note that there does exists a suitable $j \in S$ such that the edge (i, j) exists. We now start the following recursive procedure that takes as input a subset $S' \subseteq S$ and initially $S' = S$. We repeat the following procedure till $|S'| = 1$ (in which case $S' = \{j\}$ and we are done):

1. Pick an arbitrary an arbitrary subset $S'' \subset S'$ of size $\lceil |S'|/2 \rceil$ and run a 1^+ query on $(S'', \emptyset)$.

2. If the answer is zero set $S' \leftarrow S' \setminus S''$ otherwise set $S' \leftarrow S''$. Return to Step 1.

The correctness of this recursive procedure can be easily verified. Further, the number of recursive calls is at most $\lceil \log |S| \rceil$, which implies that the overall sub-routine makes at most $1 + \lceil \log |S| \rceil \leqslant 1 + \lceil \log v \rceil$ queries. Note that if the sub-routine outputs "fail," then it makes only one query.

Given the sub-routine above, the implementation of the DFS algorithm is fairly obvious. At any stage, the DFS algorithm maintains a spanning forest on a subset $F \subseteq [v]$ of vertices. Further, there is a "current" vertex $i \in F$. Initially pick $i \in [v]$ arbitrarily and set $F = \{i\}$. Repeat the following steps until $F = [v]$:

1. Use the sub-routine in the paragraph above on vertex i and $S = [v] \setminus F$.

2. If the subroutine outputs "fail" then set i to be the parent of the current vertex i. If no parent exists, set i to be an arbitrary vertex in $[v] \setminus F$ and update F to include this new current vertex i. Go back to step 1.

3. Otherwise let j be the vertex returned by the sub-routine. Make j the child of the current vertex in the spanning forest. Then set i to be j and add j to F. Return to step 1.

One can check that the above actually implements the DFS algorithm and thus, as a special case, computes $\textsc{Conn}_v$. To complete the proof we need to bound the total number of queries. Note that all the queries are made in the invocations to the sub-routine from the second paragraph. Further, all the invocations fall in either one of these two categories: (i) The invocation returned "fail" and there is exactly one such invocation for every vertex in $[v]$ and (ii) The invocation returned a vertex in $[v]$ and there is exactly one such invocation for each edge in the final spanning forest. Recall that the number of 1^+ queries made category (i) and (ii) invocations are 1 and at most $1 + \lceil \log v \rceil$ respectively. Finally, noting that the spanning forest has at most $v - 1$ edges, the total number of queries made by our algorithm is at most

$$1 \cdot v + (\lceil \log v \rceil + 1)(v - 1) \leqslant v + v \lceil \log v \rceil,$$

as desired. □

Acknowledgments

We thank Dana Angluin, Jiang Chen, Sarah C. Eisenstat, Onur Soysal and Yitong Yin for helpful discussions.

References

[1] M. Aigner. *Combinatorial Search*. Wiley-Teubner Series in Computer Science, 1988.

[2] Y. Ben-Asher and I. Newman. Decision trees with boolean threshold queries. *J. Comput. Syst. Sci.*, 51(3):495–502, 1995.

[3] M. Ben-Or. Lower bounds for algebraic computation trees. In *STOC '83*, pages 80–86, 1983.

[4] N. H. Bshouty. A subexponential exact learning algorithm for DNF using equivalence queries. *Information Processing Letters"*, 59(3):37–39, 1996.

[5] H. Buhrman and R. de Wolf. Complexity measures and decision tree complexity: A survey. *Theoretical Computer Science*, 288(1):21–43, 2002.

[6] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Principles Of Distributed Computing (PODC)*, pages 197–206, 2005.

[7] M. Demirbas, O. Soysal, and M. Hussain. Singlehop collaborative feedback primitives for wireless sensor networks. *INFOCOM miniconference*, 2008.

[8] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM Journal on Computing*, 5(2):181–186, 1976.

[9] D.-Z. Du and F. K. Hwang. *Combinatorial Group Testing and its Applications*. World Scientific, 2000.

[10] D. Dubhashi and D. Ranjan. Balls and bins: A study in negative dependence. *Random Structures and Algorithms*, 13(2):99–124, 1998.

[11] A. Ehrenfeucht and D. Haussler. Learning decision trees from random examples. *Information and Computation*, 82(3):231–246, 1989.

[12] K. Hamza. The smallest uniform upper bound on the distance between the mean and the median of the binomial and Poisson distributions. *Statistics and Probability Letters*, 23(1):21–25, 1995.

[13] E. Kushilevitz and Y. Mansour. Learning decision trees using the fourier spectrum. *SIAM Journal on Computing*, 22(6):1331–1348, 1993.

[14] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[15] S. Moran, M. Snir, and U. Manber. Applications of ramsey's theorem to decision tree complexity. *J. ACM*, 32(4):938–949, 1985.

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

[17] R. Paturi. On the degree of polynomials that approximate symmetric boolean functions (preliminary version). In *STOC '92*, pages 468–474, 1992.

[18] A. L. Rosenberg. On the time required to recognize properties of graphs: a problem. *SIGACT News*, 5(4):15–16, 1973.

[19] A. C.-C. Yao. Monotone bipartite graph properties are evasive. *SIAM Journal on Computing*, 17(3):517–520, 1988.

A From threshold functions to symmetric functions

We begin with the following simple observation:

Lemma A.1. *For any $2 \leqslant k \leqslant t \leqslant n$, assume that there is a deterministic k^+ query algorithm that on input $x \in \{0,1\}^n$ can determine whether $|x|$ is $0, 1, \ldots, t-1$ or at least t with $Q_1^{(k)}(t,n)$ queries; and also assume there is a deterministic k^+ query algorithm that can determine if $|x|$ is $n, n-1, \ldots, n-t+1$ or at most $n-t$ with $Q_0^{(k)}(t,n)$ queries. Then for any symmetric function f, there is a deterministic k^+ query algorithm that decides f with $Q_1^{(k)}(\Gamma(f),n) + Q_0^{(k)}(\Gamma(f),n)$ k^+ queries. The same result also holds in the zero-error randomized k^+ model of computation (where we refer to the expected number of queries made).*

Proof. The algorithm for f is a simple one: first run one of the given algorithms to decide if $|x|$ is $0, 1, \ldots, \Gamma(f)-1$ or at least $\Gamma(f)$. Note that if $|x| < \Gamma(f)$, then since we know $|x|$, we can determine $f(x)$. Similarly, using the other algorithm we can determine $|x|$ if $|x| > n - \Gamma(f)$ in which case we again know $f(x)$ exactly. Thus, the only case left is $\Gamma(f) \leqslant |x| \leqslant n - \Gamma(f)$. However, in this case by definition of $\Gamma(\cdot)$, the value of f is constant (and hence, we can determine $f(x)$).

It is immediate that the query complexity of the algorithm above is at most $Q_1^{(k)}(\Gamma(f),n) + Q_0^{(k)}(\Gamma(f),n)$ in the deterministic case; the zero-error randomized case follows by linearity of expectation. □

It is easy to verify that the algorithm for T_n^t given by Ben-Asher Newman actually determines, with $O(t\log(n/t))$ 1^+ queries, whether $|x|$ is $0, 1, \ldots, t-1$ or at least t. Further, by changing the roles of 0 and 1, the same algorithm can also determine if $|x|$ is $n, n-1, \ldots, n-t+1$ or at most $n-t$ with $O(t\log(n/t))$ 1^+ queries. Since any k^+ query can simulate a 1^+ query, we have that both $Q_0^{(k)}(t,n)$ and $Q_1^{(k)}(t,n)$ are $O(t\log(n/t))$. This along with Lemma A.1 proves the upper bound for deterministic 2^+ decision tree complexity in Theorem 1.5.

By Theorem 5.1, zero error randomized algorithms we may use $Q_1^{(k)}(t,n)$ is $O(t/k)$. Again, switching the roles of 0 and 1 in the proof of Theorem 5.1, we may use $Q_0^{(k)}(t,n) = O(t/k)$. Thus, Lemma A.1 proves the rest of the upper bounds in Theorem 1.5.

B Rank of Symmetric Functions

In this section, we establish for symmetric functions f a relation between $\textsc{rank}(()f)$ and $\Gamma(f)$. To do so, we first introduce the *gap* measure of symmetric boolean functions.

Definition B.1. The gap of the symmetric function $f : \{0,1\}^n \to \{0,1\}$ is

$$\textsc{gap}(f) = \max_{0 \leqslant a \leqslant b \leqslant n} \{b - a : f_a = f_{a+1} = \cdots = f_b\}.^9$$

There is a precise relation between the rank of a symmetric function f and the value of $\textsc{gap}(f)$.

[9] Note that $\textsc{gap}(f)$ does not quite measure the length of the longest gap in f; rather $\textsc{gap}(f)+1$ does. But, as we see in the rest of the section, for our purposes the present definition is (slightly) more convenient.

Lemma B.2. *For any symmetric function $f : \{0,1\}^n \to \{0,1\}$,*

$$\textsc{rank}(f) + \textsc{gap}(f) = n.$$

Proof. When f is a constant function, then $\textsc{rank}(f) = 0$ and $\textsc{gap}(f) = n$, so the Lemma holds. We now show that the Lemma also holds when f is not a constant function by induction on n.

When $n = 1$, the only non-constant functions are $f = x_1$ and $f = \overline{x}_1$, in which case $\textsc{gap}(f) = 0$ and $\textsc{rank}(f) = 1$.

Consider now a fixed $n > 1$, and let T be a tree computing f. Since f is non-constant, the tree has depth at least one. Without loss of generality, let T query x_n at its root. Define the functions $g_0, g_1 : \{0,1\}^{n-1} \to \{0,1\}$ by letting

$$\begin{aligned} g_0(x_1, \ldots, x_{n-1}) &= f(x_1, \ldots, x_{n-1}, 0) \text{ and} \\ g_1(x_1, \ldots, x_{n-1}) &= f(x_1, \ldots, x_{n-1}, 1). \end{aligned}$$

Let (a_0, b_0) represent the extremities of the longest gap in f. If there are multiple gaps with maximal length, choose the left-most one. The longest gap in g_0 has the same length as the longest gap in f when $b_0 < n$, and is shorter by one otherwise, so

$$\textsc{gap}(g_0) = \begin{cases} \textsc{gap}(f) - 1 & \text{, when } b_0 = n, \\ \textsc{gap}(f) & \text{, when } b_0 < n. \end{cases}$$

Similarly, let (a_1, b_1) represent the extremities of the right-most longest gap in f. Then

$$\textsc{gap}(g_1) = \begin{cases} \textsc{gap}(f) - 1 & \text{, when } a_1 = 0, \\ \textsc{gap}(f) & \text{, when } a_1 > 0. \end{cases}$$

The functions g_0 and g_1 are both symmetric, so by the induction hypothesis

$$\textsc{rank}(g_0) = (n-1) - \textsc{gap}(g_0) = \begin{cases} n - \textsc{gap}(f) & \text{, when } b_0 = n, \\ n - \textsc{gap}(f) - 1 & \text{, when } b_0 < n \end{cases}$$

and

$$\textsc{rank}(g_1) = (n-1) - \textsc{gap}(g_1) = \begin{cases} n - \textsc{gap}(f) & \text{, when } a_1 = 0, \\ n - \textsc{gap}(f) - 1 & \text{, when } a_1 > 0. \end{cases}$$

The subtrees from the root of T compute g_0 and g_1, so

$$\textsc{rank}(T) \geqslant n - \textsc{gap}(f).$$

Furthermore, when f is not constant we can't have that both $a_1 = 0$ and $b_0 = n$, so at least one of $\textsc{rank}(g_0)$ or $\textsc{rank}(g_1)$ must equal $n - \textsc{gap}(f) - 1$. Therefore, if we choose the subtrees of T such that they compute g_0 and g_1 with optimal rank, we get that

$$\textsc{rank}(T) \leqslant n - \textsc{gap}(f).$$

To complete the proof of the Lemma, we note that the above argument is valid for any tree T that computes f, so $\textsc{rank}(f) = n - \textsc{gap}(f)$. □

With the above Lemma, we can now prove the main result of this section.

Corollary B.3. *For any symmetric function* $f : \{0,1\}^n \to \{0,1\}$*,*

$$\frac{\textsc{rank}(f)}{2} \leqslant \Gamma(f) \leqslant \textsc{rank}(f).$$

Proof. Since $f_{\Gamma(f)} = f_{\Gamma(f)+1} = \cdots = f_{n-\Gamma(f)}$, $\textsc{gap}(f) \geqslant n - 2\Gamma(f)$. Combining this inequality with Lemma B.2, we get that $\Gamma(f) \geqslant \frac{\textsc{rank}(f)}{2}$.

For the upper bound, we first note that either (i) $\{\Gamma(f), \ldots, n - \Gamma(f)\}$ is contained inside the subset of $[n+1]$ corresponding to the largest gap in f; or (ii) $\{\Gamma(f), \ldots, n - \Gamma(f)\}$ is disjoint from the largest gap. In case (i), at least one end point of the largest gap is from $\{\Gamma(f), n - \Gamma(f)\}$, which implies that $\Gamma(f) \leqslant n - \textsc{gap}(f)$. In case (ii), $\textsc{gap}(f) \leqslant \Gamma(f)$. This implies that $n - \textsc{gap}(f) \geqslant n - \Gamma(f) \geqslant \Gamma(f)$, where the last inequality follows from the fact that $\Gamma(f) \leqslant n/2$. Thus in either case, we have $\Gamma(f) \leqslant n - \textsc{gap}(f)$ and Lemma B.2 completes the proof. □