# Higher Lower Bounds On Monotone Size

Danny Harnik* Ran Raz*

**Abstract**

We prove a lower bound of $2^{\Omega\left(\left(\frac{n}{\log n}\right)^{\frac{1}{3}}\right)}$ on the monotone size of an explicit function in monotone-$\mathcal{NP}$ (where $n$ is the number of input variables). This is higher than any previous lower bound on the monotone size of a function. The previous best being a lower bound of about $2^{\Omega(n^{\frac{1}{4}})}$ for Andreev's function, proved in [AlBo87].
Our lower bound is proved by the symmetric version of Razborov's method of approximations. However, we present this method in a new and simpler way: Rather than building approximator functions for all the gates in a circuit, we use a gate elimination argument that is based on a Monotone Switching Lemma. The bound applies for a family of functions, each defined by a construction of a small probability space of $c$-wise independent random variables.

## 1 Introduction

### 1.1 Background and Previous Work

A monotone function is one that can be computed by a monotone circuit i.e., a circuit with only AND and OR gates. The monotone size of a function is the smallest number of gates needed in a monotone circuit in order to calculate the function. The size of a function is used as an indication of the function's complexity – a function that can only be calculated by a large circuit is a hard function to compute. The use of circuit size as a complexity measure applies for general circuits as well as for monotone ones, but proving the hardness of functions in the general model has seen only little success. The model of monotone computation is a restricted model that has attracted much attention over the years. In this model, strong lower bounds were proved for both the size and the depth of explicit monotone functions. Although monotone complexity hasn't helped in proving bounds for the general model, it has proven valuable in other areas, e.g. in proof theory (see [Pu97] and other references there).

We concentrate on the well studied problem of proving lower bounds on the monotone size of an explicit monotone function.

In his breakthrough paper in 1985, Razborov [Ra85] proved an $n^{\Omega(\log n)}$ lower bound on the monotone-size of the Clique function. This was the first super-polynomial bound on the monotone-size of any explicit function. Moreover, the previous highest bound was only a $4n$ lower bound [Ti84], a fact that emphasizes the significance of Razborov's result. The new bound also introduced the method of approximations, later used also to prove other lower bounds for monotone-size as well as lower bounds in other models (e.g. bounded depth circuits). Shortly after Razborov's result, Andreev [An85] used the method of approximations to prove the first exponential lower bound for a somewhat unnatural function, known as Andreev's function. Later, Alon and Boppana [AlBo87]

*Department of Applied Mathematics and Computer Science, Weizmann Institute, Rehovot, 76100 Israel. Work supported by an American-Israeli BSF grants 95-00238, 98-00349. mail: {harnik,ranraz}@wisdom.weizmann.ac.il

improved both results, proving an exponential bound for the Clique function and improving the bound for Andreev's function. The latter, a lower bound of $2^{\Omega(n^{\frac{1}{4}}\sqrt{\log n})}$, was the highest lower bound on the monotone-size of a function to date. In this paper we prove a higher lower bound of $2^{\Omega\left(\left(\frac{n}{\log n}\right)^{\frac{1}{3}}\right)}$ on the monotone size of another explicit function.

In 1995, Haken [Ha95] introduced a seemingly new method called "Bottleneck Counting" for proving a lower bound on the monotone-size of a clique-like function. This method was shown to be equivalent to the method of approximations, but presented in a different way [AmMa96], [BeUl97], [SiTs97]. However, Haken's work inspired a new symmetric version of the method of approximations, that generates simpler proofs to the previous bounds for the Clique function and Andreev's function ([AmMa96],[BeUl97]). Another way to present Haken's method was derived by Jukna [Ju97], this time as a criterion for lower bounds.

All of the methods mentioned, as well as the one that we are about to present here are equivalent to the method of approximations. Unfortunately, this means that they cannot be used to prove super-linear lower bounds in the general (non-monotone) circuit complexity model, because of a result by Razborov [Ra89] that showed that the method of approximations can only yield linear bounds on general circuit size.

## 1.2 Monotone Complexity by Switching Lemma

In our work we choose to present the lower bound proof from yet a new angle. Our method is actually equivalent to the symmetric version of the method of approximations and relies on the beautiful presentation of this method by Berg and Ulfberg [BeUl97]. However, rather than following the lines of Razborov's approximations (where one builds an auxiliary approximator circuit for a given circuit) , we present our proof as a gate elimination process. The new presentation bears great resemblance to the well known lower bound proof for the bounded depth complexity of the XOR function, using the Switching Lemma (widely known as Hastad's Switching Lemma [Ha86], but there are also earlier versions of this proof method [FuSaSi81], [Aj83], [Ya85]).

The general idea of the proof is to eliminate gates from a given circuit, by use of a "Switching Lemma". A Switching Lemma is one that proves the ability to transform ("switch") between an AND of small ORs (called a d-CNF in this paper) and an OR of small ANDs (called a c-DNF). Our Switching Lemma states that such a switch can be done, introducing just few errors to the output of the circuit.

In the gate elimination process, we eliminate gates from the bottom of the circuit by switching them and then collapsing them into their parents (this proof method works for circuits with unbounded fan-in). Each gate eliminated may introduce only a small number of errors to the output of the circuit. At the end of the process we get a degenerate circuit. It is not hard to show that this degenerate circuit cannot compute the original function and must make many errors when compared to it. This means that many errors have been introduced during the process, but since only few were introduced at each gate, we conclude that the circuit must have been large to begin with and so the lower bound is proved.

The heart of the proof method is hence the Monotone Switching Lemma. It differs from Hastad's Switching Lemma in a few ways. Hastad's Lemma discusses non-monotone CNFs and DNFs while we consider monotone ones. On the other hand, Hastad's Lemma states that the switch can be done only after making a random restriction on the input variables (and hence it can be applied only a small number of times). The monotone Lemma does not restrict variables, but rather shows that the behavior of the circuit is damaged only on a very small number of inputs.

We note that all the versions of the method of approximations (not only the symmetric version) can be viewed as a gate elimination process.

### 1.3 The Function and The New Lower Bound

Our result was achieved in two steps: First we proved the lower bound for a random DNF (i.e., we choose the function by random choice of the function's minterms) [1]. Note that this does not prove a lower bound for an explicit function. The second step is to actually prove the lower bound for an explicit function. This is done by finding a function that behaves like a random DNF, but only in the aspects that are required for the proof.

We prove the lower bound for a family of explicit functions. Each function in this family is described by a construction of a **small probability space of $c$-wise independent random variables**. We use a small probability space of $k$ $c$-wise independent random variables, each variable uniformly distributed over the domain $\{1, ..., n\}$ ($n$ is the number of inputs for the function, $k$ is the maximal number of ones in a minterm of the function and $c$ is a parameter that we take to be about $n^{\frac{1}{3}}$). An optimal construction of such a space is of size $n^c$. An example of such an optimal construction is shown in [AlBaIt86] (see also appendix A). [2]

Loosely speaking, we define a monotone function using such a probability space in the following manner:
Each point in the probability space is used to create one minterm of the function. The whole probability space defines the whole set of the function's minterms (and this defines the function). Note that this is a well defined explicit function (no randomness is involved in creating the function, as all the points in the probability space are taken to define this one function). Moreover, if the construction is efficient (polynomial-time) then the function is clearly in $\mathcal{NP}$.

For each such function, we prove a lower bound of $2^{\Omega(c)}$. Since the probability space is of size $n^c$, the function has at most $2^{c \log n}$ minterms and can be calculated by a DNF circuit of this size. This means that the lower bound is almost tight.

A final note is that our bound of $2^{\Omega((\frac{n}{\log n})^{\frac{1}{3}})}$ seems to reach the limits of this proof method (the symmetric version of the method of approximations). It seems that even for a random function, no higher bound can be achieved by this method.

### 1.4 The Paper's Outline

Section 2 defines and discusses the basics of monotone computation. Section 3 describes the general proof method including the Monotone Switching Lemma and the gate elimination process. Section 4 introduces the function for which we prove the lower bound. In Section 5 we actually prove the lower bound on the function. Section 6 includes some concluding remarks about the method and the bound.

[1] We have learned that this step has been independently proved by Berg, Ulfberg and Wigderson [BeUlWi99].

[2] Another construction that can be used is of $n$ $c$-wise independent Bernoulli random variables with probability $\frac{k}{n}$ for success. Also, constructions of approximately $c$-wise independent probability spaces [EGLNV92] can be used, all yielding the same lower bound. For a more detailed discussion see section 6.1.

# 2 Preliminaries

## 2.1 Boolean Circuits

A **Boolean Function** is a map $f : \{0,1\}^n \rightarrow \{0,1\}$.
A **Boolean Circuit** $\mathcal{C}$ is a directed acyclic finite graph with three types of labeled vertices: inputs, gates and an output (We assume in our discussion that there is only one output). The inputs are the sources of the graph (vertices with in-degree 0) and are labeled with boolean variables from $x_1, ..., x_n$, or boolean constants. The gates have a positive in-degree, and are labeled with a boolean function AND, OR or NOT (denoted $\bigwedge$ , $\bigvee$ and $\neg$ respectively). The output is the sink in the graph (a vertex with out-degree 0).
We refer to the in-degree of a vertex as its *fan-in* and its out-degree as *fan-out*. A $\neg$ gate must have fan-in 1, while $\bigwedge$ and $\bigvee$ gates may have unbounded fan-in.

A boolean circuit represents a boolean function in a natural way. Given an assignment $x \in \{0,1\}^n$ to the $n$ input variables, $\mathcal{C}(x)$ denotes the value of the circuit's output. This value is calculated by setting the value of each gate according to the boolean function it is labeled with, and the values of the vertices with a directed edge to it. For instance, if a vertex is labeled $\bigwedge$ and the vertices with a directed edge to it have values $a$ and $b$ (this example is of a gate with fan-in 2), then the vertex has value $a \bigwedge b$.
It is easy to see that any boolean function can be calculated by some boolean circuit.
The size of a circuit is defined to be the number of gates in the circuit.
The size (circuit complexity) of a function is the size of the minimal circuit that calculates it.

## 2.2 Monotone Computation

In our discussion we deal only with a restricted model of computation, the model of monotone computation.
A **Monotone Function** is a boolean function where changing an input bit from 0 to 1 may only change the function's output from 0 to 1 (the value can remain the same but can't change from 1 to 0).
A **Monotone Circuit** $\mathcal{C}$ is a boolean circuit with only $\bigwedge$ and $\bigvee$ gates (without $\neg$ gates). We allow all gates to have arbitrarily large fan-in and fan-out.

It is easy to see that a monotone circuit calculates a monotone function. Moreover, any monotone function can be calculated by a monotone circuit.
The **Size of a Monotone Circuit** $\mathcal{C}$ is the number of gates in it. The **Monotone Size of a Monotone Function** $f$ is the size of the smallest monotone circuit that calculates $f$ and is denoted mon-size$(f)$ .

An **input** for a circuit or for a function is an assignment $x \in \{0,1\}^n$ to the input variables. For each such input the circuit calculates an output value. An input can also be represented as a set of all the variables that are assigned 1 in the assignment. Throughout the paper we will use either the regular or the set representation.
Sometimes, we choose to look at the set of variables that are assigned 0 in the assignment (rather than 1). This will be referred to as the zeros set of an input.

In order to better understand the notion of monotonicity, we define an ordering on the inputs:

**Definition 1 (An Order on the Inputs - The Hamming Order)** *Given two inputs $x, y \in \{0,1\}^n$ we say that $x$ is* **contained** *in $y$ (or $x$ is* **smaller** *than $y$) if for every $i \in \{1, ..., n\}$ we have $x_i \leq y_i$.*

We note that this is a partial order.

The set representation of inputs is very natural when discussing the Hamming order since the Hamming order is equivalent to the containment relation on sets.

Using the order on the inputs we can introduce an alternative definition for a monotone function: A function $f$ is monotone if for any two inputs $x, y \in \{0,1\}^n$ such that $x$ is smaller than $y$ we have $f(x) \leq f(y)$.

A **Minterm** of a monotone function $f$ is a minimal input $x$ (with respect to the Hamming order) such that $f(x) = 1$. In other words, an input $x$ is a minterm if $f(x) = 1$ and for every input $y$ such that $y$ is smaller than $x$ we have $f(y) = 0$.
A **Maxterm** of $f$ is a maximal input $x$ such that $f(x) = 0$.
Note: When viewing a minterm as a set, we look at the set of ones in the minterm. However when viewing a maxterm as a set, we always look at the zeros set of the maxterm. So actually a maxterm can be defined as an input $x$ with a minimal zeros set such that $f(x) = 0$.

A monotone function can be defined by the collection of all its minterms in the following manner: The function outputs the value 1 for all inputs that contain a minterm and 0 otherwise. With maxterms, the rule is symmetric: The function outputs 0 for every input whose zeros set contains a maxterm (the zeros set of the maxterm) and 1 otherwise.

A **CNF** is a conjunction of disjunctions of input variables (AND of ORs). It is a two leveled monotone circuit where the top level (the output) has one $\bigwedge$ gate and the edges directed to it are all from gates at the bottom level. The bottom level contains only $\bigvee$ gates with variables as inputs.
A **DNF** is a disjunction of conjunctions of input variables (OR of ANDs).
Note: In this paper, CNFs and DNFs are monotone functions. There are also non-monotone versions of CNFs and DNFs, where negations of variables are allowed as well.

Any monotone function can be written in DNF form where each of the conjunctions at the bottom level corresponds to a minterm. The variables in each conjunction are those of the corresponding minterm. Clearly the DNF circuit outputs 1 if and only if the input contains a minterm. Any monotone function can also be written in CNF form where each of the disjunctions corresponds to a maxterm. The variables in each disjunction are those of the corresponding maxterm (the maxterm's zeros set of course). Notice that this is the reason for viewing the maxterm by the set of it's zeros and not of it's ones.

We conclude that indeed any monotone function can be calculated by a monotone circuit, moreover this can be done by a CNF or a DNF circuit.

We note one more observation: When dealing with a circuit with unbounded fan-in (as we do in our paper), it can be assumed without loss of generality, that the gates in the circuit are alternating. This means that there are no two consecutive $\bigwedge$ gates or two consecutive $\bigvee$ gates in the circuit. If there are two such gates, then the inputs for the bottom gate out of the two, can be inserted directly to the parent (the top gate), thus disconnecting the two gates. Therefore, from here on, we assume that all the circuits are of alternating form.

# 3 The Proof Method

Our proof method is a new presentation of the symmetric version of the method of approximations. It is an equivalent proof method, but resembles the structure of the lower bound proof on the bounded depth complexity of the XOR function, using the Switching Lemma. Before presenting this method we need to define restricted forms of a CNF and of a DNF:

**Definition 2 (c-DNF and d-CNF)**
*A c-DNF is a DNF in which the fan-in of all the $\bigwedge$ gates (the conjunctions) is smaller than c.*
*A d-CNF is a CNF in which the fan-in of all the $\bigvee$ gates (the disjunctions) is smaller than d.*

Note: The fan-in at the output gate has no explicit bound in both cases, the bounds are only on the fan-in of the gates in the second level, next to the input variables.
We usually denote a d-CNF function by $e^C$ and a c-DNF function by $e^D$

A "Switching Lemma" is one that proves the ability to turn a d-CNF into a c-DNF and vice versa. Loosely speaking, our Switching Lemma will state the following:

- For every d-CNF $e^C$ there exists a c-DNF $e^D$, such that $e^D = e^C$ on most of the inputs.
- For every c-DNF $e^D$ there exists a d-CNF $e^C$, such that $e^C = e^D$ on most of the inputs.

Given such a Switching Lemma, the proof is by a gate elimination argument. In general, we do the following:
Given a circuit calculating the function $f$ (without loss of generality, the circuit has alternating $\bigwedge$ and $\bigvee$ gates) we go over the gates of the circuit in a bottom up fashion. At each stage we eliminate one gate that is either a c-DNF or a d-CNF by switching it into the opposite form (there must always be such a gate at the bottom of the circuit). This allows us to eliminate the gate by inserting it into its parent gates that were originally of opposite gate type but are now the same. When eliminating such a gate we only introduce a small number of errors in the output of the modified circuit, due to the Switching Lemma.
We continue doing so until we are left with a depth two circuit (that is, without loss of generality, a d-CNF). It is left to show that this d-CNF makes errors on most of the inputs when trying to calculate the function $f$. Since we have many errors in the final stage, and since only few were inserted at each stage of the process and all errors have to be accounted for, we conclude that there were many stages in the process – many gates to begin with. This proves that the original circuit was large.

We now describe the method in more detail:

## 3.1 The Monotone Switching Lemma

The heart of this method of proof is the Switching Lemma that follows. The Lemma presented here is a general one that will later be used (for specific parameters and function) to show that the probability of an error being introduced in the switching process is very small.
Recall that when we say a c-DNF and a d-CNF we mean a monotone c-DNF and a monotone d-CNF.

**Lemma 3.1 The Monotone Switching Lemma:**
*For every d-CNF $e^C$ there exists a c-DNF $e^D$, and a family $U$ of at most $d^c$ sets of c variables each, such that:*

*1. for any input $x$ we have $e^D(x) \leq e^C(x)$.*

*2. $e^D(x) < e^C(x)$ only on an input $x$ that contains one or more of the sets in $U$.*

*For every c-DNF $e^D$ there exists a d-CNF $e^C$, and a family $U$ of at most $c^d$ sets of $d$ variables each, such that:*

*1. for any input $x$ we have $e^C(x) \geq e^D(x)$.*

*2. $e^C(x) > e^D(x)$ only on an input $x$ whose zeros set contains one or more of the sets in $U$.*

Only the direction from $e^C$ to $e^D$ is proved here as the proof for the opposite direction is analogous. The proof is based on [BeUl97] (inspired by [Ha95]) and is given via an explicit construction of a c-DNF $e^D$ that fulfills the requirements of the Lemma.
**Proof:** Suppose that the d-CNF $e^C$ is a conjunction of the disjunctions $D_1, ..., D_s$ and we want to turn it into a disjunction of the conjunctions $C_1, ..., C_t$. We use the standard way of transforming a CNF into a DNF. $e^C$ outputs 1 if and only if the input contains at least one variable from each of the disjunctions $D_1, ..., D_s$. In general, we create a conjunction $C_i$ for every possible choice of one variable from each disjunction. So, at least one of the conjunctions is satisfied if and only if $e^C$ outputs 1 and hence the new DNF is equivalent to $e^C$.
This transformation might create a very large number of conjunctions (not a problem) and every conjunction might be of large size. Dropping every conjunction that has $c$ or more variables creates the legal c-DNF $e^D$.
When dropping a conjunction, the DNF circuit may change its output from 1 to 0 on an input that contains this conjunction. In particular, this means that for all inputs $x \in \{0,1\}^n$ we have $e^D(x) \leq e^C(x)$ (proving part 1).

We now look in more depth at the transformation of a CNF into a DNF:
The transformation's goal is to create all the choices of one variable from each disjunction in $D_1, ..., D_s$ (the disjunctions of the CNF) and take each such set of variables as a conjunction in the DNF. The process of finding these desired conjunctions **can be viewed as a construction of a tree**, where the edges are labeled with variables and each node is a set of variables, including all the variables on the path from the root to that node. The leaves of the tree are sets of variables that will be the desired conjunctions.

In general, the idea is to first satisfy $D_1$ (choose one variable from $D_1$) and go on to satisfy $D_2$ and so forth until we have a variable from each disjunction. Satisfying $D_1$ is done by adding an edge from the root to a new node for every variable in $D_1$ (the edge is labeled with this variable). Clearly every leaf in the tree, at this stage, contains one variable from $D_1$. We call this "opening" $D_1$ under the root. Now we open $D_2$ under each leaf of the current tree, so every new leaf contains a variable from $D_1$ and a variable from $D_2$. However, a leaf created by opening $D_1$ may already contain a variable from $D_2$. In such a case we don't add edges to this leaf since it already contains a variable from both $D_1$ and $D_2$.

Formally: the tree is built by iteratively going over all the disjunctions, and opening each one under all the leaves of the existing tree. $D_i$ is opened under a leaf by adding an edge from the leaf to a new node for every variable in $D_i$. However, If the leaf already contains a variable from $D_i$ then no edges are added (this rule avoids the possibility of a variable appearing twice on the same path from the root to a node) [3].

[3] There is another rule that is redundant in our proof, but is essential in the proofs of other lower bounds like the one for the CLIQUE function. It may be that $D_i$ is already satisfied for reasons other than a variable from $D_i$ already

After opening all of the disjunctions in $e^C$, we get a large tree where any possibility for a choice of at least one variable from each disjunction contains one or more leaves in the tree. Also, every leaf must satisfy each of the disjunctions. Therefore, taking the leaves of the tree as the conjunctions of the DNF gives a DNF equivalent to $e^C$.

Using the construction of the tree we can conclude the proof:
We have already established that an error ($e^D < e^C$) only happens when dropping conjunctions of size $c$ or more. Moreover, such an error may be introduced only on an input that contains such a dropped conjunction and in particular, only for inputs that contain the first $c$ variables in the dropped conjunction. But the first $c$ variables in a conjunction correspond to a node in the tree at distance exactly $c$ from the root (an ancestor of the leaf corresponding to the conjunction itself). So errors are introduced only on inputs that contain a node at distance $c$ from the root. Since the branching at every node in the tree is at most $d$ (the size of a disjunction) the tree cannot have more than $d^c$ nodes at distance exactly $c$ from the root. We have proved part 2 – an error may occur only on an input that contains one of at most $d^c$ conjunctions of length $c$. ■

### 3.2 The Gate Elimination Process

In this section we describe the process of modifying a circuit using the Switching Lemma. The process eliminates gates from the circuit one by one and in a bottom up fashion, until we are left with only a d-CNF circuit. In order to allow us to view the bottom gates of the circuit as CNFs and DNFs we add to the circuit a "dummy bottom level".

**The Dummy Bottom Level:** We add to the circuit a dummy bottom level (bottom level gates are gates that have only variables as inputs) such that this set of dummy gates, once added, becomes the bottom level of the circuit. This level is added at the beginning of the process and modified throughout the process. However, gates are eliminated only from the original circuit (and not from this bottom level). Moreover, we don't count the bottom level when talking about the size of the circuit (this allows us to insert an unbounded number of gates in the dummy bottom level when eliminating a gate from the original circuit). An important restriction is that the bottom level must contain only $\bigwedge$ gates with fan-in less than $c$ and $\bigvee$ gates with fan-in less than $d$.

**Adding the Dummy Bottom Level:** Given a circuit that calculates the function $f$, we add a new $\bigwedge$ gate and a new $\bigvee$ gate for each input variable, with this variable alone as an input. Each old gate that has a variable as an input will be connected to the corresponding new gate instead (an $\bigvee$ gate to the corresponding $\bigwedge$ gate and vice versa). So at the beginning of the process the bottom level consists only of the $2n$ new gates that all have fan-in 1 (that is in particular less than $c$ and $d$). The gates outside of the bottom level are exactly the gates of the original circuit.

**Eliminating One Gate:** At each stage we eliminate one gate from the circuit that is either a d-CNF or c-DNF. Such a gate must exist in the circuit at all times, it is one that has as inputs only gates from the dummy bottom level (the bottom level's gate have bounded fan-in, just as required from a d-CNF or c-DNF). Assume, without loss of generality, that we take a d-CNF $e^C$. We eliminate the gate by switching it into a c-DNF $e^D$. $e^D$ is an $\bigvee$ of conjunctions, so $e^D$'s conjunctions

appearing in the leaf. This can happen due to restrictions on the inputs (see section 3.3 on test inputs), causing a variable that hasn't already appeared to be satisfied. In such a case, the rule is to add only one new edge to the leaf, labeled by the variable that must be satisfied by this restriction.

can be inserted directly as inputs to each of the parents of the gate (they are all $\bigvee$ gates since it was assumed that the gates are alternating). In total we erase the $\bigwedge$ gate $e^C$, by switching it to an $\bigvee$ gate and collapsing it into its parents. In doing so we might have added many new gates ($e^D$'s conjunctions) but only to the bottom level (which we don't count). Therefore, the number of gates in the circuit has decreased by one.

We continue eliminating gates until there is only one gate in the circuit (excluding the bottom level). The circuit that remains must be either a d-CNF or a c-DNF and if it is a c-DNF it can be switched into a d-CNF. So the process is ended when the modified circuit is only a d-CNF.

### 3.3 Test Inputs

In order to capture the behavior of a specific function, we designate certain inputs to be "test" inputs. The rationale of this being that if a circuit calculates a function then it must in particular behave correctly on this set of test inputs. When discussing the success of a "switch" between a d-CNF and a c-DNF, we actually count the number of test inputs that give different outputs in the d-CNF and c-DNF (we say that errors were introduced on these test inputs).
We define two sets of test inputs:

- **Positive test inputs:** A set of inputs, on which the function $f$ outputs 1.
- **Negative test inputs:** A set of inputs, on which the function $f$ outputs 0.

A natural choice for the positive test inputs is the set of all the minterms. They all get value 1 and they capture the function's behavior. The maxterms are the natural choice for the negative test inputs and therefore, when viewing a negative test output as a set we look at it's zeros set.
However, the purpose of the test inputs technique is to look at sets that are easy to analyze, and this is not always true for the minterms and maxterms. So usually, a subset of the minterms or maxterms is taken instead. In our result it will be even simpler to take a positive test inputs set for which the function $f$ doesn't always output 1, but rather $f$ outputs a 1 only on a large enough fraction of the inputs in the set (and analogously for negative test inputs). In our result, for simplicity, we use a slightly more general technique, used before for choosing the negative test inputs in Andreev's function [An85]. Instead of looking at a set of inputs and counting the number of errors on such inputs, we take a **distribution** over the test inputs and ask what is the probability of error on such an input. We look at two distributions:

- **Positive test distribution:** A distribution on the inputs, such that with high probability $f$ outputs 1 on an input chosen according to this distribution.
- **Negative test distribution:** A distribution on the inputs, such that with high probability $f$ outputs 0 on an input chosen according to this distribution.

When we say positive test input, we mean an input taken according to the positive test distribution (negative test input is analogous). The approach of working with a set of test inputs is equivalent to taking a uniform distribution over this set.

We discuss the notion of an error on an input:

**Definition 3** *A circuit $e$ makes* **an error on a positive test input** *$x$ if it outputs $0$ on the input $x$ and also $f(x) = 1$.*

**Definition 4** *A circuit $e$ makes* **an error on a negative test input** *$x$ if it outputs $1$ on the input $x$ and also $f(x) = 0$.*

Next we define when errors are introduced in a switching process:

**Definition 5** *When switching from a c-DNF $e^C$ to a d-CNF $e^D$ we say that $e^D$* **introduces an error** *on an input (either a positive test input or a negative test input) if $e^C$ didn't make an error on the input but $e^D$ does.*

This means, for instance, that $e^D$ may introduce an error on the positive test input $x$ only if $e^C(x) = 1$ but $e^D(x) = 0$. [4]

The analogue definition for the other direction follows (switching $e^D$ to $e^C$):

**Definition 6** *When switching from a d-CNF $e^D$ to a c-DNF $e^C$ we say that $e^C$* **introduces an error** *on an input (either a positive test input or a negative test input) if $e^D$ didn't make an error on the input but $e^C$ does.*

### 3.4 The Pattern of The Proof

By now we have introduced all the technical lemmas and definitions needed to prove a lower bound on monotone size. In this section we show the general pattern of the proof, and sum up all the claims that we need in order to obtain the lower bound, with a general description of how their proofs should look. First we need to prove two specializations of the Switching Lemma (proved for particular choices of positive test distribution and negative test distribution):

- **Specialization of the Switching Lemma (first direction):** When switching a d-CNF $e^C$ into a c-DNF $e^D$, an error is introduced only on a positive test input and only with very low probability (denoted by $\beta$).
- **Specialization of the Switching Lemma (second direction):** When switching a c-DNF $e^D$ into a d-CNF $e^C$, an error is introduced only on a negative test input and only with very low probability (denoted by $\gamma$).

We describe the method of the proof for these lemmas:

**Proof of First Direction:** The upper bound on the probability is found using Lemma 3.1. An error cannot be introduced on a negative test input since $e^C \geq e^D$. The probability that an error is introduced on a positive test input is bounded by $d^c$ times the maximal probability that a positive test input contains a given set of variables of size exactly $c$ (where the maximum is taken over all the possible sets of size $c$). It is left to calculate the maximal probability that a positive test input contains a given set of $c$ variables.

**Proof of Second Direction:** The proof of this Lemma is analogous to that of Lemma B. We bound the maximal probability that the zeros set of a negative test input contains a given set of $d$ variables, and multiply this by $c^d$ to get the overall bound.

Also, a Base Case Lemma is needed, in order to analyze the result at the end of the gate elimination process:

[4] A switch might also cause a change from $e^D(x) = 0$ to $e^C(x) = 1$ (for positive test input $x$), but we don't count this as an error because such a change only improves the chances that the modified monotone circuit (including $e^D$) will output 1 on the positive test input $x$ (due to the circuit's monotonicity) and therefore the circuit won't make an error on $x$ if it didn't do so before the switch.

**Base Case Lemma:** A d-CNF $e^C$ either makes an error on a positive test input with high probability (denote this probability by $\alpha_p$), or makes an error on a negative test input with high probability ($\alpha_n$).

**Method of proof of the Base Case Lemma:** We check two possibilities: either $e^C$ is identically 1, and then outputs 1 for any negative test input, or $e^C$ gets the value 0 for some input, and hence contains at least one disjunction $D$. $e^C$ will output 1 on a positive test input only if it contains at least one of the variables in $D$ and this happens only with low probability (to be shown using the specifics of the positive test inputs).

Using the three lemmas we can prove a lower bound on the function's monotone size.

**Theorem 7**

$$\textit{mon-size}(f) \geq \min\left\{\frac{\alpha_p}{\beta}, \frac{\alpha_n}{\gamma}\right\}$$

**Proof:** Given a circuit that calculates the function $f$ and has $s$ $\bigwedge$ gates and $t$ $\bigvee$ gates, we exercise the gate eliminating process on it, until the modified circuit is a d-CNF $e^C$.
Due to the Base Case Lemma, either $e^C$ makes an error on a positive test input with probability $\alpha_p$ or $e^C$ makes an error on a negative test input with probability $\alpha_n$. Suppose that the first case occurs. An error on a positive test input must have been introduced when eliminating an $\bigwedge$ gate somewhere along the circuit. The probability that this occurs is bounded by $s \cdot \beta$ (due to the first direction of the Switching Lemma and a union bound). But this means that $s \cdot \beta \geq \alpha_p$ and $s \geq \frac{\alpha_p}{\beta}$.
On the other hand if $e^C$ makes an error on a negative test input with probability $\alpha_n$, then due to the second direction of the Switching Lemma and a union bound we get $t \cdot \gamma \geq \alpha_n$ and $t \geq \frac{\alpha_n}{\gamma}$.
So the circuit either has $\frac{\alpha_p}{\beta}$ $\bigwedge$ gates or has $\frac{\alpha_n}{\gamma}$ $\bigvee$ gates. ■

## 4 The Function

We now introduce the explicit monotone function $f$ for which we prove the new lower bound. The function is defined by producing the collection of its minterms. To build the set of minterms we use a small probability space of $c$-wise independent random variables. We take a small probability space of $k$ random variables, each uniformly distributed over the domain $\{1, ..., n\}$, such that every $c$ of them are independent random variables. There are many known explicit constructions of such probability spaces, requiring only $c \log n$ random bits (see an example in Appendix A). Each construction enables us to define a set of minterms and hence define an explicit function.
The set of minterms is taken as follows: Denote a point in the probability space by $y \in \{1, ..., n\}^k$. Each point $y = (y_1, ..., y_k)$ translates into one input: The values $y_1, ..., y_k$ are the indices of all the places that are assigned a 1 in the corresponding input. Altogether, each point $y$ in the probability spaces, gives an input $x \in \{0,1\}^n$ of Hamming weight at most $k$. The number of inputs is bounded by the size of the probability space $M = 2^{c \log n}$.
Note: The inputs in the set of minterms, might not all be actually minterms. If one input in the set contains another then it isn't a true minterm (it isn't minimal). However, for the convenience of notations, we shall regard all of the inputs in the set as "minterms".
This construction for minterms may yield minterms with very low Hamming weight (for instance if $y_1 = y_2 = ... = y_k$ then the Hamming weight is 1). Minterms with very few ones are contained in a large fraction of the inputs and may yield a function that almost always outputs 1. Such a

function will have a rather small number of maxterms and is therefore an easy function (can be calculated by a rather small CNF).

In order to avoid this, we take out all inputs that have less than $\frac{k}{2}$ ones and use the remaining set as the set of minterms. We show that a large fraction of the points in the probability space yield good minterms:

**Claim 4.2** *A fraction of less than $\frac{k}{n}$ of the points in the probability space translate to inputs that have no more than $\frac{k}{2}$ ones.*

**Proof:** We use the fact that the random variables $y_1, ..., y_k$ are pairwise independent (they are actually $c$-wise independent but pairwise is enough for the proof). Given a pair of random variables $y_i$ and $y_j$, $(i, j \in \{1, ..., k\}$ and $i \neq j)$, the probability that they coincide is $\Pr[y_i = y_j] = \frac{1}{n}$. The full probability space contains $M$ sample points, each with $\binom{k}{2}$ different pairs. So altogether, the number of coinciding pairs $y_i = y_j$ in all of the probability space is $\frac{1}{n}M\binom{k}{2}$.
Assume, for a contradiction, that $\frac{k}{n}$ of the points translate to inputs with no more than $\frac{k}{2}$ ones. No more than $\frac{k}{2}$ ones means that at least $\frac{k}{2}$ of the values of $y_1, ..., y_k$, coincide with others, giving us at least $\frac{k}{2}$ pairs $i, j \in \{1, ..., k\}$ $(i \neq j)$ for which $y_i = y_j$. Counting the coinciding pairs in these $\frac{k}{n}M$ points gives us at least $\frac{k}{n}M\frac{k}{2} = \frac{k^2}{2n}M$ pairs. But this is already more than the total number of coinciding pairs in the whole probability space as $\frac{k^2}{2n}M > \frac{1}{n}\binom{k}{2}M$, contradicting the assumption.
■

After the function was introduced we can define the positive and negative test distributions:

- Positive test input: We uniformly choose a point in the probability space and translate it into an input. Due to Claim 4.2 we see that with probability at least $1 - \frac{k}{n}$ we choose one of the minterms and therefore get value 1 in the function $f$.

- Negative test input: We choose an input randomly by independently picking each variable to be 0 with probability $\frac{l}{n}$ and 1 otherwise ($l$ is a parameter). We will show that with high probability (depending on the choice of the parameters), the function $f$ outputs 0 on such an input.

We remind the reader that when viewing a negative test input as a set we view it by its zeros set, like a maxterm.

**Claim 4.3** *With probability at least $(1 - Me^{-\frac{kl}{2n}})$ the function $f$ outputs 0 on a negative test input.*

**Proof:** Suppose $x \in \{0,1\}^n$ is a negative test input. $f(x) = 0$ if and only if $x$ (the set of $x$'s zeros) intersects each and every one of the minterms (the minterm's ones). Let us check the probability that $x$ does not intersect a single given minterm. Since a minterm has at least $\frac{k}{2}$ ones, the negative test input $x$ must not get a 0 at each of these indices. $x$ gets a 1 in each variable independently with probability $(1 - \frac{l}{n})$ so altogether:

$$\Pr_x[x \text{ does not intersect a specific minterm}] \leq \left(1 - \frac{l}{n}\right)^{\frac{k}{2}} \leq e^{-\frac{kl}{2n}}$$

There are no more than $M$ different minterms. We look for the probability that $x$ does not intersect at least one of them (this equals the probability that $f(x) = 1$). Using a union bound we get:

$$\Pr_x[f(x) = 1] = \Pr_x[x \text{ does not intersect at least one minterm}] \leq Me^{-\frac{kl}{2n}}$$

■

At this point, we introduce our choices for the parameters $c$, $d$, $k$ and $l$. However, for convenience, we will continue to use them as parameters when proving the lower bound. We choose the following values for the parameters (for simplicity assume that these values are all integers):

$$k = l = n^{\frac{2}{3}} \log^{\frac{1}{3}} n$$

and

$$c = d = \frac{1}{2} \left( \frac{n}{\log n} \right)^{\frac{1}{3}}$$

Recall that $M = 2^{c \log n}$, so due to Claim 4.3, the probability that $f$ outputs 0 on a negative test input is at least:

$$1 - 2^{c \log n} e^{-\frac{kl}{2n}} = 1 - 2^{\frac{1}{2} n^{\frac{1}{3}} \log^{\frac{2}{3}} n} e^{-\frac{1}{2} n^{\frac{1}{3}} \log^{\frac{2}{3}} n} = 1 - \left( \sqrt{\frac{2}{e}} \right)^{n^{\frac{1}{3}} \log^{\frac{2}{3}} n}$$

In particular, for large enough $n$, it is more than any constant probability (we take $\frac{3}{4}$ as our constant).
The probability that $f$ outputs 1 on a positive test input is at least $1 - \frac{k}{n} = 1 - n^{-\frac{1}{3}} \log^{\frac{1}{3}} n$ and this is also, for large enough $n$, more than any constant probability (again, take $\frac{3}{4}$ as the constant).

Altogether, with probability at least $\frac{3}{4}$, $f$ outputs 1 on a positive test input and with probability at least $\frac{3}{4}$, $f$ outputs 0 on a negative test input.

# 5 The Proof

**Lemma 5.4 (Base Case Lemma:)** *A d-CNF $e^C$ either makes an error on a negative test input with probability at least $\frac{3}{4}$, or $e^C$ makes an error on a positive test input with probability at least $\frac{1}{4}$.*

**Proof:** If $e^C \equiv 1$ then, in particular, it outputs 1 on any negative test input. By Claim 4.3 we saw that with probability at least $\frac{3}{4}$, the function $f$ outputs 0 on a negative test input $x$. Altogether, $e^C$ makes an error on $x$ with probability at least $\frac{3}{4}$.
If $e^C \not\equiv 1$ then $e^C$ must output the value 0 for some input and therefore, $e^C$ necessarily contains at least one disjunction $D$. $e^C(x) = 1$ for a positive test input $x$ only if the two sets $D$ and $x$ intersect (if not then $D(x) = 0$ and so $e^C(x) = 0$ ). Recall that $x$ was derived by from the string of indices $y$. The length of $D$ is at most $d$ while $y$ has $k$ indices.
The probability over the choice of $y$ that one given variable from $D$ coincides with one specific index in $y$ is exactly $\frac{1}{n}$.
The probability that this one variable from $D$ coincides with at least one of the $k$ indices of $y$ is therefore at most $\frac{k}{n}$ (by a union bound).
The probability that at least one of at most $d$ variables in $D$ coincides with at least one of the $k$ indices of $y$ is at most $d\frac{k}{n}$ (again by a union bound).
So the probability that $e^C(x) = 1$ is at most $\frac{dk}{n} = \frac{1}{2}$. Also, $f(x) = 1$ with probability at least $\frac{3}{4}$ (by Claim 4.2). So, with probability at least $\frac{1}{4}$, $e^C$ makes an error on the positive test input $x$ (both $f(x) = 1$ and $e^C(x) = 0$). ■

**Lemma 5.5 (Specialization of the Switching Lemma (First Direction):)** *When switching a d-CNF $e^C$ into a c-DNF $e^D$, $e^D$ introduces an error only on a positive test input and this happens with probability at most $(\frac{dk}{n})^c$.*

**Proof:** By Lemma 3.1 (first direction) we have at most $d^c$ sets of $c$ variables, and an error may be introduced only on a positive test input $x$ that contains such a set $C$ of length $c$. Recall that a positive test input $x$ is created by uniformly choosing a point $y \in \{1, ..., n\}^k$ in a c-wise independent probability space, and transforming the $k$ indices $y_1, ..., y_k$ into an input $x$. In order for $x$ to contain a conjunction $C$, we need the indices in $C$ to be contained in the set of the indices of $y$. There are $\binom{k}{c}$ choices of a size $c$ subset $y_{i_1}, ...., y_{i_c}$ out of the random variables $y_1, ..., y_k$. At least one such subset must coincide exactly with the set $C$. We can bound the probability that the set $y_{i_1}, ...., y_{i_c}$ equals exactly $C$, using the fact that these are independent random variables (this is where we use the $c$-wise independence of $y_1, ..., y_k$):

$$\Pr_y[\{y_{i_1}, ...., y_{i_c}\} = C] \leq \frac{c!}{n^c}$$

(since there are $c!$ ways to match every variable in $y_{i_1}, ...., y_{i_c}$ with one variable in $C$, and the probability that each one of these matched pairs coincide is $\frac{1}{n^c}$).
We multiply by $\binom{k}{c}$ to bound the probability that at least one of the possible choices of a subset $y_{i_1}, ...., y_{i_c}$ gives exactly the set $C$:

$$\Pr_y[\{y_1, ..., y_k\} \text{ contains the indices of } C] \leq \binom{k}{c} \cdot \frac{c!}{n^c} = \frac{k(k-1)...(k-c+1)}{c!}\frac{c!}{n^c} \leq \left(\frac{k}{n}\right)^c$$

So far we calculated the probability that a positive test input contains a specific set $C$ of $c$ variables. But there are up to $d^c$ such sets, so the probability that an error was introduced in the switch is:

$$\Pr_x[e^D \text{ introduces an error on } x] \leq d^c \cdot \left(\frac{k}{n}\right)^c = \left(\frac{dk}{n}\right)^c$$

■

Since $(\frac{dk}{n}) = \frac{1}{2}$, the probability that $e^D$ introduces an error on a positive test input is at most $(\frac{1}{2})^c$

**Lemma 5.6 (Specialization of the Switching Lemma (Second Direction):)** *When switching a c-DNF $e^D$ into a d-CNF $e^C$, $e^C$ introduces an error only on a negative test input and this happens with probability at most $(\frac{cl}{n})^d$.*

**Proof:** The zeros set of a negative test input $x$ contains a set $D$ of $d$ variables if $x$ gets a zero at each one of these $d$ variables. The probability for this is $(\frac{l}{n})^d$ since these are $d$ independent choices. $e^C$ can introduce an error only on a negative test input that contains at least one of at most $c^d$ disjunctions, each of size exactly $d$ (Lemma 3.1 second direction). Therefore, this happens with probability at most $c^d \cdot (\frac{l}{n})^d = (\frac{cl}{n})^d$.

■

Since $(\frac{cl}{n}) = \frac{1}{2}$, the probability that $e^C$ introduces an error on a negative test input is at most $(\frac{1}{2})^d$.

We can now prove our main theorem, a lower bound on the monotone size of the function $f$:

**Theorem 8** *mon-size*$(f) = 2^{\Omega((\frac{n}{\log n})^{\frac{1}{3}})}$

**Proof:** Due to Theorem 7 and Lemmas 5.4, 5.5 and 5.6 We get that the circuit either has at least $\frac{3}{4}2^d$ $\bigwedge$ gates or it has at least $\frac{1}{4}2^c$ $\bigvee$ gates.
Since $c = d = \frac{1}{2}(\frac{n}{\log n})^{\frac{1}{3}}$ we get:

$$\text{mon-size}(f) = 2^{\Omega((\frac{n}{\log n})^{\frac{1}{3}})}$$

■

# 6 Concluding Remarks

## 6.1 Other types of $c$-wise independent constructions

In order to define the function $f$, as described in Section 4, we used a construction of $k$ $c$-wise independent uniformly distributed random variables. Other types of constructions can be used to create different functions that obtain the same lower bound. We suggest a construction of $n$ $c$-wise independent Bernoulli random variables, each with probability $\frac{k}{n}$ for success. Each point in the probability space gives an $n$ bit string that serves as a minterm. However, we only take as a minterm an input that has between $2k$ and $\frac{k}{2}$ ones. It can be shown that a large fraction of the space still serves as minterms and from here on the proof of the lower bound is quite easy.

One can also use constructions of almost $c$-wise independent random variables [EGLNV92]. These constructions give probability spaces that give statistic difference of only $\epsilon$ from truly $c$-wise independent random variables. Such constructions yield the same bound as the one achieved above.

## 6.2 The Equivalence to the Method of approximations

The method of approximations builds an auxiliary approximator circuit for every given circuit. This circuit is created in a bottom up fashion, each gate approximated by the approximators for its input gates. The symmetric version of approximations, keeps two approximators for each gate, a c-DNF and a d-CNF. In our gate elimination method, each gate, when eliminated, is in one of these two forms and is then switched to the other. These two c-DNF and d-CNF functions involved in the elimination of a gate, are exactly the functions used as approximators of this gate in the method of approximations.

In general, every use of the method of approximations can be viewed as a gate elimination process, since the approximators are always created in a bottom up fashion. In the corresponding elimination process, each gate in its turn, can be replaced by its approximator and eliminated from the circuit.

# References


[Aj83] M. Ajtai, $\Sigma_1^1$ Formulae on finite structures, *Annals of Pure and Applied Logic* **24** (1983), pp . 1–48.

[AlBaIt86] N. Alon, L. Babai and A. Itai, A fast simple randomized parallel algorithm for the Maximal Independent Set problem, *J. of Algorithms* **7** (1986), pp. 567–583.

[AlBo87] N. Alon and R. Boppana, The monotone circuit complexity of Boolean functions, *Combinatorica* **7(1)** (1987), pp. 1–22.

[AmMa96] K. Amano and A. Maruoka, Potential of the approximation method, *Proc. of the 37th IEEE Symp. on the Foundations of Computer Science* (1996), pp. 431–440.

[An85] A. Andreev, On a method for obtaining lower bounds for the complexity of individual monotone functions, *Dolk. Akad. Nauk. SSSR* **282**(5) (1985), pp. 1033–1037 (in Russian). English translation in: *Soviet Math. Dokl.* **31**(3) (1985), pp. 530–534.

[BeUl97] C. Berg and S. Ulfberg, Symmetric approximation arguments for monotone lower bounds without sunflowers, To appear in: *Computational Complexity.*

[BeUlWi99] C. Berg, S. Ulfberg and A. Wigderson, Manuscript in preparation.

[BoSi90] R. Boppana and M. Sipser, The complexity of finite functions,in *Handbook of Theoretical Computer Science: Volume A Algorithms and Complexity*, J. van Leeuwen editor, MIT Press/Elsevier, 1990, pp. 757–804.

[EGLNV92] G. Even, O. Goldreich, M. Luby, N. Nisan and B. Velickovic, Approximations of general independent distributions, *Proc. of the 24th ACM Symp. on the Theory of Computing* (1992), pp. 10–16.

[FuSaSi81] M. Furst, J. Saxe and M. Sipser, Parity, circuits and the polynomial time hierarchy, *Proc. of the 22th IEEE Symp. on the Foundations of Computer Science* (1981), pp. 260–270.

[Ha86] J. Hastad, Almost optimal lower bounds for small depth circuits, *Proc. of the 18th ACM Symp. on the Theory of Computing* (1986), pp. 6–20. Also in *Randomness and Computation, Advances in Computing Research*, Vol 5, ed. S. Micali, JAI Press Inc (1989), pp. 143–170.

[Ha95] A. Haken, Counting bottlenecks to show monotone P$\neq$NP, *Proc. of the 36th IEEE Symp. on the Foundations of Computer Science* (1995), pp. 36–40.

[Ju97] S. Jukna, Finite limits and monotone computations: the lower bound criterion, *Proc. of the 12th IEEE Conference on Computational Complexity* (1997), pp. 302–312.

[Pu97] P. Pudlak, Lower bounds for resolution and cutting planes proofs and monotone computation, *The Journal of Symbolic Logic*, 62(3) (1997), pp. 981–998.

[Ra85] A. Razborov, Lower bounds on the monotone complexity of some Boolean function, *Dolk. Akad. Nauk. SSSR* **281**(4) (1985), 598–607 (in Russian). English translation in: *Soviet Math. Dokl.* **31** (1985), 354–357.

[Ra89] A. Razborov, On the method of approximation, *Proc. of the 21th ACM Symp. on the Theory of Computing* (1989), pp. 167–176.

[SiTs97] J. Simon and S.C. Tsai, A note on the bottleneck counting argument, In *12th Annual IEEE Conference on Computational Complexity* (1997).

[Ti84] J. Tiekenheinrich A $4n$ lower bound on monotone network complexity of a one-output boolean function. *Inform. Process. Lett.* 18 (1984), pp. 201–202.

[Ya85] A. Yao, Separating the polynomial-time hierarchy by oracles, *Proc. of the 26th IEEE Symp. on the Foundations of Computer Science* (1985), pp. 1–10.

# A A $c$-wise Independent Construction

We show a simple construction (due to [AlBaIt86]) of $q$ $c$-wise independent random variables, each uniformly distributed over the values $\{1, ..., q\}$ (for a prime number $q$). The construction uses the set of all possible polynomials of degree at most $c - 1$ over the field $\mathrm{GF}(q)$. There are $q^c$ such polynomials and each one corresponds to one point in the probability space. Given a polynomial $P$ in the set, the corresponding values of the random variables $X_1, ..., X_q$ are defined by $X_i = P(i)$.

It is easy to show that this induces a uniform distribution on each of the random variables and furthermore, that every $c$ of the random variables behave independently. The size of the probability space is $q^c$ requiring just $c \log q$ random bits as a seed.