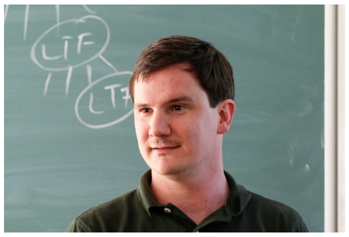
Some Ways of Thinking Algorithmically About Impossibility[1]

Ryan Williams,
MIT Computer Science and AI Laboratory, Cambridge, MA, USA



Computational complexity lower bounds like $\mathsf{P} \neq \mathsf{NP}$ assert impossibility results for all possible programs of some restricted form. As there are presently enormous gaps in our lower bound knowledge, a central question on the minds of today's complexity theorists is *how will we find better ways to reason about all efficient programs?*

I argue that some progress can be made by (very deliberately) thinking *algorithmically* about lower bounds. Slightly more precisely, to prove a lower bound against some class $\mathcal{C}$ of programs, we can start by treating $\mathcal{C}$ as a set of inputs to another (larger) process, which is intended to perform some basic analysis of programs in $\mathcal{C}$. By carefully studying the algorithmic "meta-analysis" of programs in $\mathcal{C}$, we can learn more about the *limitations* of the programs being analyzed.

This essay is mostly self-contained; scant knowledge is assumed of the reader.

1. INTRODUCTION

We use the term *lower bound* to denote an assertion about the computational intractability of a problem. For example, the assertion "factoring integers of 2048 bits cannot be done with a Boolean circuit of 10^6 gates" is a lower bound which we hope is true (or at least, if the lower bound is false, we hope that parties with sinister motivations have not managed to find such a magical circuit).

The general problem of mathematically proving computational lower bounds is largely a mystery. The stability of modern commerce relies on certain lower bounds being true (most prominently in cryptography and computer security). Yet for practically all of the prominent lower bound problems, we do not know how to begin proving them true—we do not even know *step zero*. (For some major open problems, such as the Permanent versus Determinant problem in arithmetic complexity [Mulmuley 2012], we do have good candidates for step zero, and possibly step one.) Many present lower bound conjectures may well be false. In spite of considerable intuitions that we have about lower bounds, we must admit that our formal understanding of them is awfully weak. This translates to a lack of understanding about algorithms as well.

In this article, I describe two recently developed ways of viewing lower bound problems in a constructive, algorithmic way. Of course the *idea* of viewing lower bounds in this way is not at all new. A good example from mathematical logic is the framework of Ehrenfeucht-Frässé games [Frässé 1950; Ehrenfeucht 1961], with which one can prove that certain core problems cannot be expressed in certain logics, by constructing winning strategies for a special two-player game on logical structures. The methods described here are rather different from EF games, and are deeply rooted in the language and methods of worst-case algorithm design.

[1]**This article is an expanded version of an invited article by the author that appeared in the proceedings of the 25th EACSL Annual Conference on Computer Science Logic.** Supported in part by NSF CCF-1212372 and a Sloan Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1.1. Barriers

Why are lower bounds so difficult to prove? There are formal reasons, which are often called “complexity barriers.” These are theorems which demonstrate that the usual tools for reasoning about computability and lower bounds—such as universal simulation—are simply too abstract to distinguish modes of computation like P and NP. There are three major classes of barriers known.

Relativization, Algebrization, Natural Proofs. Many ways of reasoning about algorithm complexity remain valid when one adds “oracles” to the computational model: that is, one adds an instruction that can call an *arbitrary* function $\mathcal{O} : \{0,1\}^\star \to \{0,1\}$ in one step, as a black box. When a proof of a theorem is true no matter which $\mathcal{O}$ is added to the instruction set of the relevant computational model(s), we say that the proof “relativizes.” Relativizing proofs of statements are generally quite powerful and broadly applicable. However, relativizing proofs are of limited use in computational complexity lower bounds: for instance, $\mathsf{P} = \mathsf{NP}$ when some oracles $\mathcal{O}$ are added to polynomial-time and nondeterministic-polynomial-time algorithms (picking a powerful enough oracle will do), but $\mathsf{P} \neq \mathsf{NP}$ when some other oracles $\mathcal{O}'$ are added (as proved by Baker, Gill, Solovay [Baker et al. 1975]). Therefore, no proof resolving P versus NP can be a relativizing proof. Practically all other major open problems in lower bounds exhibit a similar resistance to arbitrary oracles, and a surprisingly large fraction of theorems in complexity theory do relativize.

The more recently developed “algebrization” barrier [Aaronson and Wigderson 2009] teaches a similar lesson, applied to a broader set of algebraic techniques that was designed to get around relativization. (Instead of looking at oracles, they look at more general algebraic objects, where an oracle O that takes n-bit inputs can be “lifted” to a low-degree polynomial $\tilde{\mathcal{O}} : \mathbb{F}^n \to \mathbb{F}$ over a field $\mathbb{F}$ of order greater than two.) These barriers teach us that, in order to make progress on problems resembling P versus NP, it will be necessary to study the guts of programs at a somewhat low level, and reason more closely about their abilities relative to their simple instruction sets.[2]

The Razborov-Rudich “natural proofs” barrier [Razborov and Rudich 1997] has a more subtle pedagogical point compared to the other two. Informally, they show that strong lower bound proofs *cannot* produce a polynomial-time algorithm for determining whether a given function is hard or easy to compute—otherwise, such an algorithm would (in a formal sense) refute *stronger* lower bounds that we also believe to hold. It turns out that many lower bound proofs from the 1980s and 1990s have such an algorithm embedded in them.

1.2. Intuition and Counter-Intuition

There are also strong intuitive reasons for why lower bounds are hard to prove. The most common one is that it seems extraordinarily difficult to reason effectively about the infinite set of all possible efficient programs, including programs that *we will never see or execute*, and argue that none of them can solve an NP-complete problem. Based on this train of thought, some famous computer scientists such as Donald Knuth have dubbed problems like P versus NP to be “unknowable” [Knuth 2015].

But how difficult is it, really, to reason about all possible efficient programs? Let us give some counter-intuition, which will build up to the main point of this article. We begin with the observation that, while reasoning about lower bounds appears to be difficult, reasoning about *worst-case algorithms* does not appear to be. Reasoning computationally about an infinite number of finite objects is commonplace in the analysis

[2]There are definitely “non-relativizing” and even “non-algebrizing” techniques in complexity theory, but they are a minority; see Section 3.4 in Arora and Barak [Arora and Barak 2009] for more discussion.

of worst-case algorithms. There, we have some computable function $f : \Sigma^\star \to \Sigma^\star$ that we would like to compute faster, and one proves that a particular efficient procedure P outputs $f(x)$ on *all possible* finite inputs x. That is, often in algorithm analysis we manage to reason about all possible finite inputs x, even those x that we will never see or encounter in the real world. Our idea is that, if we can find "meta-computational" problems which

(a) treat their inputs as *programs*,
(b) determine interesting properties of the function computed by the input program, and
(c) have interesting/non-trivial algorithms,

then we can hope to import ideas from the design and analysis of algorithms into the theory of complexity lower bounds. (Yes, this is vague, but it is counter-*intuition*, after all.)

Sanity Check: Computability Theory. We must be careful with this counter-intuition. Which computational problems actually satisfy those three conditions? Undergraduate computability theory (namely, Rice's theorem [Rice 1953]) tells us that, if our inputs x encode programs that take arbitrarily long inputs, then it is undecidable to determine non-trivial properties of the function being computed by the program x. Thus, it would seem that any problem that satisfies conditions (a) and (b) above will fail to satisfy condition (c).

One source of this undecidability is the "arbitrary length" of inputs. In particular, if x encodes a program that takes only *finitely* many inputs, say only inputs of length n, then one can produce the entire finite function computed by the input x, and decide non-trivial properties of that function in a computable way. To simplify the discussion (and without significantly losing generality), we might as well think of the input x as encoding a Boolean logic circuit over AND, OR, and NOT gates, taking some n bits of input and outputting some m bits. Now, x simply encodes a directed acyclic graph with additional labels on its nodes, and a procedure P operating on x's can be said to be a "circuit analysis" program, which reasons about the aggregate behavior of finite circuits on their inputs.

However, one of the major lessons of the theory of NP-hardness is that, while reasoning about arbitrary programs may be undecidable, reasoning about arbitrary circuits *is* often decidable but is still highly likely to be intractable. Probably the simplest possible circuit analysis problem is:

Given a Boolean circuit C, does C compute the all-zeroes function?

This problem is already very difficult to solve; it is equivalent to the NP-complete problem CIRCUIT SATISFIABILITY (a.k.a. Circuit SAT) which asks if there is some input on which C outputs 1. From this point of view, the assertion $\mathsf{P} \neq \mathsf{NP}$ tells us that arbitrary programs are intractable to analyze, even over finitely many inputs: we cannot feasibly determine if a given circuit is *trivial* or not. As circuit complexity is inherently tied to P versus NP, the assertion $\mathsf{P} \neq \mathsf{NP}$ appears to have negative consequences for its own provability; this looks depressing. (This particular intuition has been proposed many times before; for instance, the Razborov-Rudich work on "natural proofs" [Razborov and Rudich 1997] may be viewed as one way to formalize it.)

Slightly Faster SAT Algorithms? The hypothesis $\mathsf{P} \neq \mathsf{NP}$ only says that *very* efficient circuit analysis is impossible. More precisely, for circuits C with k bits of input encoded in n bits, $\mathsf{P} \neq \mathsf{NP}$ means there is no $k^{O(1)} \cdot n^{O(1)}$ time algorithm for detecting if C is satisfiable. There is a giant gap between this kind of bound and the $2^k \cdot n^{O(1)}$ time bound obtained by simple exhaustive search over all 2^k possible inputs to the circuit.

What if we simply asked for a *non-trivial running time* for detecting the satisfiability of C, something that is merely faster than exhaustive search?

I would like to argue that finding any asymptotic improvement over 2^k time for CIRCUIT SATISFIABILITY is already a very interesting problem. This is not an obvious point to argue. First off, without any further knowledge of its inner workings, a $1.9^k \cdot n^{O(1)}$-time algorithm for CIRCUIT SATISFIABILITY would not be terribly more useful in *practice* than the $2^k \cdot n^{O(1)}$ time of exhaustive search: one would only see a different for small values of k, and the rest of the instances would remain intractable.[3] Work on worst-case algorithms for SAT over many years (such as [Monien and Speckenmeyer 1979; Dantsin 1981; Monien and Speckenmeyer 1985; Kullmann and Luckhardt 1997; Pudlák 1998; Hirsch 2000; Schöning 2002; Paturi et al. 2005; Wahlström 2007; Chen et al. CC14], see the survey [Dantsin and Hirsch 2009]) has been primarily motivated by the intrinsic interest in understanding whether trivial exhaustive search is optimal for solving the Satisfiability problem.

The Non-Black-Box-ness of Circuit SAT Algorithms. There is also a deeper reason to pursue minor improvements in SAT algorithms. Any algorithm for CIRCUIT SAT running in (say) time $1.9^k \cdot n^{O(1)}$ must necessarily provide a kind of "non-relativizing" analysis of *every* given circuit C, an analysis which relies on the structure and encoding of C. If you were required to design a CIRCUIT SAT algorithm which could only access C as a black-box *oracle*, obtaining the output $C(y)$ from inputs y and no other information about C, then your algorithm would necessarily require at least 2^k steps in the worst case. The reason is simple: if you are completely blind to the insides of the circuit C, then even a small ($O(k)$ size) circuit could hide a satisfying k-bit input from you.

In more detail, note there is a trivial circuit Z which always outputs 0, and for every k-bit input y, there is an $O(k)$-size circuit Z_y which outputs 1 if and only if the input equals y. Given the code of your black-box SAT algorithm A, one can note the sequence of k-bit inputs $y_1, y_2, \ldots$ that A calls the oracle on, assuming that the oracle circuit keeps outputting 0. At the end of the execution of A, if it has only seen 0-outputs, then A must conclude that the circuit is not satisfiable (otherwise, it would give the wrong answer on the all-zeroes circuit). Furthermore, if A did not query the oracle on some k-bit input y, then the circuit Z_y will be satisfiable and yet A will have concluded it is unsatisfiable. It follows that, for A to be correct on all circuits Z and Z_y, it needs to call the oracle circuit on all 2^k possible inputs.

Therefore, a $1.9^k \cdot n^{O(1)}$ time algorithm for CIRCUIT SAT must necessarily use the fully-given representation of the circuit in some critical ways, to work faster than exhaustive search. Even an algorithm running in $O(2^k/k)$ time on circuits of size $O(k)$ would be interesting, for the same reason.[4]

A Possible Road to Circuit Complexity Lower Bounds. The ability to analyze a given circuit more efficiently than analyzing a black box suggests a further implication: a CIRCUIT SAT algorithm running faster than exhaustive search could potentially be used to prove a circuit complexity *limitation*. At the very least, if the CIRCUIT SAT

[3]This attitude is not shared in cryptography, where any improvement in exhaustive search over all keys may be considered a "break" in the cryptosystem.

[4]Perhaps you do not believe that CIRCUIT SAT can be solved any faster than $2^k \cdot n^{O(1)}$ steps. This belief turns out to be inessential for the main intuition and the formal theorems that follow. For example, you may instead believe that we can non-deterministically approximate the fraction of satisfying assignments to a k-input circuit of size n in $1.9^k \cdot n^{O(1)}$ time; this is also something that oracle access to a circuit cannot accomplish. Furthermore, if you do not believe even this, then your lack of faith in algorithms requires you to have strong beliefs in the power of Boolean circuits—they would be powerful enough to solve nondeterministic exponential-time problems. See the references [Impagliazzo et al. 2002; Williams 2013a].

problem can be solved faster than exhaustive search on a given collection of circuits $\mathcal{C}$ (some of which encode the all-zero function, and some which do not), then the collection $\mathcal{C}$ *fails* to obfuscate the all-zeroes function from some algorithm running in less than 2^k steps. That is, the assumed CIRCUIT SAT algorithm can "efficiently" distinguish all circuits encoding the all-zeroes function from those circuits which do not; these circuit cannot hide satisfying inputs as well as oracles can. This points to a potential deficiency in $\mathcal{C}$ that the CIRCUIT SAT algorithm takes advantage of. Surprisingly, this intuitive viewpoint can be made formal.

Outline. In the remainder of this article, we first describe some known connections between circuit satisfiability algorithms and circuit complexity lower bounds (Section 2). Then, we turn to a more recent example of how algorithms and lower bounds are tied to each other, in a way that we believe should be of interest to the union of logicians and computer scientists (Section 3). In particular, we reconsider the basic problem of testing circuit functionality via input-output examples, define the *test complexity* as a way of measuring the difficulty of testing, and describe how circuit complexity lower bounds are *equivalent* to test complexity upper bounds. We conclude the article with some hopeful thoughts.

2. CIRCUIT SAT VERSUS CIRCUIT COMPLEXITY

Let us briefly review some relevant notions from the theory of circuit complexity; for more, see the textbook of Arora and Barak ([Arora and Barak 2009], Chapter 6).

Circuit Complexity. A Boolean circuit with n inputs and one output is a directed acyclic graph with n sources and one sink, and labels of AND/OR/NOT on all other nodes. Each circuit computes some finite function $f : \{0,1\}^n \to \{0,1\}$. To compute infinite languages, of the form $L : \{0,1\}^\star \to \{0,1\}$, the computational model is extended to an infinite family of circuits $\mathcal{F} = \{C_n\}_{n=1}^{\infty}$, where C_n has n inputs and one output. For such a family $\mathcal{F}$, we say that $\mathcal{F}$ *computes* L if for all $x \in \{0,1\}^\star$ we have $C_{|x|}(x) = L(x)$. For a function $s : \mathbb{N} \to \mathbb{N}$, a family $\mathcal{F}$ has *size* $s(n)$ if for all n, the number of nodes (i.e., gates) in C_n is at most $s(n)$. A language L *has polynomial-size circuits* if there is a polynomial $p(n)$ and a family $\mathcal{C}$ of size $p(n)$ that computes L. The class of all languages having polynomial-size circuits is denoted by P/poly. One should think of P/poly as the class of computations for which the minimum "sizes" of computations do not grow considerably with the input length to those computations.

The class P/poly is poorly understood. It could be enormously powerful, or it could be fairly weak. It is easy to see that every language of the form $\{1^n \mid n \in S\}$ (for some subset $S \subseteq \mathbb{N}$) is in P/poly; however, a simple counting argument shows there are undecidable languages of this form. Therefore P/poly contains some undecidable languages. In that sense, P/poly is powerful, but this comes from the fact that the computational model defining P/poly can have infinite-length descriptions. (This observation also shows that traditional thought in computability theory is probably not going to be very helpful in understanding the power of P/poly.) However, P/poly also looks obviously limited, in the sense that for all input lengths n, only polynomial-in-n resources need to be spent in order to decide all 2^n inputs of that length. A counting argument shows that for every n, some function $f : \{0,1\}^n \to \{0,1\}$ requires circuits of size exponential-in-n; in fact, *most* functions do.

A prominent question in complexity theory is:

How does P*/poly relate to the Turing-based classes of classical complexity theory, like* P*,* NP*,* PSPACE*, etc.?*

It is pretty easy to see that $\mathsf{P} \subset \mathsf{P}$/poly: every "finite segment" of a polynomial-time algorithm can be simulated with a polynomial-size circuit. It is conjectured that $\mathsf{NP} \not\subset$

P/poly (which would in turn imply P $\neq$ NP). But it is an open problem to prove that NEXP $\not\subset$ P/poly! That is, every language in the *exponential-time version of* NP could in fact have polynomial-size circuits. It is amazing that a problem like this is still open. Fifty years ago, Hartmanis and Stearns [Hartmanis and Stearns 1965] showed that some $O(n^3)$-time computations are more powerful than all $O(n)$-time ones; how is it possible that we can't distinguish exponential time Turing machines from polynomial size circuits? This open problem demonstrates how truly difficult it is to prove lower bounds on circuit complexity; perhaps the infinite circuit model is powerful!

2.1. Enter Circuit SAT

Let's return to thinking about the role of CIRCUIT SATISFIABILITY. Earlier, we were arguing that a faster algorithm for Circuit SAT would point to some deficiency in the power of circuits: unlike a black-box oracle, circuits would be unable to successfully "hide" satisfying inputs from algorithms that run in $o(2^k)$ steps. We would like to say:

The *existence* of a "faster" algorithm A solving CIRCUIT SAT,
for *all* circuits C from a class of circuits $\mathcal{C}$

$$\Longrightarrow$$

The *existence* of an "interesting" function $f : \{0,1\}^\star \to \{0,1\}$,
that is not computable by *all* circuit families from that class $\mathcal{C}$

Written in the above way, the logical quantifiers match up well, and the idea of using an algorithm to prove a circuit complexity lower bound looks less counter-intuitive. Indeed, we can say a formal statement as described in the above box. Here is one version.

THEOREM 2.1 ([WILLIAMS 2013A; 2014B]). *Suppose for all polynomials p,* Circuit Satisfiability *of circuits with n inputs and $p(n)$ size is decidable in $O(2^n/n^{10})$ time. Then* NEXP $\not\subset$ P*/poly. That is, there are (explicit) functions computable in nondeterministic exponential time that do not have polynomial-size circuits.*

(The polynomial n^{10} is almost certainly not optimal, and it depends on the precise machine model, but it suffices.) Notice the required improvement over exhaustive search: it would normally take $2^n \cdot p(n)$ time to solve SAT on circuits of $p(n)$ size. In order to solve satisfiability fast enough, we need to be able to "divide by an arbitrary polynomial" in the running time. This is a much weaker requirement than bounds like 1.9^n time, which had been the primary focus of researchers.

Here we will briefly describe the proof of Theorem 2.1. For more technical details, see the original paper [Williams 2013a] and the surveys [Williams 2011; Santhanam 2012; Oliveira 2013].

Proof Outline of Theorem 2.1. The known proofs proceed by contradiction. We assume:

(A) There is a CIRCUIT SAT algorithm A running in $2^n/n^{10}$ time, and
(B) Every function $f \in$ NEXP has polynomial-size circuits. (Note that it is equivalent to assume that a single function, complete for NEXP under polynomial-time reductions, is computable with polynomial-size circuits.)

These two assumptions are inherently algorithmic in nature: item (A) asserts that CIRCUIT SATISFIABILITY can be solved faster, and item (B) asserts that a huge class of decidable problems are computable with polynomial-size circuit families.

The main idea is to utilize these two algorithmic assumptions to solve NEXP-complete problems in a way which is provably *impossible*. Namely, assumptions (A) and (B) together are shown to imply that *every* function computable in nondeterministic time $O(2^n)$ is computable in nondeterministic time $O(2^n/n^2)$, which contradicts the time hierarchy theorem for nondeterminism [Žák 1983].

Compressible Witnesses for NEXP.. Our proof uses a highly non-trivial consequence of item (B). Recall that NEXP consists of those problems L for which there is a verifier algorithm V which runs in time exponential in $|x|$ so that for all inputs x, $x \in L$ if and only if there is a witness y of length exponential in $|x|$ such that $V(x,y)$ accepts. (The definition of NP would replace "exponential" with "polynomial.")

Work of Impagliazzo, Kabanets, and Wigderson [Impagliazzo et al. 2002] shows an important consequence of item (B): if NEXP is in P/poly, then all exponential-time verifiers have *highly compressible* witnesses. More precisely, they prove that item (B) implies:

(C) For all $L \in$ NEXP and all *exponential-time verifiers V for* L, for every $x \in L$, there is a poly($|x|$)-size circuit C_x with poly($|x|$) inputs with truth table y such that $V(x,y)$ accepts.[5]

In other words, every string x in L has a witness that has a very succinct representation, of only poly(n) size.

The Impossible Algorithm.. Let L be an arbitrary language that is computable in nondeterministic 2^n time. We want to construct another nondeterministic algorithm that computes L in $O(2^n/n^2)$ time, which implies the desired contradiction (as mentioned above). Here is a high-level outline of the nondeterministic algorithm:

Algorithm Outline: On input x,

(1) Nondeterministically *guess* poly(n)-size circuit C_x which is intended to encode a witness for x (Note that if $x \in L$ then such a C_x exists, by item (C).)
(2) Deterministically *verify* that C_x is a correct guess. Here we wish to use the CIRCUIT SAT algorithm asserted by item (A).

Clearly, Step 1 takes poly(n) time to guess the circuit C_x. We need to find conditions for which Step 2 will run in $O(2^n/n^2)$ time, assuming the CIRCUIT SAT algorithm of item (A).

The deterministic verification of Step 2 is a subtle process. The CIRCUIT SAT algorithm of item (A) can check in $O(2^n/n^{10})$ time whether a given n-input circuit always outputs 0, over all possible 2^n input assignments. A key observation is that it would suffice to build a larger circuit D_x with $m \leq n + 8\log n$ inputs and poly(n) size (in $O(2^n/n^2)$ time) that is based on C_x, such that C_x encodes a witness for x if and only if D_x is not satisfiable. Then, running the CIRCUIT SAT algorithm of item (A) on D_x takes time

$$O\left(2^m/m^{10}\right) \leq O\left(2^{n+8\log(n)}/(n+8\log n)^{10}\right) \leq O\left(2^n/n^2\right),$$

and would be unsatisfiable if and only if C_x encodes a witness for x, as desired. Hence, if we could build such a D_x, then we could implement step 2 of the above algorithm outline in time $O(2^n/n^2)$.

[5]We use the notation poly(n) to denote an arbitrary fixed polynomial factor of n.

How to Use the Circuit SAT Algorithm.. So how can we build a circuit D_x which is unsatisfiable if and only if C_x encodes a witness for x? We turn to special structural properties of NEXP computations. In particular, every language L computable in nondeterministic $O(2^n)$ time can be (very) efficiently reduced to the following NEXP-complete problem:

SUCCINCT 3SAT: *Given a circuit E of n inputs, does its truth table of 2^n bits encode a satisfiable 3CNF formula?*

In particular, as we evaluate E on various inputs, the outputs encode *clauses* of an exponentially-large 3CNF formula. Sharp technical results on the NP-hardness of 3SAT [Fortnow et al. 2005; Williams 2013a] show how to reduce any nondeterministic $O(2^n)$-time L to a poly(n)-size instance of SUCCINCT 3SAT with at most $n + 4\log n$ inputs. That is, we have a polynomial-time algorithm A that, given an instance x, outputs a circuit E_x which is a yes-instance of SUCCINCT 3SAT if and only if $x \in L$.

From here, the desired circuit D_x can be constructed in the following way. Note that the witness circuit C_x guessed for an instance of SUCCINCT 3SAT will encode a variable assignment. On an input y of length $n + 4\log n$, our circuit D_x:

— Prints the "yth clause" of the 3CNF formula encoded by E_x.
— Uses three copies of the guessed circuit C_x to verify if at least one of the three literals in the yth clause is satisfied by the assignment encoded by C_x.
— Outputs 0 if and only if the yth clause is indeed satisfied.

Then, such a D_x is unsatisfiable if and only if C_x encodes a satisfying assignment to the 3CNF formula encoded by E_x. This completes the proof outline of Theorem 2.1. □

While the above proof yields the desired outcome, it is obviously an indirect method, and feels lacking. It is an interesting open problem to find simpler and/or more informative proofs of this algorithms-to-lower-bounds connection.

Applying the Connections. The framework behind Theorem 2.1 has been generalized so that circuit satisfiability algorithms for various circuit classes $\mathcal{C}$ imply lower bounds for computing functions in NEXP with circuits from $\mathcal{C}$. So far, through the design of new circuit satisfiability algorithms, this framework has led to several unconditional circuit lower bound results:

— NEXP does not have so-called ACC^0 circuits of polynomial size [Williams 2014b],
— NE/1∩coNE/1 (a potentially weaker class) does not have ACC^0 circuits of polynomial size [Williams 2013b].
— NEXP does not have ACC^0 circuits of polynomial size, augmented with a layer of neurons (linear threshold gates) that connect directly to the inputs [Williams 2014a].
— E$^{\mathsf{NP}}$ does not have sub-quadratic size circuits composed of arbitrary symmetric and linear threshold gates [Tamaki 2016].
— E$^{\mathsf{NP}}$ does not have ACC^0 circuits of $2^{n^{o(1)}}$ size, augmented with a layer of $n^{2-\varepsilon}$ neurons (linear threshold gates) that connect to another layer of $2^{n^{o(1)}}$ neurons, and this second layer connects directly to the inputs [Alman et al. 2016].

All results except for the second were obtained by designing explicit circuit satisfiability algorithms for the relevant circuit classes; the second was obtained by sharpening the complexity-theoretic arguments themselves.

One might not believe that even slightly faster circuit satisfiability algorithms are possible. We stress that it is possible that one might prove that $\mathsf{NEXP} \not\subset \mathsf{P}/\text{poly}$ without providing a new CIRCUIT SAT algorithm. There are at least two ways in which this could be done:

1. Instead of solving the NP-hard CIRCUIT SAT problem, it is possible to show [Williams 2013a; Santhanam and Williams 2014] that one only needs to deterministically solve the following CIRCUIT APPROXIMATION PROBABILITY PROBLEM (CAPP):

Given a circuit C that is promised to have either at least 2^{n-1} satisfying assignments, or zero satisfying assignments, determine which is the case.

With randomness, it is trivial to distinguish between the two cases with high probability, even in a black-box way: just pick uniform random inputs and check if any of them make C output 1. By using succinct probabilistically checkable proofs in place of SUCCINCT 3SAT [Ben-Sasson et al. 2005; Ben-Sasson and Viola 2014], a deterministic solution to CAPP running in $2^n/n^{10}$ time on circuits of polynomial size would also imply $\mathsf{NEXP} \not\subset \mathsf{P}/\text{poly}$.

2. It could be that the *assumption* $\mathsf{NEXP} \subset \mathsf{P}/\text{poly}$ could be used to imply the existence of satisfiability algorithms sufficient for proving $\mathsf{NEXP} \not\subset \mathsf{P}/\text{poly}$. (It is in fact known that $\mathsf{NEXP} \subset \mathsf{P}/\text{poly}$ implies faster algorithms for solving some NP problems, but CIRCUIT SAT is not known to be among them!)

3. CIRCUIT COMPLEXITY AND TESTING CIRCUITS WITH DATA

We now turn to a problem related to program verification, and describe a connection to circuit complexity. In practice, programs are often verified by the quick-and-dirty method of trial and error: the program is executed on a suite of carefully chosen inputs, and one checks that the outputs of the program are what is expected. For a given function to compute, it is natural to ask when trial and error can be efficient: when does can one use a small number of input-output examples, and determine correctness of the program with certainty?

If there are no constraints on what the program can be, then there is not much to say about this problem: without any further information, the program is a black box, and one would simply have to try all possible inputs to know whether the program does what it should. But in testing, we're never given a program as a black box; we know *something* about it, such as its total size. Could such side information be useful for the testing problem? Formal mathematical work on this kind of problem from years ago (such as [Howden 1976; DeMillo and Lipton 1978; Budd and Angluin 1982]) focused on restrictions to the functions to be computed and how to efficiently generate test data for them, rather than including more side information about the programs themselves to make the testing problem less black-box.

Recently with Brynmor Chapman [Chapman and Williams 2015], we have proposed a general circuit-analysis problem that we call *data design*. Let $f : \{0,1\}^\star \to \{0,1\}$ be a function to test for, and a class $\mathcal{C}$ of size-s circuits that can implement "slices" of f (restricted to fixed-length inputs). The task of *data design for f* is to select a small suite of input-output *test data* that can be used to determine whether a given n-input circuit C from $\mathcal{C}$ computes f restricted to n-bit inputs.

More formally, we say that the *test complexity* of f (as a function of the circuit size s) is the minimum number of input-output examples such that, for any circuit C of size s, one can determine with certainty whether C computes f (on all n-bit inputs, where

n is the number of inputs to C), by evaluating the circuit C on the examples. Note that every function f depending on all of its inputs has test complexity $O(s2^s)$: the test suite may contain all possible input-output pairs for f on all input lengths $n = 1, \ldots, s$. When can test suites be made smaller?

While the data design problem certainly has practical motivation, we are mainly interested in the problem due to its intriguing inversion of the roles of program and input. The circuit C computing f is the *input* to the data design problem, and the *program* for testing the circuit is the collection of input-output examples needed to determine if C computes a slice of f. We have uncovered a surprising correspondence between upper bounds on data design and lower bounds on circuit complexity. In general:

Designing small suites of data for testing whether $\mathcal{C}$-circuits compute f
is *equivalent* to
Proving $\mathcal{C}$-circuit lower bounds on computing f.

For example:

THEOREM 3.1 ([CHAPMAN AND WILLIAMS 2015]). *A function f is in* P*/poly if and only if for some $\varepsilon > 0$, the test complexity of testing circuits for f is greater than 2^{s^ε} for almost every s.*

So if we wanted to prove that (for example) that NEXP $\not\subset$ P/poly, it would be necessary and sufficient to design test suites of sub-exponential test complexity for a function in NEXP. Intuitively, such a correspondence is possible because the circuit design problem and the data design problems work with similar types of unknowns. The circuit C designed must compute f on all n-bit inputs, and the set of data designed must test the functionality of f for all s-size circuits C. Let us outline the ideas of the proof.

Proof Outline of Theorem 3.1. There are two parts to the equivalence.

Circuit Complexity Upper Bounds Imply Test Complexity Lower Bounds.. Let $k \geq 1$. If the function f has a circuit of size n^k on n-bit inputs, then a test suite for size-s circuits must include at least $2^{\Omega(s^{1/k})}$ input-output pairs. That is, an upper bound on the circuit complexity of f implies a *lower bound* on the test complexity of f.

To prove this, we observe a couple of simple facts. First, if f has a circuit of size n^k on n-bit inputs, then there is some circuit C of size s that computes f on inputs of length about $s^{1/k}$. Second, for every input y of length $s^{1/k}$, there is a circuit C_y of about $s+O(s^{1/k})$ size which equals C on all inputs except for y. From these two facts, one can show that the test suite for size-s circuits will need to include all possible input-output pairs for f on $(\delta \cdot s^{1/k})$-bit inputs (for some $\delta > 0$), in order to distinguish the good size-s circuit C from all of the other (bad) C_y circuits. That is, the test suite needs at least $2^{\delta s^{1/k}}$ input-output pairs.

Circuit Complexity Lower Bounds Imply Test Complexity Upper Bounds.. For the other side of the equivalence, we show that if f does not have a circuit of size $O(n^{k+1})$ on n-bit inputs, then there is a set of only poly(n^k) input-output pairs $(x, f(x))$ on n-bit inputs that is sufficient for testing all circuits of size up to n^k. (This statement is something of a "dual" to a result in learning theory [Bshouty et al. 1996] on learning small circuits with an NP oracle.) Then, a lower bound on the circuit complexity of f

for circuits of size n^{k+1} ensures that the test complexity for f is not too large for testing circuits of size n^k.

The above result applies theorems on "sparse" strategies for zero-sum games. (Recall that for zero-sum games, a *strategy* is just a probability distribution on the set of possible actions for a player.) Such theorems [Althöfer 1994; Lipton and Young 1994] say that for every zero-sum game where the "row player" has m possible actions and the "column player" has n possible actions, there is a strategy for the row player with only $O(\log n)$-size support and for the column player with only $O(\log m)$-size support, that closely approximates the optimal strategy of the game.

To apply such results here, the key idea is to consider a zero-sum game with a Circuit Player (ranging over all circuits of size up to n^k) and an Input Player (ranging over all inputs of length up to n). If f does not have a circuit of size $O(n^{k+1})$ on n-bit inputs, then for every tuple $(C_1, \ldots, C_{O(n)})$ of n^k-size circuits that could form a sparse strategy for the Circuit Player, there is an input $x^\star$ such that $f(x^\star) \neq MAJORITY(C_1(x^\star), \ldots, C_{O(n)}(x^\star))$. That is, every sparse strategy of the Circuit Player badly fails to compute f on at least one input $x^\star$: for every sparse strategy, a random choice of a circuit from the strategy fails to compute f on $x^\star$ with probability very close to $1/2$.

As every sparse strategy for the Circuit Player badly fails to compute f, one can show that there must be a good sparse strategy for the Input Player: there is a poly(n^k)-size set S of input-output pairs $(x, f(x))$ such that for every circuit C of size up to n^k, there is an $(x, f(x)) \in S$ such that $C(x) \neq f(x)$. This set S is precisely the test suite that we wanted to obtain.

Let f_n be the restriction of f to inputs of length n. Putting the two above items together, we (informally) have

$$\begin{aligned} f \in \mathsf{P}/\text{poly} &\iff (\exists k)(\forall n)[f_n \text{ has } n^k\text{-size circuits}] \\ &\iff (\exists k)(\forall s)[\text{the test complexity of } f \text{ is } 2^{\Omega(s^{1/k})}] \end{aligned}$$

This concludes the proof outline. □

Designing reliable exhaustive circuit tests and circuit complexity lower bounds are therefore deeply related tasks, when phrased in the above language. Not only would small test suites detect errors efficiently, they would also be useful for formal verification. Assuming the circuit being tested is in the appropriate class $\mathcal{C}$, passing a test suite would be a proof of correctness on all inputs. In turn, proving that a small test suite (relative to the circuit size) works is equivalent to proving a limitation on what can be computed in $\mathcal{C}$. This "constructive" algorithmic viewpoint on lower bounds is still in its early stages of development, and it remains to be seen how effectively one can prove lower bounds with it.

4. CONCLUSION

Knowledge from all areas of theoretical computer science could contribute significantly to the general projects outlined here. Computer scientists will have to develop new methods of argument in order to make a serious dent in the major lower bound problems, and it is worth trying every sort of reasonable argument we can think of (at least once). Perhaps the logic side of computer science will provide some of these new proof methods.

REFERENCES

Scott Aaronson and Avi Wigderson. 2009. Algebrization: A New Barrier in Complexity Theory. *ACM TOCT* 1 (2009). Issue 1.

Josh Alman, Timothy M. Chan, and R. Ryan Williams. 2016. Polynomial Representations of Threshold Functions and Algorithmic Applications. In *FOCS*. 467–476.

Ingo Althöfer. 1994. On sparse approximations to randomized strategies and convex combinations. *Linear Algebra Appl.* 199 (1994), 339–355.

Sanjeev Arora and Boaz Barak. 2009. *Computational Complexity - A Modern Approach*. Cambridge University Press.

Theodore Baker, John Gill, and Robert Solovay. 1975. Relativizations of the P =? NP Question. *SIAM J. Comput.* 4, 4 (1975), 431–442.

Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. 2005. Short PCPs Verifiable in Polylogarithmic Time. In *IEEE Conference on Computational Complexity (CCC*. 120–134.

Eli Ben-Sasson and Emanuele Viola. 2014. Short PCPs with projection queries. In *ICALP*. 163–173.

Nader Bshouty, Richard Cleve, Ricard Gavalda, Sampath Kannan, and Christino Tamon. 1996. Oracles and Queries that are Sufficient for Exact Learning. *J. Comput. System Sci.* 52, 2 (1996), 268–286.

Timothy A. Budd and Dana Angluin. 1982. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica* 18 (1982), 31–45.

Brynmor Chapman and Ryan Williams. 2015. The Circuit-Input Game, Natural Proofs, and Testing Circuits With Data. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS*. 263–270.

Ruiwen Chen, Valentine Kabanets, Antonina Kolokolova, Ronen Shaltiel, and David Zuckerman. 2015. See CCC'14. Mining Circuit Lower Bound Proofs for Meta-Algorithms. *Computational Complexity* 24, 2 (2015. See CCC'14), 333–392.

Evgeny Dantsin. 1981. Two propositional proof systems based on the splitting method. *Zapiski Nauchnykh Seminarov LOMI* 105 (1981), 24–44.

Evgeny Dantsin and Edward A. Hirsch. 2009. Worst-Case Upper Bounds. In *Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications*. Vol. 185. IOS Press, 403–424.

Richard A. DeMillo and Richard J. Lipton. 1978. A probabilistic remark on algebraic program testing. *Inform. Process. Lett.* 7, 4 (1978), 192–195.

Andrzej Ehrenfeucht. 1961. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae* 49 (1961), 129–141.

Lance Fortnow, Richard J. Lipton, Dieter van Melkebeek, and Anastasios Viglas. 2005. Time-space lower bounds for satisfiability. *JACM* 52, 6 (2005), 835–865.

Roland Frässé. 1950. Sur une nouvelle classification des systmes de relations. *Comptes Rendus* 230 (1950), 1022–1024.

Juris Hartmanis and Richard Stearns. 1965. On the Computational Complexity of Algorithms. *Trans. Amer. Math. Soc. (AMS)* 117 (1965), 285–306.

Edward A. Hirsch. 2000. New Worst-Case Upper Bounds for SAT. *J. Autom. Reasoning* 24, 4 (2000), 397–420. DOI:http://dx.doi.org/10.1023/A:1006340920104

W. E. Howden. 1976. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering SE-2* 3 (1976), 208–214.

Russell Impagliazzo, Valentine Kabanets, and Avi Wigderson. 2002. In search of an easy witness: Exponential time vs. probabilistic polynomial time. *J. Comput. Syst. Sci.* 65, 4 (2002), 672–694.

Donald E. Knuth. 2015. Personal communication. (2015).

Oliver Kullmann and Horst Luckhardt. 1997. Deciding propositional tautologies: Algorithms and their complexity. Technical report, Fachbereich Mathematik, Johann Wolfgang Goethe Universität. (1997).

Richard J. Lipton and Neal E. Young. 1994. Simple strategies for large zero-sum games with applications to complexity theory. 734–740.

Burkhard Monien and Ewald Speckenmeyer. 1979. 3-satisfiability is testable in $O(1.62^r)$ steps. Technical Report Bericht Nr. 3/1979, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn. (1979).

Burkhard Monien and Ewald Speckenmeyer. 1985. Solving Satisfiability in Less Than 2^n Steps. *Discrete Applied Mathematics* 10, 3 (1985), 287–295.

Ketan Mulmuley. 2012. The GCT program toward the *P* vs. *NP* problem. *Commun. ACM* 55, 6 (2012), 98–107. DOI:http://dx.doi.org/10.1145/2184319.2184341

Igor Oliveira. 2013. *Algorithms versus Circuit Lower Bounds*. Technical Report TR13-117. ECCC.

Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. 2005. An improved exponential-time algorithm for k-SAT. *JACM* 52, 3 (2005), 337–364.

Pavel Pudlák. 1998. Satisfiability - Algorithms and Logic. In *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, (MFCS'98)*. 129–141. DOI: http://dx.doi.org/10.1007/BFb0055762

Alexander Razborov and Steven Rudich. 1997. Natural Proofs. *J. Comput. Syst. Sci.* 55, 1 (1997), 24–35.

H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

Rahul Santhanam. 2012. Ironic Complicity: Satisfiability Algorithms and Circuit Lower Bounds. *Bulletin of the EATCS* 106 (2012), 31–52.

Rahul Santhanam and Ryan Williams. 2014. On Uniformity and Circuit Lower Bounds. *Computational Complexity* 23, 2 (2014), 177–205.

Uwe Schöning. 2002. A Probabilistic Algorithm for k-SAT Based on Limited Local Search and Restart. *Algorithmica* 32, 4 (2002), 615–623.

Suguru Tamaki. 2016. A Satisfiability Algorithm for Depth Two Circuits with a Sub-Quadratic Number of Symmetric and Threshold Gates. *Electronic Colloquium on Computational Complexity (ECCC)* 23 (2016), 100.

Magnus Wahlström. 2007. *Algorithms, Measures, and Upper Bounds for Satisfiability and Related Problems*. Ph.D. Dissertation. Linköping University.

Ryan Williams. 2011. Guest column: a casual tour around a circuit complexity bound. *ACM SIGACT News* 42, 3 (2011), 54–76.

Ryan Williams. 2013a. Improving Exhaustive Search Implies Superpolynomial Lower Bounds. *SIAM J. Comput.* 42, 3 (2013), 1218–1244.

Ryan Williams. 2013b. Natural Proofs Versus Derandomization. In *STOC*. 21–30.

Ryan Williams. 2014a. New algorithms and lower bounds for circuits with linear threshold gates. In *STOC*. 194–202.

Ryan Williams. 2014b. Nonuniform ACC Circuit Lower Bounds. *JACM* 61, 1 (2014), 2.

Stanislav Žák. 1983. A Turing machine time hierarchy. *Theoretical Computer Science* 26, 3 (1983), 327–333.