Electronic Colloquium on Computational Complexity, Report No. 24 (2013)

Compression of Boolean Functions

Valentine Kabanets*
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
kabanets@cs.sfu.ca

Antonina Kolokolova†
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada
kol@cs.mun.ca

February 6, 2013

Abstract

We consider the problem of compression for "easy" Boolean functions: given the truth table of an n-variate Boolean function f computable by some *unknown small circuit* from a *known class* of circuits, find in deterministic time $\mathsf{poly}(2^n)$ a circuit C (no restriction on the type of C) computing f so that the size of C is less than the trivial circuit size $2^n/n$.

We get both positive and negative results. On the positive side, we show that several circuit classes for which lower bounds are proved by a method of random restrictions:

- AC^0,
- (de Morgan) formulas, and
- (read-once) branching programs,

allow non-trivial compression for circuits up to the size for which lower bounds are known. On the negative side, we show that compressing functions from any class $\mathcal{C} \subseteq \mathsf{P/poly}$ implies superpolynomial lower bounds against $\mathcal{C}$ for a function in NEXP; we also observe that compressing monotone functions of polynomial circuit complexity or functions computable by large-size AC^0 circuits would also imply new superpolynomial circuit lower bounds.

Finally, we apply the ideas used for compression to get zero-error randomized #SAT-algorithms for de Morgan and complete-basis formulas, as well as branching programs, on n variables of about quadratic size that run in expected time $2^n/2^{n^\epsilon}$, for some $\epsilon > 0$ (dependent on the size of the formula/branching program).

*Research partially supported by an NSERC Discovery grant.
†Research partially supported by an NSERC Discovery grant.

ISSN 1433-8092

1 Introduction

Finding a succinct representation of a given object is an important natural problem studied in various settings under various names. It is known as *data compression* in information theory and engineering, where objects to be compressed could be, e.g., a text file, a photographic image, or a video file. In computational complexity theory, a related problem is *circuit minimization*: the goal is to construct an (approximately) minimal-size Boolean circuit computing a given Boolean function, where the function is given by its truth table. Finally, in the computational learning theory, the goal is to *learn* (construct) a concise representation of a given object, after an interaction with a teacher who can answer certain types of queries about the object to be learned.

In this paper, we consider the problem of compressing Boolean functions from a given class $\mathcal{C}$ of functions with concise representations (e.g., functions computable by polynomial-size AC^0 circuits). Given the truth table of n-variate Boolean f function from $\mathcal{C}$, we want to find some Boolean circuit (not necessarily of the type $\mathcal{C}$) computing f such that the size of the circuit is less than $2^n/n$ (which is the trivial size achievable for any n-variate Boolean function). This is different than $\mathcal{C}$-circuit minimization considered by [AHM$^+$08] where the task is to construct a small circuit of the type $\mathcal{C}$. Our setting is closer to that of computational learning theory (non-proper exact learning [Ang87]).

There are two natural parameters to minimize: the *size* of the found circuit and the *running time* of the compression algorithm. Since the algorithm is given the full truth table as input, we consider it efficient if it runs in time $2^{O(n)}$ (polynomial in its input size). Ideally, we would like to find a circuit as small as the promised size of the concise representation of a given function f. However, any non-trivial savings over the generic $2^n/n$ circuit size [Lup58] are interesting.

The compression task as defined above can be viewed as *lossless* compression: we want the compressed image (circuit) to compute the given function exactly. One can also consider the notion of *lossy* compression where the task is to find a circuit that only approximates the given function. This is related to the concept of PAC learning [Val84].

1.1 Motivation

We want to understand "easy" functions so that we can prove lower bounds for "hard" functions. Understanding to what extent an algorithm for an easy function can be extracted from its truth table is a natural question. We have a generic circuit construction algorithm due to Lupanov [Lup58] that for every n-variate Boolean function constructs a Boolean circuit of size at most $(1+o(1))\cdot 2^n/n$. Can we improve upon this size for the truth tables of functions known to have small circuits?

There has been a renewed interest recently in the apparent connection between proofs of circuit lower bound and various algorithms for circuits from the same class: for example, SAT-algorithms [Wil11, San10, IMP12, ST12], and pseudorandom generators [IMZ12]. These results do use ideas from the lower bound *proofs*, rather than the mere existence of hard functions as in, e.g., the well-known "hardness vs. randomness" line of research initiated by [BM84, Yao82, NW94]. It is natural to ask whether we can use the known circuit lower-bound proofs to get *compression* algorithms for the corresponding circuit classes.

1.2 Our results

We show that, indeed, compression algorithms can be obtained from known lower-bound proofs, in particular, those proofs that use the method of *random restrictions*. These include the lower

bounds for AC^0 circuits [FSS84, Yao85, Hås86], for de Morgan formulas [Sub61, And87, Hås98], for branching programs [Nec66], and for read-once branching programs (see, e.g., [ABCR99]).

We show how to compress Boolean functions computable by AC^0 circuits, (de Morgan) formulas, and (read-once) branching programs of size s, where s is essentially the maximal size for which a circuit lower bound for the corresponding circuit class is known. We have the following (see Theorems 6–10 below for the exact statements and the proofs):

Compression Theorem. *(1) Boolean n-variate functions computed by AC^0 circuits of size s and depth d are compressible in time $\mathsf{poly}(2^n)$ to circuits of size at most $2^{n-n/O(\log s)^{d-1}}$. (2) Boolean n-variate functions computed by de Morgan formulas of size at most $n^{2.49}$, by formulas over the complete basis of size at most $n^{1.99}$, or by branching programs of size at most $n^{1.99}$ are compressible in time $\mathsf{poly}(2^n)$ to circuits of size at most 2^{n-n^ϵ}, for some $\epsilon > 0$ (dependent on the size of the formula/branching program). (3) Boolean n-variate functions computed by read-once branching programs of size at most $2^{0.48 \cdot n}$ are compressible in time $\mathsf{poly}(2^n)$ to circuits of size at most $2^{0.99 \cdot n}$.*

On the other hand, we show that it may be difficult to compress functions from sufficiently general circuit classes (see Theorems 11 and 15, and Corollary 14):

Compression "Barrier" Theorem. *Compressing Boolean functions from a given class $\mathcal{C}$ of polynomial-size circuits to any circuit size less than $2^n/n$ implies superpolynomial lower bounds against the class $\mathcal{C}$ for a language in NEXP.*

This theorem complements a result of Williams [Wil10] which essentially says that deciding the satisfiability of circuits from a class $\mathcal{C}$ in time slightly less than that of the trivial brute-force SAT-algorithm implies superpolynomial circuit lower bounds against $\mathcal{C}$ for a language in NEXP. Thus, both non-trivial SAT algorithms and non-trivial compression algorithms for a circuit class $\mathcal{C}$ imply superpolynomial lower bounds against that class. While our theorem indicates the difficulty of obtaining compression algorithms, it may in fact be a way to obtain new circuit lower bounds, as was demonstrated in the case of SAT algorithms for ACC^0 by Williams [Wil11]. Hence, we use the word "barrier" in quotes.

Finally, while we do not have a formal connection between compression and SAT algorithms, we do use the same ideas as in our Compression Theorem to get non-trivial #SAT algorithms for formulas of about quadratic size (see Theorem 16):

#SAT algorithms: *Counting the number of satisfying assignments for n-variate de Morgan formulas of size $n^{2.49}$, formulas over the complete basis of size $n^{1.99}$, or branching programs of size $n^{1.99}$ can be done by a zero-error randomized algorithm in expected time 2^{n-n^ϵ}, for some $\epsilon > 0$ (dependent on the size of the formula/branching program).*

Note that the running time of our #SAT-algorithm is similar to the compressed circuit size for formulas given in item (2) of the Compression Theorem. Our algorithm complements an algorithm for AC^0-#SAT of [IMP12] who get the running time similar to the compressed circuit size for AC^0 given in item (1) of the Compression Theorem. Also, our #SAT algorithms are based on the random-restriction analysis of formulas (also known as *shrinkage* of formulas), and so they complement the recent shrinkage-based constructions of pseudorandom generators for formulas and branching programs due to [IMZ12].

While there are known non-trivial *deterministic* Formula-SAT algorithms for de Morgan and complete-basis formulas [San10, ST12], those algorithms are only for *linear*-size formulas. Our #SAT algorithm for small branching programs appears to be new.

1.3 Our techniques

Suppose we are given the truth table of a function $f : \{0,1\}^n \to \{0,1\}$ that is computable by a DNF with t terms. It is well-known that finding a minimum-size DNF computing f is a special case of the Set Cover problem. A greedy heuristic for Set Cover [Joh74, Lov75, Chv79] applied to the DNF problem will find a DNF for f on at most $O(t \cdot n)$ terms, in deterministic time $\mathsf{poly}(2^n)$ (see Section 3 below for details).

The idea is to try to use a similar Set-Cover approach for compressing other classes of circuits. The main property used by the greedy Set-Cover heuristic for compressing functions computable by "small" DNFs is that such a DNF is a disjunction of "not too many" circuits of "small size" (conjunctions of at most n literals). Each such circuit corresponds to a set in the Set-Cover instance. The "small size" of the circuits implies that the number of sets (and hence, the total instance size of the Set-Cover problem) is also "small". In particular, if each circuit can be described by binary string of size $O(n)$, we get that the total size of the Set-Cover instance size is at most $2^{O(n)}$, which in turn implies that the Set-Cover heuristic will run in time $\mathsf{poly}(2^n)$.

This suggests that we can use the same greedy algorithm for compressing any n-variate Boolean function computable as a disjunction of "not too many" circuits of description size $O(n)$ each. In fact, it is possible to generalize a bit further, and to allow "few" circuits that do not have $O(n)$ description size (as long as these circuits do not accept too many inputs) (see Theorem 5).

The method of random restrictions (used for proving circuit lower bounds against AC^0 and formulas of almost quadratic size) provides a nice structural characterization of the class of n-variate Boolean functions f computable by small circuits. Roughly, it implies that the universe $\{0,1\}^n$ can be partitioned into "not too many" disjoint regions, such that the restriction of the original function f to "almost every" region is a "very simple" function [San10, IMP12, KR12]. Thus, f is computable by the disjunction of "not too many" circuits so that "almost all" of them have small description size. So our generalized greedy heuristic can be applied!

For compression algorithms, we use the "simplicity" of circuits in the disjunction to argue that they have linear-size descriptions (as required in order to achieve $\mathsf{poly}(2^n)$ running time). For #SAT algorithms, we use the "simplicity" of the circuits to argue that there will be *few distinct* subfunctions associated with the regions of the partition of $\{0,1\}^n$. Once we solve #SAT (using a brute-force algorithm) for all distinct subfunctions and store the results, we can solve #SAT for almost all regions by the table look-up, achieving a noticeable speed-up overall.

1.4 Related work

The complexity of circuit minimization was studied in [KC00, AHM$^+$08, Fel09]. In particular, [AHM$^+$08, Fel09] show that finding an approximately *minimal*-size DNF for a given truth table of an n-variate Boolean function is NP-hard, for the approximation factor n^γ for some constant $0 < \gamma < 1$.

Compressing Boolean functions is related to the setting exact learning with membership and equivalence queries [Ang87]. In the learning setting, the size of the hypothesis produced by the

learning algorithm is upper-bounded by the running time of the algorithm. In our setting, we decouple the hypothesis (compressed image) size from the running time of the learning (compression) algorithm: we allow more running time, but ask for a small-size compression. This enables us to get a good compression algorithm for small DNFs, whereas learning small DNFs is a challenging open problem (cf. [KS04]). On the other hand, all positive results in the exact learning framework yield compression algorithms for the corresponding class of Boolean functions. We can compress, for example, Boolean functions computable by polynomial-size depth-2 circuits with a Mod_p output gate (for a fixed prime p) whose inputs are arbitrary modular gates, or whose inputs are arbitrary threshold gates [BBTV97].

The remainder of the paper We give basic definitions and state structural results for AC^0 and small formulas in Section 2. In Section 3, we state and analyze the greedy algorithm that we use for compression. Our compression results are given in Section 4, compression "barrier" results in Section 5, and #SAT algorithms in Section 6. We state a number of open questions in Section 7.

2 Preliminaries

2.1 Basics

Here we recall some basic definitions of circuit classes considered in our paper; for more background on circuit complexity, consult any of the following [Weg87, BS90, Juk12].

A *literal* is either a variable, or the negation of a variable; the sign of the variable is said to be positive in the first case, and negative otherwise. A *DNF* is a disjunction of terms, where each *term* is a conjunction of literals. The following is a basic fact: For any subset $S \subseteq \{0,1\}^n$ of size t, there is a DNF $D(x_1, \ldots, x_n)$ on t terms that evaluates to 1 on each $a \in S$, and is 0 outside of S.

A *Boolean circuit* on n inputs is a directed acyclic graph with a single node of out-degree 0 (the output gate), and n in-degree 0 nodes (input gates), where each input gate is labeled by one of the variables $x_1, \ldots, x_n$, and each non-input gate by a logical function on at most 2 inputs (e.g., AND, OR, and NOT). The *size* of the circuit is the total number of gates; the *depth* is the length of a longest path in the circuit from an input gate to the output gate. The class AC^0 is a class of constant-depth circuits with NOT, AND and OR gates, where AND and OR gates have unbounded fan-in.

For a circuit class $\mathcal{C}$ and a size function $s(n)$, we denote by $\mathcal{C}[s(n)]$ the class of $s(n)$-size n-input circuits of the type $\mathcal{C}$. When no $s(n)$ is explicitly mentioned, it is assumed to be some $\mathsf{poly}(n)$.

A *Boolean formula* F on n input variables $x_1, \ldots, x_n$ is a tree whose root node is the output gate, and whose leaves are labeled by literals over the variables $x_1, \ldots, x_n$; all non-input gates are labeled by logical functions over 2 inputs. The size of the formula F, denoted by $L(F)$, is the total number of leaves. A *de Morgan formula* is a formula where the only logical functions used are AND and OR.

A *branching program* F on n input variables $x_1, \ldots, x_n$ is a directed acyclic graph with one source and two sinks (labeled 0 and 1), where each non-sink node is of out-degree 2 and is labeled by an input variable x_i, $1 \leqslant i \leqslant n$. The two outgoing edges of each non-terminal node are labeled by 0 and 1. The branching program computes by starting at the source node, and following the path in the graph using the edges corresponding to the values of the variables queried in the nodes. The program accepts if it reaches the sink labeled 1, and rejects otherwise. The size of a branching

program F, denoted by $L(F)$, is the number of nodes in the underlying graph. A branching program is (syntactic) *read-once* if on every path no variable occurs more than once.

A *decision tree* is a branching program whose underlying graph is a tree; the size of a decision tree is the number of leaves.

A *restriction* ρ of the variables $x_1, \ldots, x_n$ is a an assignment of Boolean values to some subset of the variables; the assigned variables are called set, while the remaining variables are called free. For a circuit (formula or branching program) F on input variables $x_1, \ldots, x_n$ and a restriction ρ, we define the restriction $F|_\rho$ as the circuit on the free variables of ρ, obtained from F after the set variables are "hard-wired" and the circuit is simplified.

Given a (bounded fan-in) circuit of size s, we can describe it using $O(s \log s)$ bits (by specifying the gate type and at most two incoming gates for each of the s gates). The same bound is true also for formulas and branching programs of size s.

2.2 Structure of easy Boolean functions

The following uses the (generalized) Switching Lemma [Hås86, Raz93, Bea94, IMP12].

Lemma 1 ([IMP12])**.** *Every depth d Boolean circuit C with s gates on n inputs has an equivalent DNF with at most $\mathsf{poly}(n) \cdot s \cdot 2^{n(1-\mu)}$ terms, where $\mu \geqslant 1/O(\log(s/n) + d \log d)^{d-1}$.*

Inspired by the shrinkage result of [Sub61], Santhanam [San10] obtains the following structural result for de Morgan formulas of linear size.

Lemma 2 ([San10])**.** *For every constant $c > 0$, any n-variate Boolean function computed by a formula of size at most cn has an equivalent DNF with at most $2^{n(1-\mu)}$ terms, where $\mu \geqslant 1/c^k$ for some absolute constant $k > 0$ (independent of c).*

Komargodski and Raz [KR12] define a randomized algorithm for generating a restriction for the inputs of a given de Morgan formula F on n variables such that exactly k variables are left free. The algorithm chooses one variable at a time (either deterministically, or using randomness, depending on the current restricted formula), sets the chosen variable either 0 or 1 uniformly at random (independently of the choice of the variable), and continues with the new restricted formula. For a given formula F on n variables and an integer parameter $1 \leqslant k \leqslant n$, let us denote by $R_k(F)$ the resulting distribution over such random restrictions. One of the main technical results of [KR12] is that a de Morgan formula F shrinks by a factor $(k/n)^{3/2}$ with high probability when hit with a random restriction from $R_k(F)$. A similar statement is true also for formulas over the general basis and for branching programs[1], but with the constant 3/2 replaced by 1. More precisely, we have the following "high-probability" version of the classical "expected shrinkage" result of Subbotovskaya [Sub61].

Lemma 3 ([KR12])**.** *There is a constant $c > 0$ such that for every formula (branching program) F on n variables, any $\epsilon > 0$, and any integer $1 \leqslant k \leqslant n$,*

$$\mathbb{P}_\rho \left[L(F|_\rho) \geqslant c \left(\frac{k}{n}\right)^\Gamma L(F) \right] \leqslant 2^{-k/n^\epsilon},$$

[1]The result for branching programs is implicit in [KR12]; it follows by the same arguments as for formulas over the complete basis. The only thing required is that the size of an n-variable branching program decreases by the factor $(1 - 1/n)$ in expectation when a single random variable is set randomly to 0 or 1; this can be shown easily (using the same argument as for general formulas).

where a restriction ρ is sampled according to the distribution $R_k(F)$, and where $\Gamma = 3/2$ for the case of de Morgan formulas, and $\Gamma = 1$ for the case of formulas over the complete basis and for the case of branching programs.

We use this to obtain the following structural characterization of small formulas and branching programs.

Corollary 4. *Let $F(x_1, \ldots, x_n)$ be any formula (branching program) of size $O(n^d)$, where the constant d is such that $d < 2.5$ for de Morgan formulas, and $d < 2$ for formulas over the complete basis and for branching programs. There exist constants $0 < \delta, \gamma < 1$ (dependent on d) such that for $k = \lceil n^\delta \rceil$ the following holds. The Boolean function computed by F is computable by a decision tree of depth $n - k$ whose leaves are labeled by the restrictions of F (determined by the path leading to the leaf) such that all but $2^{-n^{\delta/2}}$ fraction of the leaf labels are formulas (branching programs) on k variables of size less than n^γ .*

Proof. We consider the case of de Morgan formulas only; the case of formulas over the complete basis or branching programs can be argued analogously. Let $d = 2.5 - \nu$, for some constant $\nu > 0$. Set $\delta := \nu/4$. By Lemma 3 applied to F with $\epsilon := \delta/2$, we get that for all but $2^{-n^{\delta/2}}$ fraction of restrictions ρ from $R_k(F)$ the restricted formula $F|_\rho$ has size less than $O(n^d/n^{1.5(1-\delta)}) = O(n^{1-(5/8)\nu})$, which is at most $n^{1-\nu/2}$ for large enough n. Set $\gamma := 1 - \nu/2$.

A random restriction $\rho \in R_k(F)$ depends on the random bits used to choose $n-k$ variables to be set, as well as on $n - k$ uniformly random bits to assign to the chosen variables. These two strings of random bits are independent. Therefore, by averaging, there is a fixed choice of randomness for selecting the variables so that the probability of picking a good restriction (which shrinks the formula) is at least preserved, where the probability is over the uniformly random bits assigned to the chosen variables. Thus we get a decision tree of depth $n - k$ such that all but $2^{-n^{\delta/2}}$ fraction of its leaves correspond to the restrictions of F (obtained by following the path in the decision tree to that leaf) of size less than n^γ. □

3 Greedy Set-Cover heuristic

We first recall a well-known greedy algorithm for the problem of Set Cover [Joh74, Lov75, Chv79]. Let U be a universe, and let $S_1, \ldots, S_t \subseteq U$ be subsets. Suppose U can be covered by ℓ of the subsets. Then the following algorithm will find an approximately minimal set cover.

> Repeat the following, until all of U is covered: find a subset S_i that covers at least $1/\ell$ fraction of points in U which were not covered before, and add S_i to the set cover.

For the analysis, observe that since ℓ subsets cover U, they also cover every subset of U. Hence, in each iteration of the algorithm, there exists a subset that covers at least $1/\ell$ fraction of the not-yet-covered points. After each iteration, the size of the set of points that are not covered reduces by the factor $(1 - 1/\ell)$. Thus, after t iterations, the number of points not yet covered is at most $|U| \cdot (1 - 1/\ell)^t \leqslant |U| \cdot e^{-t/\ell}$, which is less than 1 for $t = O(\ell \cdot \ln |U|)$. Hence, this algorithm finds a set cover that is at most the factor $O(\ln |U|)$ larger than the minimal set cover.

It is easy to adapt the described algorithm to find approximately minimal DNFs. Let $f : \{0,1\}^n \to \{0,1\}$ be given by its truth table. Suppose that there exists a DNF computing f such that the DNF consists of ℓ terms (conjunctions). With each term a on n variables, we associate the

set $S_a = a^{-1}(1)$ of points of $\{0,1\}^n$ where it evaluates to 1. We enumerate over all possible terms a on n variables, and keep only those sets S_a where $S_a \subseteq f^{-1}(1)$ (i.e., S_a doesn't cover any zero of f); note that all ℓ terms of the minimal DNF for f will be kept. Next we run the greedy set cover algorithm on the universe $U = f^{-1}(1)$ and the collection of sets S_a chosen above. By the analysis above, we get $O(\ell \cdot \log |U|)$ terms such that their disjunction computes f. That is, we find a DNF for f of size at most $O(n)$ factor larger than that of the minimal DNF for f.

The running time of the described algorithm is polynomial in 2^n and the number of sets S_a. The latter is the number of all possible terms on n variables, which is at most 2^{2n} (we can use an n-bit string to describe the characteristic functions of a subset of n variables, and another n-bit string to describe the signs of the chosen variables). Thus, the overall running time is $\mathsf{poly}(2^n)$.

A similar algorithm works also for a function $f : \{0,1\}^n \to \{0,1\}$ computed by a circuit of the form $\vee_{i=1}^{\ell} C_i$, for $\ell \leqslant 2^n$, where each circuit $C_i(x_1, \ldots, x_n)$ has a "short" description. In particular, if each C_i is described by $O(n)$ bits, then we can find in time $\mathsf{poly}(2^n)$ a collection of at most $O(\ell \cdot n)$ such circuits whose disjunction computes f.

We can generalize the algorithm further to the case where a given n-variate Boolean function f is computed by a circuit $\vee_{i=1}^{\ell+1} C_i$, for $\ell \leqslant 2^n$, where all but one circuit are small, while the remaining circuit accepts few inputs. More precisely, we have the following.

Theorem 5. *There is a deterministic $\mathsf{poly}(2^n)$-time algorithm A satisfying the following. Let $f : \{0,1\}^n \to \{0,1\}$ be any function computable by a circuit $\vee_{i=1}^{\ell+1} C_i$, for $1 \leqslant \ell \leqslant 2^n$, where the circuits $C_1, \ldots, C_\ell$ have both circuit size and description size at most cn for a constant $c > 0$, while the last circuit $C_{\ell+1}$ evaluates to 1 on at most fraction α of points in $\{0,1\}^n$, for some $0 \leqslant \alpha < 1$.*

Given the truth table of f and the parameters ℓ, c, and α, algorithm A finds a circuit for f of the form $\vee_{i=1}^{m} D_i$, where $m = O(n \cdot \ell)$, the circuits $D_1, \ldots, D_{m-1}$ are of size $O(n)$ each, and the circuit D_m is a DNF with $O(\alpha 2^n)$ terms. Hence the overall size of the found circuit is $O(\ell n^2 + \alpha n 2^n)$.

Proof. Let $U = f^{-1}(1)$, and let $\beta = |U|/2^n$. If $\beta \leqslant 2\alpha$, then our algorithm A outputs the circuit which is a DNF with $\beta 2^n$ terms, where each term evaluates to 1 on a single point in U, and is 0 everywhere else. Note that the size of this circuit is $O(\alpha n 2^n)$, as required.

If $\beta > 2\alpha$, then algorithm A does the following.

> Enumerate[2] all linear-size circuits C of description size at most cn, keeping only those C where $C^{-1}(1) \subseteq f^{-1}(1)$. Call the kept circuits *legal*. Let $\mathcal{S} = \varnothing$.
>
> Repeat the following until the number of not-yet-covered points of U becomes at most $2\alpha 2^n$: find a legal circuit C such that the set $C^{-1}(1)$ covers at least $1/(2\ell)$ fraction of not-yet-covered points in U, and add C to the set $\mathcal{S}$.
>
> Once the number of non-covered points in U becomes at most $2\alpha 2^n$, construct a DNF D that evaluates to 1 on each non-covered point, and is 0 everywhere else. Output the disjunction of D and the circuits in $\mathcal{S}$.

For the analysis, let $W = C_{\ell+1}^{-1}(1)$, and let let $V = U \setminus W$. We claim that at each iteration of the algorithm before the last iteration, the set of not-yet-covered points in V is at least as big as the set of not-yet-covered points in W. Indeed, otherwise the total number of not-yet-covered points at that iteration is at most $2 \cdot |W| \leqslant 2\alpha 2^n$, making this the last iteration of the algorithm.

[2]Here we assume the correspondence between circuits and their descriptions is efficiently computable and is known.

Next observe that at each iteration before the last one, the set of not-yet-covered points in V is non-empty, and is covered by ℓ legal circuits. Hence, there is a legal circuit that covers at least $1/\ell$ fraction of non-covered points in V, which, by the earlier remark, constitutes at least $1/(2\ell)$ fraction of all non-covered points of U. Thus our algorithm will always find a required legal circuit C. It follows that after each iteration, the size of not-yet-covered points in U decreases by the factor $(1 - 1/(2\ell))$, and hence the total number of iterations is $t = O(\ell \cdot \log |U|) = O(\ell \cdot n)$.

Thus, after at most t iterations, at most $2\alpha 2^n$ points of U are still not covered. We denote the t found circuits $D_1, \ldots, D_t$, and let D_{t+1} be the DNF with at most $2\alpha 2^n$ terms which evaluates to 1 on the non-covered points of U, and is 0 everywhere else. Note that the circuit size of D_{t+1} is $O(\alpha n 2^n)$, while all D_i's, for $1 \leqslant i \leqslant t$, are of circuit size $O(n)$ by construction. Also note that the overall running time of the described algorithm is $\mathsf{poly}(2^n, t) = \mathsf{poly}(2^n)$. The theorem follows. □

4 Compression

4.1 AC^0 functions

As we mentioned in Section 3, a Boolean function $f : \{0,1\}^n \to \{0,1\}$ computable by a DNF with $s(n)$ terms can be efficiently compressed to a DNF with $O(n \cdot s(n))$ terms. Since Boolean functions computable by small AC^0 circuits can be computed by somewhat small DNFs (of less than the trivial size) by Lemma 1, we also get non-trivial compression for such AC^0 functions. The following theorem follows easily from Lemma 1 and the greedy algorithm in Theorem 5.

Theorem 6. *There is a deterministic* $\mathsf{poly}(2^n)$*-time algorithm A satisfying the following. Let $f : \{0,1\}^n \to \{0,1\}$ be any Boolean function computable by an AC^0 circuit of depth d and size $s = s(n)$. Given the truth table of f as well as the parameters d and s, algorithm A produces a DNF for f with at most $\mathsf{poly}(n) \cdot s \cdot 2^{n(1-\mu)}$ terms, where $\mu \geqslant 1/O(\log s)^{d-1}$.*

Note the described algorithm achieves nontrivial compression for depth-d AC^0 circuits of size up to $2^{n^{1/(d-1)}}$, the size for which we know lower bounds against AC^0 for explicit functions.

4.2 Formulas and branching programs

Similarly, using the existence of somewhat small DNFs for linear-size de Morgan formulas stated in Lemma 2, we get compression of the functions computable by such formulas.

Theorem 7. *There is a deterministic* $\mathsf{poly}(2^n)$*-time algorithm A satisfying the following. Let $f : \{0,1\}^n \to \{0,1\}$ be any Boolean function computable by a de Morgan formula of size cn for some constant $c > 0$. Given the truth table of f as well as the parameter c, algorithm A produces a DNF for f with at most $O(n) \cdot 2^{n(1-\mu)}$ terms, where $\mu \geqslant 1/c^k$ for some absolute constant $k > 0$ (independent of c).*

Seto and Tamaki [ST12] extended the results of [San10] from linear-size de Morgan formulas to linear-size formulas over the complete basis. In particular, they show that such formulas can be computed by a somewhat small decision tree whose nodes are labeled by parities of subsets of input variables. While we do not know how to use this structural characterization from [ST12] to get compression for linear-size formulas over the complete basis, we do get compression of both de Morgan and complete-basis formulas (even of quadratic size) by using the structural characterization of small formulas implicit in the work of Komargodski and Raz [KR12] (cf. Corollary 4 above).

Theorem 8. *There is an efficient compression algorithm that, given the truth table of a formula (branching program) F on n variables of size $L(F) \leqslant n^d$, the algorithm produces an equivalent Boolean circuit of size at most 2^{n-n^ϵ}, for some constant $0 < \epsilon < 1$ (dependent on d), where the constant d is such that*

- *$d < 2.5$ for de Morgan formulas, and*
- *$d < 2$ for formulas over the complete basis and for branching programs.*

Proof. Let F be a de Morgan formula, a complete-basis formula, or a branching program of the size stated in the theorem. By Corollary 4, this F can be computed by a decision tree of depth $m := n - n^\delta$ such that all but at most $\alpha := 2^{-n^{\delta/2}}$ fraction of the leaves correspond to restricted subformulas of F of size n^γ on $k := n^\delta$ variables, for some constants $0 < \delta, \gamma < 1$ dependent on d.

Each leaf of the decision tree corresponds to a restriction of some subset of m input variables. Let us associate with each leaf i, $1 \leqslant i \leqslant 2^m$, of the decision tree, the conjunction c_i of m literals that defines the corresponding restriction. Also let F_i, for $1 \leqslant i \leqslant 2^m$, denote the restriction of the original F corresponding to the restriction given by c_i. We get that

$$F \equiv \vee_{i=1}^{2^m}(c_i \wedge F_i).$$

We know that all but $b := \alpha \cdot 2^m$ of formulas F_i are sublinear-size n^γ. Let us assume, without loss of generality, that all the first $\ell := 2^m - b$ formulas F_i are small. Define the circuits $C_i := (c_i \wedge F_i)$, for $1 \leqslant i \leqslant \ell$, and $C_{\ell+1} := \vee_{i=\ell+1}^{2^m}(c_i \wedge F_i)$.

Observe that the circuit $C_{\ell+1}$ can evaluate to 1 on at most $b \cdot 2^k = \alpha \cdot 2^n$ inputs from $\{0,1\}^n$ (since the decision tree of depth m partitions the set $\{0,1\}^n$ into 2^m disjoint subsets of size 2^k each, and $C_{\ell+1}$ corresponds to b such subsets). Each circuit C_i, for $1 \leqslant i \leqslant \ell$, is of size at most $O(m + n^\gamma) \leqslant O(n)$. We also claim that each such circuit can be described by a string of $O(n)$ bits. Indeed, we can specify the conjunction c_i using $2n$ bits (n bits to describe the subset of variables in the conjunction, and another n bits to specify the signs of the variables), and we can specify the formula (branching program) F_i of size n^γ by at most $O(n^\gamma \log n) \leqslant O(n)$ bits in the standard way.

Thus we get that $F \equiv \vee_{i=1}^{\ell+1} C_i$ satisfies the assumption of Theorem 5. Running the greedy algorithm of Theorem 5, we get a circuit for F of total size at most $O(\ell n^2 + \alpha n 2^n) \leqslant \mathsf{poly}(n) \cdot (2^{n-n^\delta} + 2^{n-n^{\delta/2}})$, which is at most 2^{n-n^ϵ} for $\epsilon \approx \delta/2$. □

4.3 Read-once branching programs

It is convenient for us to use the following canonical form of a read-once branching program. We call a program *full* if, for every node v of the program, all paths leading from the start node to v query the same set of variables (not necessarily in the same order).

Lemma 9. *Every read-once branching program F of size s on n inputs has an equivalent full read-once branching program F' of size $s' \leqslant 3n \cdot s$.*

Proof. Given F, construct F' inductively as follows. Consider nodes of F in the topological order from the start node. The start node obviously satisfies the fullness property. For every node v of F with distinct predecessor nodes $u_1, \ldots, u_t$, for $t \geqslant 2$, let X_i denote the set of variables queried by the paths from start to u_i; note that, by the inductive hypothesis, all paths leading to u_i query the same set X_i of variables. Let $X = \cup_{i=1}^t X_i$. For every $i \in \{1, \ldots, t\}$, let $\Delta_i = X \setminus X_i$ be the set

of "missing" variables. If $\Delta_i \neq \varnothing$, replace the edge (u_i, v) by a multi-path $u_i, w_1, w_2, \ldots, w_r, v$, for $r = |\Delta_i|$, where w_j's are new nodes labeled by the "missing" variables from Δ_i (in any fixed order), with the edge (u_i, w_1) labeled as the edge (u_i, v), and each w_j has two edges to its successor node on the path, labeled by 0 and by 1, respectively.

Since our original program is read-once, no variable from the set X for a node v can occur after v. Thus, adding the queries to the "missing" variables for every predecessor of v preserves the property of being read-once, and preserves the functionality of the branching program. It also makes the node v and all of its predecessors satisfy the fullness property. Hence, after considering all nodes v, we obtain a required full read-once branching program F' equivalent to F. The size of F' is at most $s + 2sn$ since we add at most n dummy nodes for each of at most $2s$ edges of F. □

Theorem 10. *There is a deterministic* $\mathsf{poly}(2^n)$*-time algorithm A satisfying the following. Let $f : \{0,1\}^n \to \{0,1\}$ be any Boolean function computable by a read-once branching program of size s. Given the truth table of f, algorithm A produces a formula for f of size at most $O(sn^3 \cdot 2^{n/2})$.*

Proof. By Lemma 9, f is computable by a full read-once branching program F of size $s' = 3sn$. For $0 \leqslant k \leqslant n$ to be chosen later, consider the set B of all nodes at distance $n - k$ from the start node. Clearly, there are at most s' such nodes. For every such node v, let X_v be the set of $n - k$ variables queried on every path from the start to v. Let Y_v be the remaining k variables. Associate with v the function h_v in the variables X_v computed by the branching subprogram with v as the new accepting terminal node (and the same start node), and the function g_v in the variables Y_v computed by the branching subprogram with v as the new start node (and the same terminal nodes). We may assume that the functions g_v are distinct for distinct nodes in B; otherwise, we merge all nodes with the same g_v (on the same subset of k variables) into a single node. We have

$$f \equiv \vee_{v \in B}(h_v \wedge g_v). \tag{1}$$

Consider any $v \in B$. Let ρ be a restriction of the variables X_v corresponding to some path from the start to v. We have $g_v = f|_\rho$, and h_v is the disjunction of all restrictions ρ' of the variables X_v such that $f|_{\rho'} = g_v = f|_\rho$. Thus, to describe any disjunct in the representation of f given by Eq. (1), it suffices to specify a restriction of some subset of $n - k$ variables of f; this can be described using $O(n)$ bits.

We now run the greedy Set-Cover heuristic to find at most $O(s'n)$ functions, each describable by a restriction of some $n - k$ variables as explained above, whose disjunction equals f. For each restriction ρ specifying one of these functions, the corresponding function can be computed as an AND of a DNF of size 2^k (for the function $f|_\rho$ on k variables) and a DNF of size 2^{n-k} (for all restrictions ρ' on $n - k$ variables that yield $f|_{\rho'} = f|_\rho$). The overall circuit size of each of these $O(s'n)$ functions is then $O(n(2^k + 2^{n-k}))$, and the overall size of the circuit computing f is $O(s'n^2(2^k + 2^{n-k}))$, which is at most $O(sn^3 \cdot 2^{n/2})$, if we set $k = n/2$. The running time of the compression algorithm is $\mathsf{poly}(2^n)$ since we only need to enumerate all $O(n)$-size descriptions. □

5 Circuit lower bounds from compression

It is quite easy to see that compressibility of Boolean functions is a special case of a natural property in the sense of [RR97]; so the existence of compression algorithms for a circuit class $\mathcal{C}$ implies that there is no strong PRG in $\mathcal{C}$. Here we argue that such compression algorithms would also yield circuit lower bounds against $\mathcal{C}$ for a language in NEXP.

5.1 Arbitrary subclass of polynomial-size circuits

The following is a refinement of a result in [IKW02].

Theorem 11. *Let $\mathcal{C} \subseteq \mathsf{P/poly}$ be any circuit class. Suppose that for every $c \in \mathbb{N}$ there is a deterministic polynomial-time algorithm that compresses a given truth table of an n-variate Boolean function $f \in \mathcal{C}[n^c]$ to a circuit of size less than $2^n/n$. Then $\mathsf{NEXP} \not\subseteq \mathcal{C}$.*

Proof. Suppose, for the sake of contradiction, that $\mathsf{NEXP} \subseteq \mathcal{C} \subseteq \mathsf{P/poly}$.

Claim 12. *If $\mathsf{NEXP} \subseteq \mathcal{C}$, then for every $L \in \mathsf{NEXP}$ there is a $c \in \mathbb{N}$ such that, for all sufficiently large n, every n-bit string $x \in L$ has a witness computable by an $\mathcal{C}$-circuit of size n^c.*

Proof. By [IKW02], the assumption $\mathsf{NEXP} \subseteq \mathsf{P/poly}$ implies that, for every language $L \in \mathsf{NEXP}$, there exists a constant $c_L \in \mathbb{N}$ such that every sufficiently large input $x \in L$ has a NEXP-witness that is the truth table of some Boolean function of circuit complexity n^{c_L}. For every $L \in \mathsf{NTIME}(2^{n^e})$, define a new language $L' \in \mathsf{EXP}$ as follows: on inputs x, y, where $|x| = n$ and $|y| = n^e$, search through the circuits of size n^{c_L} until find an NEXP-witness for $x \in L$. If no such witness is found, then output 0. Otherwise, output the yth bit of the found witness (which is the truth table of a n^{c_L}-size circuit). We get that for every $x \in L$, a string y is such that $(x, y) \in L'$ iff the yth bit of the lex first witness for x (as found by the algorithm enumerating all n^{c_L} size circuits) is 1. Since $\mathsf{EXP} \subseteq \mathcal{C}$, we get that $L' \in \mathcal{C}$. So, every $x \in L$ has a witness that is the truth table of Boolean function computable by a polynomial-size $\mathcal{C}$-circuit. □

Consider now a universal language L for NE, with $L \in \mathsf{NTIME}(2^{n^2})$. For $\mathsf{NTIME}(2^{cn})$ for every $c \in \mathbb{N}$, the witness size for inputs of size n is bounded by $2^{cn} \leqslant 2^{n^2}$ for large enough n. We think of witnesses for NE languages (on inputs of size n) as the truth tables of m-variate Boolean functions for $m = n^2$: such a string of length 2^m is a witness iff its prefix of appropriate length is a witness. By Claim 12 above, we get that there is a constant $c_0 \in \mathbb{N}$ such that yes-instances x, $|x| = n$, of every language in NE have witnesses that are truth tables of $m = n^2$-variate Boolean functions computable in $\mathcal{C}[m^{c_0}]$.

Suppose we have a deterministic $\mathsf{poly}(2^n)$-time compression algorithm for n-variate Boolean functions in $\mathcal{C}[n^{2c_0}]$. Consider the following NE algorithm:

> On input x of size n, nondeterministically guess a binary string of length 2^n. Run the compression algorithm on the guessed string. Accept iff the compression algorithm didn't produce a circuit of size less than $2^n/n$ for this string.

Observe that the described algorithm accepts every input x since there are incompressible strings of every length 2^n. Its running time is $\mathsf{poly}(2^n)$ dependent on the running time of the assumed compression algorithm. Note that every witness for an input x is a string that our compression algorithm fails to compress, which means that the witness is the truth table of an n-variate Boolean function that requires $\mathcal{C}$-circuits of size greater than n^{2c_0}. If we think of this 2^n-bit witness as the prefix of a 2^{n^2}-bit truth table of an $m = n^2$-variate Boolean function, we conclude that the latter m-variate Boolean function requires $\mathcal{C}$ circuits of size greater than m^{c_0}. But this contradicts the fact we established earlier that every NE language must have $\mathcal{C}[m^{c_0}]$ computable witnesses. □

Remark 13. It is easy to get an analogue of Theorem 11 also for deterministic *lossy* compression algorithms, with the same proof.

5.2 Large-size AC^0 functions

Compressing functions computable by "large" AC^0 circuits (of size 2^{n^ϵ} with $\epsilon \gg 1/d$, where d is the depth of the circuit) is difficult since every function computable by a polynomial-size NC^1 circuit has an equivalent AC^0 circuit of size 2^{n^ϵ} (and some depth d dependent on ϵ). The existence of a compression algorithm for such large AC^0 circuits would imply a *natural property* in the sense of [RR97] useful against NC^1. The latter implies that no strong enough PRG can be computed by NC^1 circuits [RR97, AHM+08].

Using Theorem 11, we get that such compression would also imply that $\mathsf{NEXP} \not\subset \mathsf{NC}^1$.

Corollary 14. *For every $\epsilon > 0$ there is a $d \in \mathbb{N}$ such that the following holds. If there is a deterministic polynomial-time algorithm that compresses a given truth table of an n-variate Boolean function $f \in \mathsf{AC}^0_d[2^{n^\epsilon}]$ to a circuit of size less than $2^n/n$, then $\mathsf{NEXP} \not\subset \mathsf{NC}^1$.*

5.3 Monotone functions

Every monotone Boolean function on n variables can be computed by a (monotone) circuit of size $O(2^n/n^{1.5})$ [Pip77, Red79]. We argue that compressing polynomial-size monotone functions is as hard as compressing arbitrary functions in $\mathsf{P/poly}$.

Theorem 15. *If there is an efficient algorithm that compresses a given truth table of an m-variate monotone Boolean function of monotone circuit size $\mathsf{poly}(m)$ to a (not necessarily monotone) circuit of size at most $2^m/m^{1.51}$, then there is an efficient algorithm for compressing arbitrary n-variate $\mathsf{P/poly}$-computable Boolean functions to circuits of size less than $2^n/n$, and hence, $\mathsf{NEXP} \not\subseteq \mathsf{P/poly}$.*

Proof sketch. The idea is to use the well-known connection between non-monotone functions and monotone slice functions [Ber82]. We use an optimal embedding of an arbitrary n-variate Boolean function f into the middle slice of a monotone slice function g on m variables for $m = n + (\log n)/2 + \Theta(1)$ due to [KKM12]. Given a truth table of f, we can efficiently construct the truth table of this monotone function g. The mapping between n-bit inputs of f and the corresponding m-bit inputs of g (of Hamming weight $m/2$) is computable and invertible in time $\mathsf{poly}(m) = \mathsf{poly}(n)$. Hence, a circuit for g of size at most $2^m/m^{1.51}$ yields a circuit for f of size at most $O((2^n/n^{1.01}) + \mathsf{poly}(n))$, which is less than $2^n/n$ for large enough n. Appealing to Theorem 11 concludes the proof. □

6 #SAT algorithms from shrinkage

Impagliazzo, Mathews, and Paturi [IMP12] showed that (a generalization of) Håstad's Switching Lemma for AC^0 yields a randomized (zero-error) algorithm for counting the number of satisfying assignments of a given depth-d AC^0 circuit on n inputs of size $s = s(n) \leqslant 2^{n^{1/(d-1)}}$, whose running time is essentially $2^{n-n/(\log s)^{d-1}}$. Inspired by Subbotovskaya's result on the shrinkage of de Morgan formulas under random restrictions [Sub61], Santhanam [San10] gave a deterministic algorithm for counting the number of satisfying assignments of cn-size de Morgan formulas on n variables whose running time is $2^{n-\epsilon n}$, for $\epsilon = \mathsf{poly}(1/c)$. This algorithm has been extended to cn-size formulas over the complete basis by Seto and Tamaki [ST12] with the running time $2^{n-\epsilon n}$, for $\epsilon = 2^{-O(c^3)}$. Note that none of these algorithms appears able to handle formulas of size bigger than, say, $n^{1.5}$.

We show that the "high-probability" shrinkage result of [KR12] can be used to get randomized #SAT algorithms for formulas and branching programs of about quadratic size.

Theorem 16. *There is a randomized (zero-error) algorithm for counting the number of satisfying assignments in a given formula on n variables of size at most n^d which runs in (expected) time $t(n) \leqslant 2^{n-n^\alpha}$, for some constant $0 < \alpha < 1$ (dependent on d), where the constant d is such that*

- *$d < 2.5$ for de Morgan formulas, and*
- *$d < 2$ for formulas over the complete basis and for branching programs.*

Proof. Let F be a given formula (branching program) of the size stipulated in the theorem. By Lemma 3, there is an efficient randomized algorithm for constructing a decision tree specified in Corollary 4, where the randomness is used only for choosing the variables to be set. Recall that the depth of this tree is $n - k$ for $k = n^\mu$, where $0 < \mu < 1$.

By averaging, we get that, with probability at least $1/2$, the constructed decision tree has all but at most 2^{-n^ϵ} fraction of its leaves correspond to restricted subformulas of F on k variables of size at most n^γ, for some $0 < \epsilon, \gamma < 1$ (dependent on d). Given a particular decision tree, we can check if sufficiently many of its leaves correspond to such small subformulas. If it is not the case, we re-compute the decision tree, using fresh randomness. Within n trials, we get a good decision tree with probability at least $1 - 2^{-n}$. Thus, a good decision tree of depth $n - k$ can be constructed in expected time at most $2^{n-k} \cdot \mathsf{poly}(n)$.

Suppose we have found a good decision tree of depth $n - k$. This tree defines a partition of the domain $\{0,1\}^n$ into 2^{n-k} *disjoint* regions of size 2^k each. Thus, to count the number of satisfying assignments of F, it suffices to count the number of satisfying assignments for the restriction of F to each region, and add up these counts over all regions. To save over the naive brute-force #SAT-algorithm, we make the following crucial observation: *Almost all of 2^{n-k} branches are labeled by sublinear-size formulas, and the number of such formulas is much less than the total number of branches.* Hence, many leaves are labeled by equivalent formulas! If we pre-compute and store the number of satisfying assignments for all small formulas, we will gain a noticeable speed-up.

In more detail, once we find a good decision tree, we continue as follows.

> *Stage 1.* Enumerate over all k-input formulas of size at most n^γ, and compute the number of satisfying assignments for each. Enter the results into an array indexed by such k-input formulas.
>
> *Stage 2.* Go over all 2^{n-k} branches of the decision tree, and for each leaf-formula do the following: if it is of formula size at most n^γ, look up its number of satisfying assignments in the array constructed earlier; otherwise, compute the number of its satisfying assignment by a brute-force algorithm in time $2^k \cdot \mathsf{poly}(n)$. Add up all the numbers over all branches, and output the result.

Obviously, the described algorithm is always correct. For its time analysis, observe that the size of the array built in Stage 1 is $T \leqslant 2^{O(n^\gamma \log n)}$, and we can fill it in time at most $T \cdot 2^k \cdot \mathsf{poly}(n) \leqslant 2^{n^{\epsilon'}}$ for some $\epsilon' < 1$. For Stage 2, we spend time at most $2^{n-k} \cdot \mathsf{poly}(n)$ (for small subformulas) plus $2^{n-k} \cdot 2^k \cdot \mathsf{poly}(n)/2^{n^\epsilon}$ (for large subformulas). It follows that the overall running time for finding a good decision tree and for Stages 1 and 2 is at most 2^{n-n^α}, for some $\alpha > 0$, as required. □

7 Open questions

We have shown efficient compressibility of functions computable by small circuits from several classes $\mathcal{C}$ where known lower bounds are shown using the method of random restrictions. Can we

extend this to other circuit classes with known lower bounds, e.g., constant-depth circuits with prime-modular gates for which the polynomial-approximation method was used [Raz87, Smo87]? Can we compress functions computable by ACC^0 circuits? By Theorem 11, this would yield an alternative proof of Williams's result [Wil11]. Incompressibility is a special case of a natural property in the sense of [RR97]. Can we argue that all known circuit lower bound proofs yield compression algorithms for the corresponding circuit classes?

The compressed circuit sizes for our compression algorithms are barely less than exponential. Is it possible to achieve better compression for the circuit classes considered?

We have used the ideas of our compression algorithm for small formulas to get also a #SAT-algorithm for small formulas. Is there a general connection between compression and SAT algorithms?

Finally, the focus of the present paper has been on lossless compression. For small AC^0 circuits and small AC^0 circuits with few threshold gates, one can get nontrivial lossy compression using the Fourier transform [LMN93, GS10]. What about lossy compression for other circuit classes?

Acknowledgements We would like to thank Ran Raz and Dieter van Melkebeek for helpful discussions.

References

[ABCR99] A.E. Andreev, J. L. Baskakov, A. E. F. Clementi, and J. D. P. Rolim. Small pseudo-random sets yield hard functions: New tight explict lower bounds for branching programs. In *ICALP*, pages 179–189, 1999.

[AHM+08] E. Allender, L. Hellerstein, P. McCabe, T. Pitassi, and M.E. Saks. Minimizing disjunctive normal form formulas and AC^0 circuits given a truth table. *SIAM Journal on Computing*, 38(1):63–84, 2008.

[And87] A.E. Andreev. On a method of obtaining more than quadratic effective lower bounds for the complexity of π-schemes. *Vestnik Moskovskogo Universiteta. Matematika*, 42(1):70–73, 1987. English translation in *Moscow University Mathematics Bulletin*.

[Ang87] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[BBTV97] F. Bergadano, N.H. Bshouty, C. Tamon, and S. Varricchio. On learning programs and small depth circuits. In *Computational Learning Theory, Third European Conference, EuroCOLT '97*, pages 150–161, 1997.

[Bea94] P. Beame. A switching lemma primer. Technical report, Department of Computer Science and Engineering, University of Washington, 1994.

[Ber82] S.J. Berkowitz. On some relationships between monotone and non-monotone circuit complexity. Technical report, University of Toronto, 1982.

[BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:850–864, 1984.

[BS90] R. B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of theoretical computer science (vol. A)*, pages 757–804. MIT Press, Cambridge, MA, USA, 1990.

[Chv79] V. Chvátal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[Fel09] V. Feldman. Hardness of approximate two-level logic minimization and PAC learning with membership queries. *Journal of Computer and System Sciences*, 75(1):13–26, 2009.

[FSS84] M. Furst, J.B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, April 1984.

[GS10] P. Gopalan and R. A. Servedio. Learning and lower bounds for AC^0 with threshold gates. In *Proceedings of the 13th international conference on Approximation, and 14th International conference on Randomization, and combinatorial optimization: algorithms and techniques*, APPROX/RANDOM'10, pages 588–601, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hås86] J. Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 6–20, 1986.

[Hås98] J. Håstad. The shrinkage exponent of de morgan formulae is 2. *SIAM Journal on Computing*, 27:48–64, 1998.

[IKW02] R. Impagliazzo, V. Kabanets, and A. Wigderson. In search of an easy witness: Exponential time vs. probabilistic polynomial time. *Journal of Computer and System Sciences*, 65(4):672–694, 2002.

[IMP12] R. Impagliazzo, W. Matthews, and R. Paturi. A satisfiability algorithm for AC^0. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 961–972, 2012.

[IMZ12] R. Impagliazzo, R. Meka, and D. Zuckerman. Pseudorandomness from shrinkage. In *Proceedings of the Fifty-Third Annual IEEE Symposium on Foundations of Computer Science*, 2012.

[Joh74] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[Juk12] S. Jukna. *Boolean Function Complexity: Advances and Frontiers*. Springer, 2012.

[KC00] V. Kabanets and J.-Y. Cai. Circuit minimization problem. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, pages 73–79, 2000.

[KKM12] G. Karakostas, J. Kinne, and D. van Melkebeek. On derandomization and average-case complexity of monotone functions. *Theoretical Computer Science*, 434:35–44, 2012.

[KR12] I. Komargodski and R. Raz. Average-case lower bounds for formula size. *Electronic Colloquium on Computational Complexity (ECCC)*, 19, 2012.

[KS04] A.R. Klivans and R.A. Servedio. Learning DNF in time $2^{\tilde{O}(n^{1/3})}$. *Journal of Computer and System Sciences*, 68(2):303–318, 2004.

[LMN93] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, Fourier transform and learnability. *Journal of the Association for Computing Machinery*, 40(3):607–620, 1993.

[Lov75] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[Lup58] O.B. Lupanov. On the synthesis of switching circuits. *Doklady Akademii Nauk SSSR*, 119(1):23–26, 1958. English translation in *Soviet Mathematics Doklady*.

[Nec66] E.I. Nechiporuk. On a Boolean function. *Doklady Akademii Nauk SSSR*, 169(4):765–766, 1966. English translation in *Soviet Mathematics Doklady*.

[NW94] N. Nisan and A. Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49:149–167, 1994.

[Pip77] N. Pippenger. The complexity of monotone boolean functions. *Theory of Computing Systems*, 11:289–316, 1977.

[Raz87] A.A. Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *Mathematical Notes*, 41:333–338, 1987.

[Raz93] A.A. Razborov. Bounded arithmetic and lower bounds in boolean complexity. In *Feasible Mathematics II*, pages 344–386. Birkhauser, 1993.

[Red79] N.P. Red'kin. On the realization of monotone boolean functions by contact circuits. *Problemy Kibernetiki*, 35:87–110, 1979. (in Russian).

[RR97] A.A. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55:24–35, 1997.

[San10] R. Santhanam. Fighting perebor: New and improved algorithms for formula and QBF satisfiability. In *Proceedings of the Fifty-First Annual IEEE Symposium on Foundations of Computer Science*, pages 183–192, 2010.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 77–82, 1987.

[ST12] K. Seto and S. Tamaki. A satisfiability algorithm and average-case hardness for formulas over the full binary basis. In *Proceedings of the Twenty-Seventh Annual IEEE Conference on Computational Complexity*, pages 107–116, 2012.

[Sub61] B.A. Subbotovskaya. Realizations of linear function by formulas using $\vee$, $\&$, $^-$. *Doklady Akademii Nauk SSSR*, 136(3):553–555, 1961. English translation in *Soviet Mathematics Doklady*.

[Val84] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Weg87] I. Wegener. *The Complexity of Boolean Functions*. J. Wiley, New York, 1987.

[Wil10] R. Williams. Improving exhaustive search implies superpolynomial lower bounds. In *Proceedings of the Forty-Second Annual ACM Symposium on Theory of Computing*, pages 231–240, 2010.

[Wil11] R. Williams. Non-uniform ACC circuit lower bounds. In *Proceedings of the Twenty-Sixth Annual IEEE Conference on Computational Complexity*, pages 115–125, 2011.

[Yao82] A.C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[Yao85] A.C. Yao. Separating the polynomial-time hierarchy by oracles. In *Proceedings of the Twenty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, pages 1–10, 1985.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de