Approximating the Noise Sensitivity of a Monotone Boolean Function

Ronitt Rubinfeld
CSAIL at MIT, Cambridge, MA, USA
Blavatnik School of Computer Science at Tel Aviv University, Israel
`https://people.csail.mit.edu/ronitt/`
ronitt@csail.mit.edu

Arsen Vasilyan
CSAIL at MIT, Cambridge, MA, USA
vasilyan@mit.edu

Abstract

The *noise sensitivity* of a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is one of its fundamental properties. For noise parameter δ, the noise sensitivity is denoted as $NS_\delta[f]$. This quantity is defined as follows: First, pick $x = (x_1, \ldots, x_n)$ uniformly at random from $\{0,1\}^n$, then pick z by flipping each x_i independently with probability δ. $NS_\delta[f]$ is defined to equal $\Pr[f(x) \neq f(z)]$. Much of the existing literature on noise sensitivity explores the following two directions: (1) Showing that functions with low noise-sensitivity are structured in certain ways. (2) Mathematically showing that certain classes of functions have low noise sensitivity. Combined, these two research directions show that certain classes of functions have low noise sensitivity and therefore have useful structure.

The fundamental importance of noise sensitivity, together with this wealth of structural results, motivates the algorithmic question of approximating $NS_\delta[f]$ given an oracle access to the function f. We show that the standard sampling approach is essentially optimal for general Boolean functions. Therefore, we focus on estimating the noise sensitivity of *monotone* functions, which form an important subclass of Boolean functions, since many functions of interest are either monotone or can be simply transformed into a monotone function (for example the class of *unate* functions consists of all the functions that can be made monotone by reorienting some of their coordinates [21]).

Specifically, we study the algorithmic problem of approximating $NS_\delta[f]$ for monotone f, given the promise that $NS_\delta[f] \geq 1/n^C$ for constant C, and for δ in the range $1/n \leq \delta \leq 1/2$. For such f and δ, we give a randomized algorithm performing $O\left(\frac{\min(1,\sqrt{n}\delta \log^{1.5} n)}{NS_\delta[f]} \text{poly}\left(\frac{1}{\epsilon}\right)\right)$ queries and approximating $NS_\delta[f]$ to within a multiplicative factor of $(1 \pm \epsilon)$. Given the same constraints on f and δ, we also prove a lower bound of $\Omega\left(\frac{\min(1,\sqrt{n}\delta)}{NS_\delta[f]\cdot n^\xi}\right)$ on the query complexity of any algorithm that approximates $NS_\delta[f]$ to within any constant factor, where ξ can be any positive constant. Thus, our algorithm's query complexity is close to optimal in terms of its dependence on n.

We introduce a novel *descending-ascending view* of noise sensitivity, and use it as a central tool for the analysis of our algorithm. To prove lower bounds on query complexity, we develop a technique that reduces computational questions about query complexity to combinatorial questions about the existence of "thin" functions with certain properties. The existence of such "thin" functions is proved using the probabilistic method. These techniques also yield new lower bounds on the query complexity of approximating other fundamental properties of Boolean functions: the *total influence* and the *bias*.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation

Keywords and phrases Monotone Boolean functions, noise sensitivity, influence

Digital Object Identifier 10.4230/LIPIcs.APPROX-RANDOM.2019.52

Category RANDOM

Related Version Full version: `https://arxiv.org/abs/1904.06745`

Funding *Ronitt Rubinfeld*: NSF grants CCF-1650733, CCF-1733808, IIS-1741137 and CCF-1740751
Arsen Vasilyan: NSF grant IIS-1741137, EECS SuperUROP program, the MIT Summer UROP program and the DeFlorez Endowment Fund

Acknowledgements We are grateful to the anonymous referees, Daniel Grier and MIT EECS Communication Lab for helpful comments and suggestions.

1 Introduction

Noise sensitivity is a property of any Boolean function $f : \{0,1\}^n \to \{0,1\}$ defined as follows: First, pick $x = (x_1, \dots, x_n)$ uniformly at random from $\{0,1\}^n$, then pick z by flipping each x_i independently with probability δ. Here δ, the noise parameter, is a given positive constant no greater than $1/2$ (and at least $1/n$ in the interesting cases). With the above distributions on x and z, the noise sensitivity of f, denoted as $NS_\delta[f]$, is defined as follows:

$$NS_\delta[f] \stackrel{\text{def}}{=} \Pr[f(x) \neq f(z)] \tag{1}$$

Noise sensitivity was first explicitly defined by Benjamini, Kalai and Schramm in [3], and has been the focus of multiple papers: e.g. [3, 7, 8, 10, 12, 17, 22]. It has been applied to learning theory [4, 7, 8, 9, 10, 11, 15], property testing [1, 2], hardness of approximation [13, 16], hardness amplification [19], combinatorics [3, 12], distributed computing [18] and differential privacy [7]. Multiple properties and applications of noise sensitivity are summarized in [20] and [21]. Much of the existing literature on noise sensitivity explores the following directions: (1) Showing that functions with low noise-sensitivity are structured in certain ways. (2) Mathematically showing that certain classes of functions have low noise sensitivity. Combined, these two research directions show that certain classes of functions have low noise sensitivity and therefore have useful structure.

The fundamental importance of noise sensitivity, together with this wealth of structural results, motivates the algorithmic question of approximating $NS_\delta[f]$ given an oracle access to the function f. It can be shown that standard sampling techniques require $O\left(\frac{1}{NS_\delta[f]\epsilon^2}\right)$ queries to get a $(1+\epsilon)$-multiplicative approximation for $NS_\delta[f]$. In the full version of the paper, we show that this is optimal for a wide range of parameters of the problem. Specifically, it cannot be improved by more than a constant when ϵ is a sufficiently small constant, δ satisfies $1/n \leq \delta \leq 1/2$ and $NS_\delta[f]$ satisfies $\Omega\left(\frac{1}{2^n}\right) \leq NS_\delta[f] \leq O(1)$.

It is often the case that data possesses a known underlying structural property which makes the computational problem significantly easier to solve. A natural first such property to investigate is that of monotonicity, as a number of natural function families are made up of functions that are either monotone or can be simply transformed into a monotone function (for example the class of *unate* functions consists of all the functions that can be made monotone by reorienting some of their coordinates [21]). Therefore, we focus on estimating the noise sensitivity of monotone functions.

The approximation of the related quantity of total influence (henceforth just influence) of a monotone Boolean function in this model was previously studied by [24, 23][1]. Influence, denoted by $I[f]$, is defined as n times the probability that a random edge of the Boolean cube (x, y) is *influential*, which means that $f(x) \neq f(y)$. (This latter probability is sometimes referred to as the *average sensitivity*). It was shown in [24, 23] that one can approximate the influence of a monotone function f with only $\tilde{O}\left(\frac{\sqrt{n}}{I[f]\text{poly}(\epsilon)}\right)$ queries, which for constant ϵ beats the standard sampling algorithm by a factor of $\sqrt{n}$, ignoring logarithmic factors.

[1] [23] is the journal version of [24] and contains a different algorithm that yields sharper results. However, our algorithmic techniques build on the conference version [24].

Despite the fact that the noise sensitivity is closely connected to the influence [20, 21], the noise sensitivity of a function can be quite different from its influence. For instance, for the parity function of all n bits, the influence is n, but the noise sensitivity is $\frac{1}{2}(1-(1-2\delta)^n)$ (such disparities also hold for monotone functions, see for example the discussion of influence and noise sensitivity of the majority function in [21]). Therefore, approximating the influence by itself does not give one a good approximation to the noise sensitivity.

The techniques in [24, 23] also do not immediately generalize to the case of noise sensitivity. The result in [24, 23] is based on the observation that given a descending[2] path on the Boolean cube, at most one edge in it can be influential. Thus, to check if a descending path of any length contains an influential edge, it suffices to check the function values at the endpoints of the path. By sampling random descending paths, [24, 23] show that one can estimate the fraction of influential edges, which is proportional to the influence.

The most natural attempt to relate these path-based techniques with the noise sensitivity is to view it in the context of the following process: first one samples x randomly, then one obtains z by taking a random walk from x by going through all the indices in an arbitrary order and deciding whether to flip each with probability δ. The intermediate values in this process give us a natural path connecting x to z. However, this path is in general not descending, so it can, for example, cross an even number of influential edges, and then the function will have the same value on the two endpoints of this path. This prevents one from immediately applying the techniques from [24, 23].

We overcome this difficulty by introducing our main conceptual contribution: the *descending-ascending view* of noise sensitivity. In the process above, instead of going through all the indices in an arbitrary order, we first go through the indices i for which $x_i = 1$ and only then through the ones for which $x_i = 0$. This forms a path between x and z that has first a descending component and then an ascending component. Although this random walk is more amenable to an analysis using the path-based techniques of [24, 23], there are still non-trivial sampling questions involved in the design and analysis of our algorithm.

An immediate corollary of our result is a query complexity upper bound on estimating the gap between the noise stability of a Boolean function and one. The noise stability of a Boolean function f depends on a parameter ρ and is denoted by $\text{Stab}_\rho[f]$ (for more information about noise stability, see [21]). One way $\text{Stab}_\rho[f]$ can be defined is as the unique quantity satisfying the functional relation $\frac{1}{2}(1-\text{Stab}_{1-2\delta}[f]) = NS_\delta[f]$ for all δ. This implies that by obtaining an approximation for $NS_\delta[f]$, one also achieves an approximation for $1-\text{Stab}_{1-2\delta}[f]$.

1.1 Results

Our main algorithmic result is the following:

▶ **Theorem 1.** *Let δ be a parameter satisfying:*

$$\frac{1}{n} \le \delta \le \frac{1}{\sqrt{n}\log^{1.5} n}$$

Suppose, $f : \{0,1\}^n \to \{0,1\}$ is a monotone function and $NS_\delta[f] \ge \frac{1}{n^C}$ for some constant C.

Then, there is an algorithm that outputs an approximation to $NS_\delta[f]$ to within a multiplicative factor of $(1 \pm \epsilon)$, with success probability at least $2/3$. In expectation, the algorithm makes $O\left(\frac{\sqrt{n}\delta \log^{1.5} n}{NS_\delta[f]\epsilon^3}\right)$ queries to the function. Additionally, it runs in time polynomial in n.

[2] A path is *descending* if each subsequent vertex in it is dominated by all the previous ones in the natural partial order on the Boolean cube.

Note that computing noise-sensitivity using standard sampling[3] requires $O\left(\frac{1}{NS_\delta[f]\epsilon^2}\right)$ samples. Therefore, for a constant ϵ, we have the most dramatic improvement if $\delta = \frac{1}{n}$, in which case, ignoring constant and logarithmic factors, our algorithm outperforms standard sampling by a factor of $\sqrt{n}$.

As in [24], our algorithm requires that the noise sensitivity of the input function f is larger than a specific threshold $1/n^C$. Our algorithm is not sensitive to the value of C as long as it is a constant, and we think of $1/n^C$ as a rough initial lower bound known in advance.

We next give lower bounds for approximating three different parameters of monotone Boolean functions: the bias, the influence and the noise sensitivity. A priori, it is not clear what kind of lower bounds one could hope for. Indeed, determining whether a given function is the all-zeros function requires $\Omega(2^n)$ queries in the general function setting, but only 1 query (of the all-ones input), if the function is promised to be monotone. Nevertheless, we show that such a dramatic improvement for approximating these quantities is not possible.

For monotone functions, we are not aware of previous lower bounds on approximating the bias or noise sensitivity. Our lower bound on approximating influence is not comparable to the lower bounds in [24, 23], as we will elaborate shortly.

We now state our lower bound for approximating the noise sensitivity. Here and everywhere else, to "reliably distinguish" means to distinguish with probability at least 2/3.

▶ **Theorem 2.** *For all constants C_1 and C_2 satisfying $C_1 - 1 > C_2 \geq 0$, for an infinite number of values of n the following is true: For all δ satisfying $1/n \leq \delta \leq 1/2$, given a monotone function $f : \{0,1\}^n \to \{0,1\}$, one needs at least $\Omega\left(\frac{n^{C_2}}{e^{\sqrt{C_1 \log n/2}}}\right)$ queries to reliably distinguish between the following two cases: (i) f has noise sensitivity between $\Omega(1/n^{C_1+1})$ and $O(1/n^{C_1})$ and (ii) f has noise sensitivity larger than $\Omega(\min(1, \delta\sqrt{n})/n^{C_2})$.*

▶ Remark 3. For any positive constant ξ, we have that $e^{\sqrt{C_1 \log n/2}} \leq n^{\xi}$.

▶ Remark 4. The range of the parameter δ can be divided into two regions of interest. In the region $1/n \leq \delta \leq 1/(\sqrt{n}\log n)$, the algorithm from Theorem 1 can distinguish the two cases above with only $\tilde{O}(n^{C_2})$ queries. Therefore its query complexity is optimal up to a factor of $\tilde{O}(e^{\sqrt{C_1 \log n/2}})$. Similarly, in the region $1/(\sqrt{n}\log n) \leq \delta \leq 1/2$, the standard sampling algorithm can distinguish the two distributions above with only $\tilde{O}(n^{C_2})$ queries. Therefore in this region of interest, standard sampling is optimal up to a factor of $\tilde{O}(e^{\sqrt{C_1 \log n/2}})$.

We define the *bias* of a Boolean function as $B[f] \overset{\text{def}}{=} \Pr[f(x) = 1]$, where x is chosen uniformly at random from $\{0,1\}^n$. It is arguably the most basic property of a Boolean function, so we consider the question of how quickly it can be approximated for monotone functions. To approximate the bias up to a multiplicative factor of $(1 \pm \epsilon)$ using standard sampling, one needs $O(1/(B[f]\epsilon^2))$ queries. We obtain a lower bound for this task similar to the previous theorem:

▶ **Theorem 5.** *For all constants C_1 and C_2 satisfying $C_1 - 1 > C_2 \geq 0$, for an infinite number of values of n the following is true: Given a monotone function $f : \{0,1\}^n \to \{0,1\}$, one needs at least $\Omega\left(\frac{n^{C_2}}{e^{\sqrt{C_1 \log n/2}}}\right)$ queries to reliably distinguish between the following two cases: (i) f has bias of $\Theta(1/n^{C_1})$ (ii) f has bias larger than $\Omega(1/n^{C_2})$.*

[3] Standard sampling refers to the algorithm that picks $O\left(\frac{1}{NS_\delta[f]\epsilon^2}\right)$ pairs x and z as in the definition of noise sensitivity and computes the fraction of pairs for which $f(x) \neq f(z)$.

Finally, we prove a lower bound for approximating influence:

▶ **Theorem 6.** *For all constants C_1 and C_2 satisfying $C_1 - 1 > C_2 \geq 0$, for an infinite number of values of n the following is true: Given a monotone function $f : \{0,1\}^n \to \{0,1\}$, one needs at least $\Omega\left(\frac{n^{C_2}}{e^{\sqrt{C_1 \log n/2}}}\right)$ queries to reliably distinguish between the following two cases: (i) f has influence between $\Omega(1/n^{C_1})$ and $O(n/n^{C_1})$ (ii) f has influence larger than $\Omega(\sqrt{n}/n^{C_2})$.*

This gives us a new sense in which the algorithm family in [24, 23] is close to optimal, because for a function f with influence $\Omega(\sqrt{n}/n^{C_2})$ this algorithm makes $\tilde{O}(n^{C_2})$ queries to estimate the influence up to any constant factor.

Our lower bound is incomparable to the lower bound in [24], which makes the stronger requirement that $I[f] \geq \Omega(1)$, but gives a bound that is only a polylogarithmic factor smaller than the runtime of the algorithm in [24, 23]. There are many possibilities for algorithmic bounds that were compatible with the lower bound in [24, 23], but are eliminated with our lower bound. For instance, prior to this work, it was conceivable that an algorithm making as little as $O(\sqrt{n})$ queries could give a constant factor approximation to the influence of **any** monotone input function whatsoever. Our lower bound shows that not only is this impossible, no algorithm that makes $O(n^{C_2})$ queries for any constant C_2 can accomplish this either.

1.2 Algorithm overview

Here, we give the algorithm in Theorem 1 together with the subroutines it uses. Additionally, we give an informal overview of the proof of correctness and the analysis of running time and query complexity, which are presented in more detail in Section 3.

First of all, recall that $NS_\delta[f] = \Pr[f(x) \neq f(z)]$ by Equation 1. Using a standard pairing argument, we argue that $NS_\delta[f] = 2 \cdot \Pr[f(x) = 1 \wedge f(z) = 0]$. In other words, we can focus only on the case when the value of the function flips from one to zero.

We introduce the *descending-ascending view of noise sensitivity* (described more formally in Subsection 3.1), which, roughly speaking, views the noise process as decomposed into a first phase that operates only on the locations in x that are 1, and a second phase that operates only on the locations in x that are set to 0. Formally, we define the noise process D in Algorithm 1.

This process gives us a path from x to z that can be decomposed into two segments, such that the first part, P_1, descends in the hypercube, and the second part P_2 ascends in the hypercube.

Algorithm 1 Process D.

1. Pick x uniformly at random from $\{0,1\}^n$. Let S_0 be the set of indexes i for which $x_i = 0$, and conversely let S_1 be the rest of indexes.
2. **Phase 1:** go through all the indexes in S_1 in a random order, and flip each with probability δ. Form the descending path P_1 from all the intermediate results. Call the endpoint y.
3. **Phase 2:** start at y, and flip each index in S_0 with probability δ. As before, all the intermediate results form an ascending path P_2, which ends in z.
4. Output P_1, P_2, x, y and z.

Since f is monotone, for $f(x) = 1$ and $f(z) = 0$ to be the case, it is necessary, though not sufficient, that $f(x) = 1$ and $f(y) = 0$, which happens whenever P_1 hits an influential edge. Therefore we break the task of estimating the probability of $f(x) \neq f(z)$ into computing the product of:

- The probability that P_1 hits an influential edge, specifically, the probability that $f(x) = 1$ and $f(y) = 0$, which we refer to as p_A.
- The probability that P_2 does not hit any influential edge, given that P_1 hits an influential edge: specifically, the probability that given $f(x) = 1$ and $f(y) = 0$, it is the case that $f(z) = 0$. We refer to this probability as p_B.

The above informal definitions of p_A and p_B ignore some technical complications. Specifically, the impact of certain "bad events" is considered in our analysis. We redefine p_A and p_B precisely in Subsection 3.2.1.

To define those bad events, we use the following two values, which we reference in our algorithms: t_1 and t_2. Informally, t_1 and t_2 have the following meaning. A typical vertex x of the hypercube has Hamming weight $L(x)$ between $n/2 - t_1$ and $n/2 + t_1$. A typical Phase 1 path from process D will have length at most t_2. To achieve this, we assign $t_1 \overset{\mathsf{def}}{=} \eta_1 \sqrt{n \log n}$ and $t_2 \overset{\mathsf{def}}{=} n\delta(1 + 3\eta_2 \log n)$, where η_1 and η_2 are certain constants.

We also define M to be the set of edges $e = (v_1, v_2)$, for which both $L(v_1)$ and $L(v_2)$ are between and $n/2 - t_1$ and $n/2 + t_1$. Most of the edges in the hypercube are in M, which is used by our algorithm and the run-time analysis.

Our analysis requires that only $\delta \leq 1/(\sqrt{n} \log^{1.5} n)$ as in the statement of Theorem 1, however the utility of the ascending-descending view can be most clearly motivated when $\delta \leq 1/(\sqrt{n} \log^2 n)$. Specifically, given that $\delta \leq 1/(\sqrt{n} \log^2 n)$, it is the case that t_2 will be shorter than $O(\sqrt{n}/\log n)$. Therefore, typically, the path P_1 is also shorter than $O(\sqrt{n}/\log n)$. Similar short descending paths on the hypercube have been studied before: In [24], paths of such lengths were used to estimate the number of influential edges by analyzing the probability that a path would hit such an edge. One useful insight given by [24] is that the probability of hitting almost every single influential edge is roughly the same.

However, the results in [24] cannot be immediately applied to analyze P_1, because (i) P_1 does not have a fixed length, but rather its lengths form a probability distribution, (ii) this probability distribution also depends on the starting point x of P_1. We build upon the techniques in [24] to overcome these difficulties, and prove that again, roughly speaking, for almost every single influential edge, the probability that P_1 hits it depends very little on the location of the edge, and our proof also computes this probability. This allows us to prove that $p_A \approx \delta I[f]/2$. Then, using the algorithm in [24] to estimate $I[f]$, we thereby estimate p_A.

Regarding p_B, we estimate it by approximately sampling paths P_1 and P_2 that would arise from process D, conditioned on that P_1 hits an influential edge. To that end, we first sample an influential edge e that P_1 hits. Since P_1 hits almost every single influential edge with roughly the same probability, we do it by sampling e approximately uniformly from among influential edges. For the latter task, we build upon the result in [24] as follows: As we have already mentioned, the algorithm in [24] samples descending paths of a fixed length to estimate the influence. For those paths that start at an x for which $f(x) = 1$ and end at a z for which $f(z) = 0$, we add a binary search step in order to locate the influential edge e that was hit by the path.

Thus, we have the following algorithm $\mathcal{A}$ (see Algorithm 2), which takes oracle access to a function f and an approximation parameter ϵ as input. In the case of success, it outputs an influential edge that is roughly uniformly distributed.

Algorithm 2 Algorithm $\mathcal{A}$ (given oracle access to a monotone function $f : \{0,1\}^n \to \{0,1\}$ and a parameter ϵ).

```
1. Assign $w = \frac{\epsilon}{3100\eta_1}\sqrt{\frac{n}{\log n}}$
2. Pick $x$ uniformly at random from $\{0,1\}^n$.
3. Perform a descending walk $P_1$ downwards in the hypercube starting at $x$. Stop at a vertex $y$ either after $w$ steps, or if you hit the all-zeros vertex. Query the value of $f$ only at the endpoints $x$ and $y$ of this path.
4. If $f(x) = f(y)$ output FAIL.
5. If $f(x) \neq f(y)$ perform a binary search on the path $P_1$ and find an influential edge $e_{inf}$.
6. If $e_{inf} \in M$ return $e_{inf}$. Otherwise output FAIL.
```

Finally, once we have obtained a roughly uniformly random influential edge e, we sample a path P_1 from among those that hit it. An obvious way to try to quickly sample such a path is to perform two random walks of lengths w_1 and w_2 in opposite directions from the endpoints of the edge, and then concatenate them into one path. However, to do this, one needs to somehow sample the lengths w_1 and w_2. This problem is not trivial, since longer descending paths are more likely to hit an influential edge, which biases the distribution of the path lengths towards longer ones.

To generate w_1 and w_2 according to the proper distribution, we first sample a path P_1 hitting any edge at the same *layer*[4] Λ_e as e. We accomplish this by designing an algorithm that uses rejection sampling. The algorithm samples short descending paths from some conveniently chosen distribution, until it gets a path hitting the desired layer.

We now describe the algorithm in more detail. Recall that we use $L(x)$ to denote the Hamming weight (which we also call the *level*) of x, which equals the number of indices i on which $x_i = 1$, and we use the symbol Λ_e to denote the whole *layer* of edges that have the same endpoint levels as e. The algorithm $\mathcal{W}$ described in Algorithm 3 takes an influential edge e as an input and samples the lengths w_1 and w_2.

Algorithm 3 Algorithm $\mathcal{W}$ (given an edge $e \overset{\text{def}}{=} (v_1, v_2)$ so $v_2 \preceq v_1$).

```
1. Pick an integer $l$ uniformly at random among the integers in $[L(v_1), L(v_1) + t_2 - 1]$. Pick a vertex $x$ randomly at level $l$.
2. As in phase 1 of the noise sensitivity process, traverse in random order through the indices of $x$ and for each index that equals to one, flip it with probability $\delta$. The intermediate results form a path $P_1$, and we call its endpoint $y$.
3. If $P_1$ does not intersect $\Lambda_e$ go to step 1.
4. Otherwise, output $w_1 = L(x) - L(v_1)$ and $w_2 = L(v_2) - L(y)$.
```

Recall that t_2 has a technical role and is defined to be equal $n\delta(1 + 3\eta_2 \log n)$, where η_2 is a certain constant. t_2 is chosen to be long enough that it is longer than most paths P_1, but short enough to make the sampling in $\mathcal{W}$ efficient. Since the algorithm involves short descending paths, we analyze this algorithm building upon the techniques we used to approximate p_A.

[4] We say that edges e_1 and e_2 are on the same layer if and only if their endpoints have the same Hamming weights. We denote the layer an edge e belongs to as Λ_e.

After obtaining a random path going through the same layer as e, we show how to transform it, using the symmetries of the hypercube, into a a random path P_1 going through e itself. Additionally, given the endpoint of P_1, we sample the path P_2 just as in the process D.

Formally, the algorithm $\mathcal{B}$ (see Algorithm 4) takes an influential edge e and returns a descending path P_1 that goes through e and an adjacent ascending path P_2, together with the endpoints of these paths.

Algorithm 4 Algorithm $\mathcal{B}$ (given an influential edge $e \overset{\text{def}}{=} (v_1, v_2)$ so $v_2 \preceq v_1$).

```
1. Use $\mathcal{W}(e)$ to sample $w_1$ and $w_2$.
2. Perform an ascending random walk of length $w_1$ starting at $v_1$ and call its endpoint $x$.
   Similarly, perform a descending random walk starting at $v_2$ of length $w_2$, call its endpoint $y$.
3. Define $P_1$ as the descending path that results between $x$ and $y$ by concatenating the two
   paths from above, oriented appropriately, and edge $e$.
4. Define $P_2$ just as in phase 2 of our process starting at $y$. Consider in random order all
   the zero indices $y$ has in common with $x$ and flip each with probability $\delta$.
5. Return $P_1$ ,$P_2$, $x$, $y$ and $z$.
```

We then use sampling to estimate which fraction of the paths P_2 continuing these P_1 paths does not hit an influential edge. This allows us to estimate p_B, which, combined with our estimate for p_A, gives us an approximation for $NS_\delta[f]$.

Formally, we put all the previously defined subroutines together into the randomized Algorithm 5 that takes oracle access to a function f together with an approximation parameter ϵ and outputs an approximation to $NS_\delta[f]$:

Algorithm 5 Algorithm for estimating noise sensitivity. (given oracle access to a monotone function $f : \{0,1\}^n \to \{0,1\}$, and a parameter ϵ).

```
1. Using the algorithm from [24] as described in Theorem 14, compute an approximation to
   the influence of $f$ to within a multiplicative factor of $(1 \pm \epsilon/33)$. This gives us $\tilde{I}$.
2. Compute $\tilde{p}_A := \delta\tilde{I}/2$.
3. Initialize $\alpha := 0$ and $\beta := 0$. Repeat the following until $\alpha = \frac{768 \ln 200}{\epsilon^2}$.
   - Use algorithm $\mathcal{A}$ from Lemma 20 repeatedly to successfully sample an edge $e$.
   - From Lemma 25 use the algorithm $\mathcal{B}$, giving it $e$ as input, and sample $P_1$, $P_2$, $x$, $y$
     and $z$.
   - If it is the case that $f(x) = 1$ and $f(z) = 0$, then $\alpha := \alpha + 1$.
   - $\beta := \beta + 1$.
4. Set $\tilde{p}_B = \frac{\alpha}{\beta}$.
5. Return $2\tilde{p}_A\tilde{p}_B$.
```

1.3 Lower bound techniques

We use the same technique to lower bound the query complexity of approximating any of the following three quantities: the noise sensitivity, influence and bias.

For concreteness, let us first focus on approximating the bias. Recall that one can distinguish the case where the bias is 0 from the bias being $1/2^n$ using a single query. Nevertheless, we show that for the most part, no algorithm for estimating the bias can do much better than the random sampling approach.

We construct two probability distributions D_1^B and D_2^B that are relatively hard to distinguish but have drastically different biases. To create them, we fix some threshold l_0 and then construct a special monotone function F^B, which has the following two properties: (1) It has a high bias. (2) It equals to one on only a relatively small fraction of points on the level l_0. We refer to functions satisfying (2) as "thin" functions. We will explain later how to obtain such a function F^B. We pick a function from D_2^B by taking F^B, randomly permuting the indices of its input, and finally "truncating" it by setting it to one on all values of x, which have Hamming weight greater than l_0.

We form D_1^B even more simply. We take the all-zeros function and truncate it at the same threshold l_0. The threshold l_0 is chosen in a way that this function in D_1^B has a sufficiently small bias. Thus D_1^B consists of only a single function.

The purpose of truncation is to prevent a distinguisher from gaining information by accessing the values of the function on the high-level vertices of the hypercube. Indeed, if there was no truncation, one could tell whether they have access to the all-zeros function by simply querying it on the all-ones input. Since F^B is monotone, if it equals to one on at least one input, then it has to equal one on the all-ones input.

The proof has two main lemmas: The first one is computational and says that if F^B is "thin" then D_1^B and D_2^B are hard to reliably distinguish. To prove the first lemma, we show that one could transform any adaptive algorithm for distinguishing D_1^B from D_2^B into an algorithm that is just as effective, is non-adaptive and queries points only on the layer l_0.

To show this, we observe that, because of truncation, distinguishing a function in D_2^B from a function in D_1^B is in a certain sense equivalent to finding a point with level at most l_0 on which the given function evaluates to one. We argue that for this setting, adaptivity does not help. Additionally, if $x \preceq y$ and both of them have levels at most l_0 then, since f is monotone, $f(x) = 1$ implies that $f(y) = 1$ (but not necessarily the other way around). Therefore, for finding a point on which the function evaluates to one, it is never more useful to query x instead of y.

Once we prove that no algorithm can do better than a non-adaptive algorithm that only queries points on the level l_0, we use a simple union bound to show that any such algorithm cannot be very effective for distinguishing our distributions.

Finally, to construct F^B, we need to show that there exist functions that are "thin" and simultaneously have a high bias. This is a purely combinatorial question and is proven in our second main lemma. We build upon Talagrand random functions that were first introduced in [25]. In [17] it was shown that they are very sensitive to noise, which was applied for property testing lower bounds [2]. A Talagrand random DNF consists of $2^{\sqrt{n}}$ clauses of $\sqrt{n}$ indices chosen randomly with replacement. We modify this construction by picking the indices without replacement and generalize it by picking $2^{\sqrt{n}}/n^{C_2}$ clauses, where C_2 is a non-negative constant. We show that these functions are "thin", so they are appropriate for our lower bound technique.

"Thinness" allows us to conclude that D_1^B and D_2^B are hard to distinguish from each other. We then prove that they have drastically different biases. We do the latter by employing the probabilistic method and showing that in expectation our random function has a large enough bias. We handle influence and noise sensitivity analogously, specifically by showing that that as we pick fewer clauses, the expected influence and noise sensitivity decrease proportionally. We prove this by dividing the points, where one of these random functions equals to one, into two regions: (i) the region where only one clause is true and (ii) a region where more than one clause is true. Roughly speaking, we show that the contribution from the points in (i) is sufficient to obtain a good lower bound on the influence and noise sensitivity.

1.4 Possibilities of improvement?

In [23] (which is the journal version of [24]), it was shown that using the chain decomposition of the hypercube, one can improve the run-time of the algorithm to $O\left(\frac{\sqrt{n}}{\epsilon^2 I[f]}\right)$ and also improve the required lower bound on $I[f]$ to be $I[f] \geq \exp(-c_1\epsilon^2 n + c_2 \log(n/\epsilon))$ for some constant c_1 and c_2 (it was $I[f] \geq 1/n^C$ for any constant C in [24]). Additionally, the algorithm itself was considerably simplified.

A hope is that techniques based on the chain decomposition could help improve the algorithm in Theorem 1. However, it is not clear how to generalize our approach to use these techniques, since the ascending-descending view is a natural way to express noise sensitivity in terms of random walks, and it is not obvious whether one can replace these walks with chains of the hypercube.

2 Preliminaries

2.1 Definitions

2.1.1 Fundamental definitions and lemmas pertaining to the hypercube

▶ **Definition 7.** *We refer to the poset over $\{0,1\}^n$ as the **n-dimensional hypercube**, viewing the domain as vertices of a graph, in which two vertices are connected by an edge if and only if the corresponding elements of $\{0,1\}^n$ differ in precisely one index. For $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ in $\{0,1\}^n$, we say that $x \preceq y$ if and only if for all i in $[n]$ it is the case that $x_i \leq y_i$.*

▶ **Definition 8.** *The **level of a vertex** x on the hypercube is the hamming weight of x, or in other words number of 1-s in x. We denote it by $L(x)$.*

We define the set of edges that are in the same "layer" of the hypercube as a given edge:

▶ **Definition 9.** *For an arbitrary edge e suppose $e = (v_1, v_2)$ and $v_2 \preceq v_1$. We denote Λ_e to be the set of all edges $e' = (v_1', v_2')$, so that $L(v_1) = L(v_1')$ and $L(v_2) = L(v_2')$.*

The size of Λ_e is $L(v_1)\binom{n}{L(v_1)}$. The concept of Λ_e will be useful because we will deal with paths that are symmetric with respect to change of coordinates, and these have an equal probability of hitting any edge in Λ_e.

As we view the hypercube as a graph, we will often refer to paths on it. By referring to a path P we will, depending on the context, refer to its set of vertices or edges.

▶ **Definition 10.** *We call a path **descending** if for every pair of consecutive vertices v_i and v_{i+1}, it is the case that $v_{i+1} \prec v_i$. Conversely, if the opposite holds and $v_i \prec v_{i+1}$, we call a path **ascending**. We consider an empty path to be vacuously both ascending and descending. We define the length of a path to be the number of edges in it, and denote it by $|P|$. We say we **take a descending random walk of length w starting at x**, if we pick a uniformly random descending path of length w starting at x.*

Descending random walks over the hyper-cube were used in an essential way in [24] and were central for the recent advances in monotonicity testing algorithms [5, 6, 14].

▶ **Lemma 11** (Hypercube Continuity Lemma). *Suppose n is a sufficiently large positive integer, C_1 is a constant and we are given l_1 and l_2 satisfying $\frac{n}{2} - \sqrt{C_1 n \log(n)} \leq l_1 \leq l_2 \leq \frac{n}{2} + \sqrt{C_1 n \log(n)}$. If we denote $C_2 \overset{\mathsf{def}}{=} \frac{1}{10\sqrt{C_1}}$, then for any ξ satisfying $0 \leq \xi \leq 1$, if it is the case that $l_2 - l_1 \leq C_2 \xi \sqrt{\frac{n}{\log(n)}}$, then, for large enough n, it is the case that $1-\xi \leq \frac{\binom{n}{l_1}}{\binom{n}{l_2}} \leq 1+\xi$*

Proof. See the full version of the paper. ◀

2.1.2 Fundamental definitions pertaining to Boolean functions

▶ **Definition 12.** *Let δ be a parameter and let x be selected uniformly at random from $\{0,1\}^n$. Let $z \in \{0,1\}^n$ be defined as follows:*

$$z_i = \begin{cases} x_i & \textit{with probability } 1-\delta \\ 1-x_i & \textit{with probability } \delta \end{cases}$$

*We denote this distribution of x and z by T_δ. Then we define the **noise sensitivity** of f as $NS_\delta[f] \overset{\mathsf{def}}{=} \Pr_{(x,z) \in_R T_\delta}[f(x) \neq f(z)]$.*

▶ **Observation 13.** *For every pair of vertices a and b, the probability that for a pair x, z drawn from T_δ, it is the case that $(x,z) = (a,b)$, is equal to the probability that $(x,z) = (b,a)$.*

Therefore, $\Pr[f(x) = 0 \wedge f(z) = 1] = \Pr[f(x) = 1 \wedge f(z) = 0]$. Hence:

$$NS_\delta[f] = 2 \cdot \Pr[f(x) = 1 \wedge f(z) = 0]$$

2.1.3 Influence estimation

To estimate the influence, standard sampling would require $O\left(\frac{n}{I[f]\epsilon^2}\right)$ samples. However, from [24] we have:

▶ **Theorem 14.** *There is an algorithm that approximates $I[f]$ to within a multiplicative factor of $(1 \pm \epsilon)$ for a monotone $f : \{0,1\}^n \to \{0,1\}$. The algorithm requires that $I[f] \geq 1/n^{C'}$ for a constant C' that is given to the algorithm. It outputs a good approximation with probability at least 0.99 and in expectation requires $O\left(\frac{\sqrt{n}\log(n/\epsilon)}{I[f]\epsilon^3}\right)$ queries. Additionally, it runs in time polynomial in n.*

2.1.4 Bounds for the error parameter and the influence

The following observation allows us to assume that without loss of generality ϵ is not too small. A similar technique was also used in [24].

▶ **Observation 15.** *When $\epsilon < O(\sqrt{n}\delta \log^{1.5}(n))$ there is a simple algorithm that accomplishes the desired query complexity of $O\left(\frac{\sqrt{n}\delta \log^{1.5}(n)}{NS_\delta[f]\epsilon^3}\right)$. Namely, this can be done by the standard sampling algorithm that requires only $O\left(\frac{1}{NS_\delta[f]\epsilon^2}\right)$ samples. Thus, since we can handle the case when $\epsilon < O(\sqrt{n}\delta \log^{1.5}(n))$, we focus on the case when $\epsilon \geq H\sqrt{n}\delta \log^{1.5}(n) \geq H\frac{\log^{1.5} n}{\sqrt{n}}$, for any constant H.*

Additionally, throughout the paper whenever we need it, we will without loss of generality assume that ϵ is smaller than a sufficiently small positive constant.

We will also need a known lower bound on influence:

▶ **Observation 16.** *For any function $f : \{0,1\}^n \to \{0,1\}$ and $\delta \le 1/2$ it is the case that $NS_\delta[f] \le \delta I[f]$. Therefore it is the case that $I[f] \ge \frac{1}{n^C}$.*

A very similar statement is proved in [17] and for completeness we prove it in the full version of the paper.

3 An improved algorithm for small values of the noise parameter

In this section we give a more in-depth motivation for the analysis of our algorithm, together with the statements of the main lemmas. For all proofs, the reader is referred to the full version.

3.1 Descending-ascending framework

The descending-ascending process

It will be useful to view noise sensitivity in the context of the noise process D (see Algorithm 1 for the definition). By inspection, x and z are distributed identically in D as in T_δ. Therefore from Observation 13:

$$NS_\delta[f] = 2 \cdot \Pr_D[f(x) = 1 \wedge f(z) = 0]$$

▶ **Observation 17.** *Since the function is monotone, if $f(x) = 1$ and $f(z) = 0$, then it has to be that $f(y) = 0$.*

3.2 Review of algorithm

Roughly speaking, our algorithm will break the task of estimating $NS_\delta[f]$ into estimating the probabilities[5] $\Pr_D[f(x) = 1 \wedge f(y) = 0]$ and $\Pr_D[f(z) = 0 | f(x) = 1 \wedge f(y) = 0]$. To estimate the former, in Lemma 23 we will advantage of the fact that δ is small, so the path P_1 is typically short, and hence the same types of techniques can be applied as in the analysis of the influence estimation algorithm.

The situation here is different from that of influence estimation because (i) the length of the path is random (ii) this probability distribution of lengths depends on the starting vertex. However, we will prove there exists a value, call it p_1, so that most influential edges are hit with probability close to p_1. It depends on δ and $I[f]$, and we can estimate it quite efficiently by estimating $I[f]$.

In order to estimate $\Pr_D[f(z) = 0 | f(x) = 1 \wedge f(y) = 0]$ we will approximate the distribution D conditioned on $f(x) = 1 \wedge f(y) = 0$. To that end, we will first sample an influential edge e that P_1 goes through, and then among all the downwards paths going through e we will sample P_1 itself. The algorithm that samples an influential edge approximately uniformly is inspired by the algorithm that estimates influence.

3.2.1 Defining bad events and technical notation

In this section, we give the parameters that we use to determine the lengths of our walks, as well as the "middle" of the hypercube. Additionally, in this section we define some notation we use.

[5] In the precise analysis there will be some bad events to take care of. For the sake of simplicity, we do not talk about them right now.

Define the following values:

$$t_1 \overset{\text{def}}{=} \eta_1 \sqrt{n \log n} \qquad\qquad t_2 \overset{\text{def}}{=} n\delta(1 + 3\eta_2 \log n)$$

Here η_1 and η_2 are large enough constants. Taking $\eta_1 = \sqrt{C} + 4$ and $\eta_2 = C + 2$ is sufficient for our purposes (recall that we were promised that $NS_\delta[f] \geq 1/n^C$ for a constant C).

Informally, t_1 and t_2 have the following intuitive meaning. A typical vertex x of the hypercube has $L(x)$ between $n/2 - t_1$ and $n/2 + t_1$. A typical Phase 1 path from process D will have length at most t_2.

We define the "middle edges" M as the following set of edges:

$$M \overset{\text{def}}{=} \{e = (v_1, v_2) : \frac{n}{2} - t_1 \leq L(v_2) \leq L(v_1) \leq \frac{n}{2} + t_1\}$$

Denote by $\overline{M}$ the rest of the edges.

We define two bad events in context of D, such that when neither of these events happen, we can show that the output has certain properties. The first one happens roughly when P_1 (from x to y, as defined by Process D) is much longer than it should be in expectation, and the second one happens when P_1 crosses one of the edges that are too far from the middle of the hypercube, which could happen because P_1 is long or because of a starting point that is far from the middle. More specifically:

- E_1 happens when both of the following hold (i) P_1 crosses an edge $e \in E_I$ and (ii) denoting $e = (v_1, v_2)$, so that $v_2 \preceq v_1$, it is the case that $L(x) - L(v_1) \geq t_2$.
- E_2 happens when P_1 contains an edge in $E_I \cap \overline{M}$.

While defining E_1 we want two things from it. First of all, we want its probability to be upper-bounded easily. Secondly, we want it not to complicate the sampling of paths in Lemma 24. There exists a tension between these two requirements, and as a result the definition of E_1 is somewhat convoluted.

We will approximate the noise sensitivity as the product of the following two quantities:

$$p_A \overset{\text{def}}{=} \Pr_D[f(x) = 1 \wedge f(y) = 0 \wedge \overline{E_1} \wedge \overline{E_2}]$$

$$p_B \overset{\text{def}}{=} \Pr_D[f(z) = 0 | f(x) = 1 \wedge f(y) = 0 \wedge \overline{E_1} \wedge \overline{E_2}]$$

Ignoring the bad events, P_A is the probability that P_1 hits an influential edge, and P_B is the probability that given that P_1 hits an influential edge P_2 does not hit an influential edge. From Observation (17), if and only if these two things happen, it is the case that $f(x) = 1$ and $f(z) = 0$. From this fact and the laws of conditional probabilities we have:

$$\Pr_D[f(x) = 1 \wedge f(z) = 0 \wedge \overline{E_1} \wedge \overline{E_2}] = \Pr_D[f(x) = 1 \wedge f(y) = 0 \wedge f(z) = 0 \wedge \overline{E_1} \wedge \overline{E_2}] = p_A p_B \quad (2)$$

We can consider for every individual edge e in $M \cap E_I$ the probabilities:

$$p_e \overset{\text{def}}{=} \Pr_D[e \in P_1 \wedge \overline{E_1} \wedge \overline{E_2}]$$

$$q_e \overset{\text{def}}{=} \Pr_D[f(x) = 1 \wedge f(z) = 0 | e \in P_1 \wedge \overline{E_1} \wedge \overline{E_2}] = \Pr_D[f(z) = 0 | e \in P_1 \wedge \overline{E_1} \wedge \overline{E_2}]$$

The last equality is true because $e \in P_1$ already implies $f(x) = 1$. Informally and ignoring the bad events again, p_e is the probability that $f(x) = 1$ and $f(y) = 0$ **because** P_1 hits e and not some other influential edge. Similarly, q_e is the probability $f(x) = 1$ and $f(z) = 0$ given that P_1 hits specifically e.

Since f is monotone, P_1 can hit at most one influential edge. Therefore, the events of P_1 hitting different influential edges are disjoint. Using this, Equation (2) and the laws of conditional probabilities we can write:

$$p_A = \sum_{e \in E_I \cap M} p_e \tag{3}$$

Furthermore, the events that P_1 hits a given influential edge and then P_2 does not hit any are also disjoint for different influential edges. Therefore, analogous to the previous equation we can write:

$$p_A p_B = \Pr_D[(f(x) = 1) \wedge (f(z) = 0) \wedge \overline{E_1} \wedge \overline{E_2}] = \sum_{e \in E_I \cap M} p_e q_e \tag{4}$$

3.2.2 Bad events can be "ignored"

In the following, we will need to consider probability distributions in which bad events do not happen. For the most part, conditioning on the fact that bad events do not happen changes little in the calculations. In this subsection, we present two relatively simple lemmas that allow us to formalize these claims.

The following observation suggests that almost all influential edges are in M.

▶ **Observation 18.** *It is the case that:*

$$\left(1 - \frac{\epsilon}{310}\right) |E_I| \leq |M \cap E_I| \leq |E_I|$$

Proof. This is the case, because:

$$|\overline{M} \cap E_I| \leq |\overline{M}| \leq 2^n n \cdot 2 \exp(-2\eta_1^2 \log(n)) = \\ 2^{n-1} \cdot 4/n^{2\eta_1^2 - 1} \leq 2^{n-1} I[f]/n = |E_I|/n \leq \frac{\epsilon}{310} |E_I| \tag{5}$$

The second inequality is the Hoeffding bound, then we used Observations 16 and 15. ◀

We now assert that ignoring these bad events does not distort our estimate for $NS_\delta[f]$.

▶ **Lemma 19.** *It is the case that:*

$$p_A p_B \leq \frac{1}{2} NS_\delta[f] \leq \left(1 + \frac{\epsilon}{5}\right) p_A p_B$$

Proof. See the full version of the paper. ◀

3.3 Main lemmas

Having rigorously defined our technical language, now we can state our main algorithmic lemmas, together with their motivation and our approach to proving them. We refer the reader to the full version of the paper for the proofs of these lemmas, the proof of the Theorem 1 using these lemmas, as well as the derivation of the lower bounds in Theorems 5, 6 and 2.

The first two lemmas allow the estimation of the probability that a certain descending random walk hits an influential edge. As we mentioned in the introduction, except for the binary search step, the algorithm in Lemma 20 is similar to the algorithm in [24]. In principle, we could have carried out much of the analysis of the algorithm in Lemma 20 by referencing an equation in [24]. However, for subsequent lemmas, including Lemma 23, we build on the application of the Hypercube Continuity Lemma to the analysis of random walks on the hypercube. In the full version of the paper, we give a full analysis of the algorithm in Lemma 20, in order to demonstrate how the Hypercube Continuity Lemma (Lemma 11) can be used to analyze random walks on the hypercube, before handling the more complicated subsequent lemmas, including Lemma 23.

▶ **Lemma 20.** *There exists an algorithm $\mathcal{A}$ (see Algorithm 2) that samples edges from $M \cap E_I$ so that for every two edges e_1 and e_2 in $M \cap E_I$:*

$$\left(1 - \frac{\epsilon}{70}\right) \Pr_{e \in_R \mathcal{A}}[e = e_2] \le \Pr_{e \in_R \mathcal{A}}[e = e_1] \le \left(1 + \frac{\epsilon}{70}\right) \Pr_{e \in_R \mathcal{A}}[e = e_2]$$

The probability that the algorithm succeeds is at least $\frac{1}{O(\sqrt{n}\log^{1.5} n / I[f]\epsilon)}$. If it succeeds, the algorithm makes $O(\log n)$ queries, and if it fails, it makes only $O(1)$ queries. In either case, it runs in time polynomial in n.

▶ Remark 21. Through the standard repetition technique, the probability of error can be decreased to an arbitrarily small constant, at the cost of $O(\frac{\sqrt{n}\log^{1.5} n}{I[f]\epsilon})$ queries. Then, the run-time still stays polynomial in n, since $I[f] \ge 1/n^C$.

▶ Remark 22. The distribution $\mathcal{A}$ outputs is point-wise close to the uniform distribution over $M \cap E_I$. We will also obtain such approximations to other distributions in further lemmas. Note that this requirement is stronger than closeness in L_1 norm.

The following lemma, roughly speaking, shows that just as in previous lemma, the probability that P_1 in D hits an influential edge e does not depend on where exactly e is, as long as it is in $M \cap E_I$. The techniques we use are similar to the ones in the previous lemma and it follows the same outline. However here we encounter additional difficulties for two reasons: first of all, the length of P_1 is not fixed, but it is drawn from a probability distribution. Secondly, this probability distribution depends on the starting point of P_1.

▶ **Lemma 23.** *For any edge $e \in M \cap E_I$ it is the case that:*

$$\left(1 - \frac{\epsilon}{310}\right) \frac{\delta}{2^n} \le p_e \le \left(1 + \frac{\epsilon}{310}\right) \frac{\delta}{2^n}$$

While we will use Lemma 23 in order to estimate p_A, the next two lemmas are for estimating p_B. To that end, we will need to sample from a distribution of descending and ascending paths going through a given edge. Informally, the requirement on the distribution is that it should be close to the conditional distribution of such paths P_1 that would arise from process D, conditioned on going through e and satisfying $\bar{E}_1$ and $\bar{E}_2$.

A first approach to sampling such P_1 would be to take random walks in opposite directions from the endpoints of the edge e and then concatenate them together. This is in fact what we do. However, difficulty comes from determining the appropriate lengths of the walks for the following reason. If P_1 is longer, it is more likely to hit the influential edge e. This biases the distribution of the descending paths hitting e towards the longer descending paths. In order to accommodate for this fact we used the following two-step approach:

1. Sample only the levels of the starting and ending points of the path P_1. This is equivalent to sampling the length of the segment of P_1 before the edge e and after it. This requires careful use of rejection sampling together with the techniques we used to prove Lemmas 20 and 23. Roughly speaking, we use the fact that P_1 is distributed symmetrically with respect to the change of indices in order to reduce a question about the edge e to a question about the layer Λ_e. Then, we use the Lemma 11 to answer questions about random walks hitting a given layer. This is handled in Lemma 24.

2. Sample a path P_1 that has the given starting and ending levels and passes through an influential edge e. This part is relatively straightforward. We prove that all the paths satisfying these criteria are equally likely. We sample one of them randomly by performing two random walks in opposite directions starting at the endpoints of e. This all is handled in Lemma 25.

▶ **Lemma 24.** *There is an algorithm $\mathcal{W}$ (see Algorithm 3) that takes as input an edge $e = (v_1, v_2)$ in $M \cap E_I$, so that $v_2 \preceq v_1$, and samples two non-negative numbers w_1 and w_2, so that for any two non-negative w_1' and w_2':*

$$\begin{aligned}\left(1-\frac{\epsilon}{70}\right)&\Pr_{\mathcal{W}(e)}[(w_1=w_1')\wedge(w_2=w_2')]\\ &\leq \Pr_D[(L(x)-L(v_1)=w_1')\wedge(L(v_2)-L(y)=w_2')|(e\in P_1)\wedge\overline{E_1}\wedge\overline{E_2}]\\ &\qquad\leq\left(1+\frac{\epsilon}{70}\right)\Pr_{\mathcal{W}(e)}[(w_1=w_1')\wedge(w_2=w_2')] \qquad (6)\end{aligned}$$

The algorithm requires no queries to f and runs in time polynomial in n.

▶ **Lemma 25.** *There exists an algorithm $\mathcal{B}$ (see Algorithm 4) with the following properties. It takes as input an edge $e = (v_1, v_2)$ in $M \cap E_I$, so that $v_2 \preceq v_1$ and outputs paths P_1 and P_2 together with hypercube vertices x, y and z. It is the case that x is the starting vertex of P_1, y is both the starting vertex of P_2 and the last vertex of P_1, and z is the last vertex of P_2. Additionally, P_1 is descending and P_2 is ascending. Furthermore, for any pair of paths P_1' and P_2' we have:*

$$\begin{aligned}\Big|\Pr_{\mathcal{B}(e)}[(P_1=P_1')\wedge(P_2=P_2')]-\Pr_D[(P_1=P_1')\wedge(P_2=P_2')|(e\in P_1)\wedge\overline{E_1}\wedge\overline{E_2}]\Big|&\\ \leq\frac{\epsilon}{70}\Pr_{\mathcal{B}(e)}[(P_1=P_1')\wedge(P_2=P_2')]& \qquad (7)\end{aligned}$$

It requires no queries to the function and takes computation time polynomial in n to draw one sample.

In the full version of this paper, we analyze the query complexity and the run-time of the Algorithm 5, thus proving Theorem 1. This is shown to be a relatively straightforward application of the four main technical lemmas we presented and discussed in this section.

References

1 Maria-Florina Balcan, Eric Blais, Avrim Blum, and Liu Yang. Active property testing. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 21–30. IEEE, 2012.

2 Aleksandrs Belovs and Eric Blais. A polynomial lower bound for testing monotonicity. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 1021–1032. ACM, 2016.

3 Itai Benjamini, Gil Kalai, and Oded Schramm. Noise sensitivity of Boolean functions and applications to percolation. *Publications Mathématiques de l'Institut des Hautes Études Scientifiques*, 90(1):5–43, 1999.

4 Eric Blais, Ryan O'Donnell, and Karl Wimmer. Polynomial regression under arbitrary product distributions. *Machine learning*, 80(2-3):273–294, 2010.

5 Deeparnab Chakrabarty and Comandur Seshadhri. An o(n) Monotonicity Tester for Boolean Functions over the Hypercube. *SIAM Journal on Computing*, 45(2):461–472, 2016.

6 Xi Chen, Rocco A Servedio, and Li-Yang Tan. New algorithms and lower bounds for monotonicity testing. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 286–295. IEEE, 2014.

7 Mahdi Cheraghchi, Adam Klivans, Pravesh Kothari, and Homin K Lee. Submodular functions are noise stable. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1586–1592. Society for Industrial and Applied Mathematics, 2012.

8 Ilias Diakonikolas, Prasad Raghavendra, Rocco A. Servedio, and Li-Yang Tan. Average Sensitivity and Noise Sensitivity of Polynomial Threshold Functions. *SIAM J. Comput.*, 43(1):231–253, 2014. doi:10.1137/110855223.

9 Adam Tauman Kalai, Adam R Klivans, Yishay Mansour, and Rocco A Servedio. Agnostically learning halfspaces. *SIAM Journal on Computing*, 37(6):1777–1805, 2008.

10 Daniel M. Kane. The Gaussian Surface Area and Noise Sensitivity of Degree-d Polynomial Threshold Functions. *Computational Complexity*, 20(2):389–412, 2011. doi:10.1007/s00037-011-0012-6.

11 Daniel M. Kane. The average sensitivity of an intersection of half spaces. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 437–440, 2014. doi:10.1145/2591796.2591798.

12 Nathan Keller and Guy Kindler. Quantitative relation between noise sensitivity and influences. *Combinatorica*, 33(1):45–71, 2013.

13 Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable CSPs? *SIAM Journal on Computing*, 37(1):319–357, 2007.

14 Subhash Khot, Dor Minzer, and Muli Safra. On monotonicity testing and boolean isoperimetric type theorems. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 52–58. IEEE, 2015.

15 Adam R Klivans, Ryan O'Donnell, and Rocco A Servedio. Learning intersections and thresholds of halfspaces. *Journal of Computer and System Sciences*, 68(4):808–840, 2004.

16 Rajsekar Manokaran, Joseph Seffi Naor, Prasad Raghavendra, and Roy Schwartz. SDP gaps and UGC hardness for multiway cut, 0-extension, and metric labeling. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 11–20. ACM, 2008.

17 Elchanan Mossel and Ryan O'Donnell. On the noise sensitivity of monotone functions. *Random Structures & Algorithms*, 23(3):333–350, 2003.

18 Elchanan Mossel and Ryan O'Donnell. Coin flipping from a cosmic source: On error correction of truly random bits. *Random Structures & Algorithms*, 26(4):418–436, 2005.

19 Ryan O'Donnell. Hardness amplification within NP. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 751–760. ACM, 2002.

20 Ryan O'Donnell. *Computational applications of noise sensitivity.* PhD thesis, Massachusetts Institute of Technology, 2003.

21 Ryan O'Donnell. *Analysis of boolean functions.* Cambridge University Press, 2014.

22 Yuval Peres. Noise stability of weighted majority. *arXiv preprint*, 2004. arXiv:math/0412377.

23 Dana Ron, Ronitt Rubinfeld, Muli Safra, Alex Samorodnitsky, and Omri Weinstein. Approximating the Influence of Monotone Boolean Functions in $O(\sqrt{n})$ Query Complexity. *TOCT*, 4(4):11:1–11:12, 2012. doi:10.1145/2382559.2382562.

24 Dana Ron, Ronitt Rubinfeld, Muli Safra, and Omri Weinstein. Approximating the Influence of Monotone Boolean Functions in $O(\sqrt{n})$ Query Complexity. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 664–675. Springer, 2011.

25 Michel Talagrand. How Much Are Increasing Sets Positively Correlated? *Combinatorica*, 16(2):243–258, 1996.