Thinking Algorithmically About Impossibility*

R. Ryan Williams

Computer Science Department, Stanford University
Stanford, CA, USA
rrw@cs.stanford.edu

Abstract

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.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes

Keywords and phrases satisfiability, derandomization, circuit complexity

Digital Object Identifier 10.4230/LIPIcs.CSL.2015.14

Category Invited Talk

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 this magical circuit).

The general problem of mathematically proving computational lower bounds is 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 [15], we do have good candidates for step zero, and possibly step one.) Many present lower bound conjectures may simply be false. In spite of our considerable intuitions 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.

* 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 simulators – 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 are equally valid when one adds "oracles" to the computational model: that is, one adds an instruction that can call an *arbitrary* function $O : \{0,1\}^\star \to \{0,1\}$ in one step, as a black box. When a proof of a theorem is true no matter which O is added to the instruction set, we say that the proof "relativizes." A relativizing proof is generally a quite powerful and broadly applicable object. However, relativizing proofs are of limited use in lower bounds: for instance, $\mathsf{P} = \mathsf{NP}$ when some oracles O are added to polynomial-time and nondeterministic-polynomial-time algorithms, but $\mathsf{P} \neq \mathsf{NP}$ when some other oracles O' are added (as proved by Baker, Gill, Solovay [3]). Practically all other open lower bound conjectures exhibit a similar resistance to arbitrary oracles, and a surprisingly large fraction of theorems in complexity theory do relativize.

The more recent "algebrization" barrier [1] 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 a more general algebraic object.) To scale relativization and algebrization, it is necessary to crack open the guts of programs, and reason more closely about their behavior relative to their simple instructions.[1]

The Razborov-Rudich "natural proofs" barrier [19] 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 our colleague Donald Knuth have dubbed problems like P versus NP to be "unknowable" [11].

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 point of this article, starting with the observation that while reasoning about lower bounds appears to be difficult, reasoning about *worst-case algorithms* does not suffer from the same appearance. Reasoning computationally about an infinite number of finite objects is commonplace in the analysis of worst-case algorithms. There, we have a function $f : \Sigma^\star \to \Sigma^\star$ in mind, and we prove that some known efficient procedure P outputs $f(x)$ on *all possible* finite inputs x. That is, often in

[1] 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 [2] for more discussion.

algorithm analysis we manage to reason about all possible x, even those x's we will never see or encounter in the real world. The idea is that, if we consider 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 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 [20]) tells us that, if x encodes a program that takes arbitrarily long inputs, it is undecidable to determine non-trivial properties of the problem solved by x. The source of this undecidability is the "arbitrary length" of inputs; if x encodes a program that takes only *finitely* many inputs, say only inputs of a fixed length, then we can produce the entire function computed by x, and decide some non-trivial properties of that function. To simplify the discussion (and without significantly losing generality), we might as well think of x as encoding a Boolean 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 reasoning about the finitary behavior of finite computational objects.

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 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 (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 hard 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" [19] 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 sort of bound and the $2^k \cdot n^{O(1)}$ time bound obtained by simple exhaustive search over all possible inputs to the circuit. What if we simply asked for a *non-trivial running time* for detecting the satisfiability of C, something 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 time algorithm would not be terribly more useful in *practice* than a 2^k one: only for small values of k would one see a difference, and the rest of the instances would remain intractable.[2] Work on worst-case

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

algorithms for SAT for many years (such as [13, 6, 14, 12, 18, 9, 22, 17, 23, 5], see the survey [7]) was primarily motivated by the intrinsic interest in understanding whether trivial exhaustive search is optimal.

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 "non-relativizing" analysis of the given circuit C, an analysis which relies on the structure and encoding of C. If you were asked to design a Circuit SAT algorithm which could only access C as an *oracle*, obtaining outputs from inputs and no other information, 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 ($k^{O(1)}$ size) circuit could hide a satisfying k-bit input from you. To thwart you, I may choose a small circuit which only outputs 1 on the "last" input you will call it on, and since you only see 0-outputs, you will need to call the circuit on all 2^k inputs to determine satisfiability. 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.[3]

A Possible Road to Circuit Complexity. 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 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 *data complexity* as a way of measuring the difficulty of testing, and describe how circuit complexity lower bounds are *equivalent* to data complexity upper bounds. We conclude the article with some hopeful thoughts.

[3] 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 [10, 26].

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 ([2], 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 $L : \{0,1\}^\star \to \{0,1\}$, we extend our computational model to have 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 animal. It could be enormously powerful, or it could be fairly weak. It is easy to see that every language over the single alphabet symbol 0 is in P/poly; however, a simple counting argument shows there are undecidable subsets over a single alphabet symbol. Therefore P/poly can "compute" some undecidable languages. In that sense, P/poly is powerful, but this really stems 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 not going to be very helpful in understanding the power of P/poly.) However, P/poly also looks obviously limited, in another sense: for each input length 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 $\mathsf{NP} \not\subset \mathsf{P/poly}$ (which would in turn imply $\mathsf{P} \neq \mathsf{NP}$). But it is an open problem to prove that $\mathsf{NEXP} \not\subset \mathsf{P/poly}$! That is, every language in the *exponential-time version of* NP could in fact have polynomial-size circuits. It looks amazing that a problem like this is still open. Fifty years ago, Hartmanis and Stearns [8] 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 from polynomial size? This open problem demonstrates how truly difficult it is to prove lower bounds on circuit complexity; maybe the infinite circuit model is powerful!

2.1 Enter Circuit SAT

Let's return to thinking about the role of Circuit Satisfiability. We were arguing that a faster algorithm for satisfiability points to a deficiency in the power of circuits, being unable to hide satisfying inputs from algorithms running in less than 2^k steps. We want 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 a function $f : \{0,1\}^\star \to \{0,1\}$,
that is not computable by *all* circuit families from that class $\mathcal{C}$

Written this way, the logical quantifiers match up nicely, and the whole 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** ([26, 28]). *Suppose for all polynomials p, 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 functions computable in nondeterministic exponential time that do not have polynomial-size circuits.*

(The polynomial n^{10} is almost certainly not optimal, but it suffices.) Notice the required improvement over exhaustive search: it would normally take $2^n \cdot p(n)$ time; in order to solve satisfiability fast enough, we need 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.

How Does The Proof Work? In this article, we can only briefly describe the proof of Theorem 2.1; for more technical details, see the surveys [24, 21, 16]. The informal statement in the box above says that a Circuit SAT algorithm A implies a hard function f. One might think that the algorithm A solving Circuit SAT may appear somewhere in the definition of f. That would be a very interesting property, but the proofs that we know do not do this explicitly. Instead, the proofs of Theorem 2.1 proceed by contradiction. We assume:

1. there is a Circuit SAT algorithm A running in $2^n/n^{10}$ time, and
2. every function $f \in$ NEXP has polynomial-size circuits. (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 (1) asserts that Circuit SAT can be solved faster, and item (2) asserts that a huge class of decidable problems can be computed with polynomial-size circuit families. Then we utilize these two assumptions to construct another algorithm which is provably *impossible*. Namely, these assumptions imply every function computable in nondeterministic time 2^n is computable in nondeterministic time $2^n/n^2$, which contradicts the time hierarchy theorem for nondeterminism [29]. At a very high level, the $2^n/n^2$ time algorithm works by:

- nondeterministically *guessing* small circuits for time 2^n computations, asserted to exist by item (2), and
- deterministically *verifying* the correctness of those circuits, using the Circuit SAT algorithm asserted by item (1).

The verification step is a subtle process. If a circuit happens to agree with our nondeterministic 2^n time function, it is not at all clear how we might use a circuit satisfiability call to check that circuit. Roughly speaking, we show that one can guess a small circuit C that is intended to succinctly encode an *accepting computation history* of the nondeterministic 2^n

time computation, and set up a larger circuit D (with C embedded in it) which is unsatisfiable if and only if C does encode an accepting history. This circuit D is carefully constructed so that its number of inputs is essentially n, logarithmic in the running time of the computation it is verifying. Then, a faster circuit satisfiability algorithm can check a 2^n time computation in less than 2^n steps.

While it yields the desired outcome, this style of proof is indirect 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 three unconditional circuit lower bound results:

- NEXP does not have so-called ACC^0 circuits of polynomial size [28],
- $\mathsf{NE}/1 \cap \mathsf{coNE}/1$ (a potentially weaker class) does not have ACC^0 circuits of polynomial size [25], and
- 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 [27].

The first and third results were obtained by designing explicit circuit satisfiability algorithms for relevant circuit classes; the second was obtained by sharpening the complexity-theoretic arguments. It is possible that we might prove $\mathsf{NEXP} \not\subset \mathsf{P}/\mathrm{poly}$ without providing a new Circuit SAT algorithm: it might be that the *assumption* $\mathsf{NEXP} \subset \mathsf{P}/\mathrm{poly}$ could imply the existence of algorithms sufficient for proving $\mathsf{NEXP} \not\subset \mathsf{P}/\mathrm{poly}$. (It is known that $\mathsf{NEXP} \subset \mathsf{P}/\mathrm{poly}$ 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 an emerging 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 it suffice to use a small number of input-output examples, and determine correctness of the program?

If we had no constraints on what the program could be, then there would not be 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 for sure. But in testing, we're never given a program as a black box; we know *something* about it, such as its size. Could side information such as program size be useful for the testing problem?

Recently with Brynmor Chapman [4], we have proposed the general problem of *data design*. Suppose we are given a function $f : \{0,1\}^\star \to \{0,1\}$, and a class $\mathcal{C}$ of size-s circuits. The task of data design 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, the *data complexity* of f (as a function of the circuit size s) is defined to be the minimum number of input-output examples such that, for any n-input circuit C of size s, one can determine with certainty whether C computes f (on all n-bit inputs), by calling C on the examples. Every function f that depends on all of its inputs has data complexity

$O(2^s)$: 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 small?

While the data design problem certainly has practical motivation, we are interested in the problem due to its intriguing inversion of the roles of program and input. The circuit computing f is the *input* to the data design problem; the *program* for testing the circuit is the collection of input-output examples. We have uncovered a surprising correspondence between upper bounds on data design and lower bounds on circuit complexity. Generally speaking, designing good 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:

▶ **Corollary 3.1** ([4]). *A function f is in $\mathsf{P}/poly$ if and only if for some $\varepsilon > 0$, the data 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 $\mathsf{NEXP} \not\subset \mathsf{P}/\mathrm{poly}$, it would be necessary and sufficient to design test suites of subexponential data 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 designed must compute f on all n-bit inputs, and the data designed must test the functionality of f for all s-size circuits. There are two parts to the equivalence:

- If f has an n-input circuit of size at most s, then standard arguments show that our test suite essentially needs all 2^n input-output pairs for f on n bits, to distinguish "good" circuits computing f from slightly different circuits which give the wrong answer on one input.
- If f does not have an n-input circuit of size s, then arguments from the theory of zero-sum games show that the test suite only needs $\mathrm{poly}(s)$ examples on n-inputs in order to test all circuits of size at most s/n.

Putting the two items together, one can show that upper bounds on data design for f are equivalent to lower bounds on the circuit complexity of f. Therefore, reliable exhaustive circuit testing and proving circuit lower bounds are deeply related tasks, in ways that are not fully understood yet. 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 works is equivalent to proving a limitation on $\mathcal{C}$. 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 new (or old!) lower bounds with it.

4 Conclusion

I believe that 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

1 Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM TOCT*, 1, 2009.

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

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

4 Brynmor Chapman and Ryan Williams. The circuit-input game, natural proofs, and testing circuits with data. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 263–270, 2015.

5 Ruiwen Chen, Valentine Kabanets, Antonina Kolokolova, Ronen Shaltiel, and David Zuckerman. Mining circuit lower bound proofs for meta-algorithms. *Computational Complexity*, 24(2):333–392, 2015. See CCC'14.

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

7 Evgeny Dantsin and Edward A. Hirsch. Worst-case upper bounds. In *Handbook of Satisfiability*, pages 403–424. 2009.

8 Juris Hartmanis and Richard Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc. (AMS)*, 117:285–306, 1965.

9 Edward A. Hirsch. New worst-case upper bounds for SAT. *J. Autom. Reasoning*, 24(4):397–420, 2000.

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

11 Donald E. Knuth. Personal communication, 2015.

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

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

14 Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.

15 Ketan Mulmuley. The GCT program toward the *P* vs. *NP* problem. *Commun. ACM*, 55(6):98–107, 2012.

16 Igor Oliveira. Algorithms versus circuit lower bounds. Technical Report TR13-117, ECCC, September 2013.

17 Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-SAT. *JACM*, 52(3):337–364, 2005. (See also FOCS'98).

18 Pavel Pudlák. Satisfiability – algorithms and logic. In *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, (MFCS'98)*, pages 129–141, 1998.

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

20 H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*

21 Rahul Santhanam. Ironic complicity: Satisfiability algorithms and circuit lower bounds. *Bulletin of the EATCS*, 106:31–52, 2012.

22 Uwe Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.

23 Magnus Wahlström. *Algorithms, Measures, and Upper Bounds for Satisfiability and Related Problems*. PhD thesis, Linköping University, 2007.

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

25 Ryan Williams. Natural proofs versus derandomization. In *STOC*, pages 21–30, 2013.

26 Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. *SIAM J. Comput.*, 42(3):1218–1244, 2013. See also STOC'10.

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

28 Ryan Williams. Nonuniform ACC circuit lower bounds. *JACM*, 61(1):2, 2014. See also CCC'11.

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