The Computational Complexity Column

BY

Michal Koucký

Computer Science Institute, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
koucky@iuuk.mff.cuni.cz
https://iuuk.mff.cuni.cz/~koucky/

The Power of Constructing Bad Inputs

Ryan Williams*
CSAIL & EECS
MIT, Cambridge, Massachusetts, USA
rrw@mit.edu

Abstract

A *lower bound*, showing that a function f cannot be computed by some class C of algorithms, necessarily shows that for every algorithm A in C, there must exist a "bad" input x such that $f(x) \neq A(x)$. We consider the computational complexity of generating such bad inputs for a given f and class C, and we study how the complexity of this task relates to existing (and major open problems in) lower bounds.

1 Introduction

Out of decades of thought on how to prove complexity lower bounds (and failing), one maxim repeatedly emerges: strong complexity lower bounds are hard for us to prove. There are many formal "barriers" known to proving complexity lower bounds, such as the relativization barrier [BGS75], Razborov-Rudich natural proofs [RR97], algebrization [AW09, IKK09] (see also [AB18, AB19]), and locality [Yao89, CHO+20]. For instance, relativization tells us that we cannot rely on any generic "black-box" arguments for proving strong complexity lower bounds, but many fundamental proof techniques in complexity theory are generic in exactly this sense. The algebrization barrier generalizes the relativization barrier, showing that the non-relativizing methods behind theorems like $\mathsf{IP} = \mathsf{PSPACE}$, querying polynomials that represent computations, are not sufficient in themselves to prove (for example) $\mathsf{P} \neq \mathsf{NP}$, $\mathsf{P} \neq \mathsf{PSPACE}$, $\mathsf{EXP} \neq \mathsf{ZPP}$, $\mathsf{NEXP} \neq \mathsf{BPP}$, $\mathsf{EXP}^{\mathsf{NP}} \neq \mathsf{BPP}$, et cetera. I've often summarized the state of affairs as: *not only can we not prove lower bounds, but we can prove that we cannot prove lower bounds.*

*Partially supported by NSF CCF-2127597. Part of this work was completed while the author was visiting the Simons Institute for the Theory of Computing, participating in the *Meta-Complexity* program.

So, we have apparently a lot of information about what the proofs of longstanding open complexity lower bounds **cannot** look like, in that we know a variety of limitations on how such proofs must proceed. The starting point of this article is to ask the question:

> Q1: *What* **could** *a proof of a strong lower bound look like?*

Intuitively, complexity barriers tell us what techniques we should try to avoid, if we wish to separate complexity classes. Can we identify *obligations* that strong lower bounds must obey, properties that such lower bound proofs (of even "easier" separations, like $\mathsf{EXP}^{\mathsf{NP}} \neq \mathsf{BPP}$, separating exponential time with an NP oracle from randomized polynomial time) must possess? Rather than studying what is **not sufficient** for lower bounds, could we get a handle what is **necessary**?

Please keep in mind that we are not starting from a blank slate: it is not that there are no lower bounds whatsoever. When one reads introductions like this, one might get the impression that there are essentially no complexity lower bounds in the literature. This is not really true. There are many areas within complexity theory, for which researchers have managed to establish hosts of interesting and strong limitations and no-go theorems. One of the most successful of these areas is *communication complexity* [KN97, RY20] which has aided complexity lower bounds in VLSI circuit design, streaming algorithms, and Turing machines, to take three examples. Nevertheless, it is felt that there is a gap (or maybe even a chasm) between what kinds of lower bounds can currently be proved, and what we'd like to prove. Perhaps a better question then, is:

> Q2: *What properties are missing from the lower bounds that we know how to prove, which we will have to include in a proof of (say)* $\mathsf{NEXP} \neq \mathsf{BPP}$*?*

What are the missing ingredients in our lower bound toolkit? In this article, I will highlight one type of answer, from a recent paper co-authored with Lijie Chen, Ce Jin, and Rahul Santhanam [CJSW21]. I will discuss an interesting way in which efficient algorithms will have to be central to resolving major complexity lower bound questions. Let me be clear that I do not claim to have fully answered the above question Q2, in any sense. If I have brought your attention to the question, and if I have gotten you to think about it on your own for ten minutes, I will consider my job successful. (Even if you think my answer to the question is terrible.)

2 Lower-Bounding as Finding Bad Inputs

To get a handle on question Q2, we start by looking very carefully at what it means to prove a lower bound against a class of algorithms.

Let $f : \{0,1\}^\star \to \{0,1\}$ be a decision problem, and let $\mathcal{A}$ be a class of algorithms. In particular, we stipulate that each $A \in \mathcal{A}$ has a finite-length description, and each A takes in arbitrary-length Boolean inputs, outputting a single bit. An algorithm A is said to compute f if for all but finitely many inputs x, $A(x) = f(x)$. (We permit "all but finitely many", because if there were only finitely many inputs on which A and f differ, then this finite set could be "hard-coded" directly, into another larger but still finite algorithm that computes f everywhere.)

Therefore, a lower bound that "f is not in $\mathcal{A}$" is a claim of the form:

$$(\forall A \in \mathcal{A})(\exists^\infty n)(\exists x_n \in \{0,1\}^n)[A(x_n) \neq f(x_n)].$$

That is, for every candidate algorithm A for computing f, someone has to "respond" with a bad input x_n on which A does not compute f correctly. In fact, in the given setup, we should construct infinitely many bad x_n inputs (if there were only finitely-many bad inputs, they could be hard-coded into the algorithm).

Now fix a function f, and fix an algorithm A from a class $\mathcal{A}$. We ask:

Q3: *What is the complexity of constructing a bad input x_n of length n?*

Suppose you are provided 1^n, the string of n ones. We are asking: how difficult is it for you to construct an input $x_n \in \{0,1\}^n$ such that $f(x_n) \neq A(x_n)$? (You can output whatever n-bit string you want, if n is not the length of some bad input; in principle, there may be only infinitely many bad n.)

2.1 Two Types of Lower Bounds

Roughly speaking, the literature on complexity lower bounds gives two types of answers to question Q3. That is, the known proofs of lower bounds yield two different types of answers to Q3:

1. **"Random" or non-constructive ways of choosing bad inputs.** In general, such lower bound proofs rely on counting/information-theoretic arguments. For example, many proofs of time-space lower bounds for explicit problems in P (*e.g.,* [Hen65, Maa84, BC82, Bea91, BJS01, Ajt02, BSSV03, MW19]) work by choosing a random x_n from some distribution of n-bit inputs, and arguing that the randomness in the string can confound the algorithm A into making a mistake.

   For a simple example of such a lower bound, one can prove that no deterministic finite automaton (DFA) computes the language of palindromes $\{xx^R \mid x \in \{0,1\}^\star\}$, by arguing that we could "compress" an *arbitrary* string x by storing the state q_x of the DFA in which the last bit of x is read, and the length $|x|$, which takes only $O(\log |x|)$ bits. Then, if the DFA recognizes

palindromes, there should be a unique path of length $|x|$ from q_x to a final state, and the string x^R must be encoded along the path. Thus we could encode every x with a description of length only $O(\log |x|)$. However, a random string x requires a description of length at least $|x| - 1$ with nonzero probability, so we have a contradiction.[1] This argument generalizes to show that any DFA that recognizes all palindromes of length n requires at least $2^{\Omega(n)}$ states. At any rate, the argument shows that a *random* x is a bad input with decent probability.

2. **"Efficient" ways of choosing bad inputs.** Many proofs of lower bounds based on diagonalization arguments provide an efficient method for generating the bad input.

   For example, the standard proof by diagonalization that the Halting Problem is undecidable can be modified to have this property. In particular, given the code of a Turing machine that claims to solve the Halting Problem, one can efficiently produce a "bad" input on which the Turing machine fails to correctly decide the Halting Problem.[2] Similarly, the old-school proof of the Time Hierarchy Theorem, where one constructs a hard function by simulating different time-bounded Turing machines on different inputs (and flipping the answer), also has this property.

The idea of focusing on constructing bad inputs, when reasoning about impossibility results in computing, is in fact quite old. For instance, the celebrated Myhill-Nerode theorem [Ner58] says that a set of strings S is not regular if and only if there exists an infinite *distinguishing set* for S: an infinite set of "bad" strings that effectively forces every prospective DFA to need infinitely many states.

Mulmuley [Mul10] has suggested that to make progress on separating P and NP, one must search for **algorithms** which can efficiently find counterexamples for any algorithms claiming to solve the conjectured hard language. This view has been dominant in the GCT approach towards the VNP vs. VP problem [Mul07, Mul12, IK20]. One can think of our present article as confirming Mulmuley's intuition in a broader sense than what was known before.

[1] This is the first theorem one learns on Kolmogorov complexity. See Chapter 6.4 of Sipser [Sip06].

[2] For completeness, we sketch the proof. Given the code of some Turing machine H, let D_H be code for the "diagonal" machine which on an input x, flips the answer of $H(\langle x, x \rangle)$. Then H cannot terminate with a correct output on the input $\langle D_H, D_H \rangle$.

3 Starting Point: The Work of Gutfreund, Shaltiel, and Ta Shma

The starting point of our work was thinking about a beautiful paper of Gutfrend, Shaltiel, and Ta Shma [GST07]. They showed:

> *If* $\mathsf{P} \neq \mathsf{NP}$*, then bad inputs for every claimed poly-time algorithm for SAT can be constructed in poly-time.*

In particular, the following theorem can be derived from their arguments.

Theorem 3.1 (Follows from [GST07])**.** *Assume* $\mathsf{P} \neq \mathsf{NP}$*. For every* n^k*-time decision algorithm A, there is an algorithm* R_A *such that, for infinitely many n,* $R_A(1^n)$ *outputs a formula* F'_n *of length n such that* F'_n *is satisfiable if and only if* $A(F'_n)$ *outputs "UNSAT". Furthermore,* $R_A(1^n)$ *runs in* $n^{O(k^2)}$ *time.*

That is, R_A runs in polynomial time, and prints formulas on which A fails to determine SAT correctly. We say that the algorithm R_A is a *refuter*, since its job is to refute the claim that A solves the SAT problem by producing bad inputs for A.

As their theorem is very important to our work, we will carefully outline a proof of the theorem.

3.1 Refuter for Algorithms Trying to Print SAT Assignments

Let's start by assuming $\mathsf{P} \neq \mathsf{NP}$, and derive a refuter for any n^k-time algorithm A that attempts to print a SAT assignment to its input formula, whenever a SAT assignment exists. That is, our algorithm A first attempts to print a SAT assignment to its input formula, and A determines "UNSAT" if the printed assignment fails to satisfy, otherwise A determines "SAT".

Under this setup, since it's forced to print a SAT assignment, the algorithm A will always correctly output "UNSAT" when it's given an unsatisfiable formula. Therefore, under this setup, every bad input for A must be a satisfiable formula on which A determines "UNSAT". (Furthermore, since we assume $\mathsf{P} \neq \mathsf{NP}$, there must be infinitely many such formulas.)

This refuter is a bit easier to describe, yet it already captures the key idea: *if an algorithm A claims to solve SAT, then exploit its claimed ability to find its own counter-examples.*

We define the refuter as follows.

$R_A(1^n)$: Use the Cook-Levin reduction to construct a 3CNF formula F_n that is satisfiable if and only if the following holds:

$$(\exists G : |G| = n)(\exists a)[G(a) = 1 \wedge A(G) \text{ outputs "UNSAT"}].$$

Run A on F_n. If A prints a satisfying assignment (G', a') to F_n, then output G'. If A outputs "UNSAT" instead, then output F_n.

In more detail, since A is a poly-time algorithm, the property

$$(\exists G : |G| = n)(\exists a)[G(a) = 1 \wedge A(G) \text{ outputs "UNSAT"}]$$

can be checked in NP. Therefore, applying the Cook-Levin reduction, there is a 3CNF formula F_n such that $F_n(G, a)$ is true if and only if $G(a) = 1 \wedge A(G)$ outputs "UNSAT". Since A runs in time n^k, the formula F_n output by the Cook-Levin reduction has size at most $n^{O(k)}$.

What does R_A do? The refuter R_A is asking A to print its own counterexamples! That is, if A outputs a valid SAT assignment (G', a') on F_n, then G' is a formula of length n that is a "bad input" for A, by definition of F_n. Now we have two cases.

Case 1: If A outputs a valid SAT assignment G' for F_n for infinitely many n, the proof is complete: for infinitely many n, $R_A(1^n)$ outputs a formula F_n of length n such that F_n is satisfiable if and only if $A(F_n)$ outputs "UNSAT".

Case 2: The alternative is that for all but finitely many n, A outputs "UNSAT" on F_n. In this case, $R_A(1^n)$ outputs F_n on all but finitely many n. But since $\mathsf{P} \neq \mathsf{NP}$, the formula F_n is actually satisfiable for infinitely many n. Therefore, for infinitely many n, $R_A(1^n)$ still outputs a formula that is satisfiable, yet A reports "UNSAT".[3]

3.2 Refuter for Algorithms Trying to Decide SAT

Gutfreund, Shaltiel, and Ta Shma actually give a refuter for every poly-time algorithm that attempts to *decide* SAT as well. This refuter works by exploiting the well-known search-to-decision reduction for SAT. In particular, given an algorithm A that only decides SAT, we can produce an algorithm B that can print SAT assignments (when they exist) making only polynomially many calls to A, by plugging in values of variables into the formula and calling A to check if the reduced formula is still satisfiable. Now, the refuter has the following form, for $i = 1, 2, 3$:

[3] A minor detail: in this case, R_A is (infinitely often) outputting a formula of length $L = n^{O(k)}$ on the string 1^n, so it does not meet our original specification, where we want to output a string of length n on 1^n. However, a modified algorithm R'_A which on the input string 1^L, determines n and runs $R_A(1^n)$, does meet the specification.

$R_A^i(1^n)$: Use the Cook-Levin reduction to construct a 3CNF formula F_n that is satisfiable if and only if the following holds:

$$(\exists G : |G| = n)(\exists a)[G(a) = 1 \wedge A(G) \text{ outputs "UNSAT"}].$$

Use search-to-decision on A to search for a SAT assignment to F_n. (Call A on F_n; if it reports "UNSAT" then abort. Try setting a variable x to 0, in F_n. If A reports "SAT" then continue with another variable. If A reports "UNSAT" then flip the value of x to 1. If A still reports "UNSAT" then abort. If A reports "SAT" then continue with another variable.)
There are three possible outcomes:

1. $A(F_n)$ = "UNSAT". In this case, output F_n.
2. A finds a SAT assignment (G', a'). In this case, output G'.
3. In the search-to-decision reduction, A reaches a subformula F_n'' such that $A(F_n'')$ = "SAT", but when a variable x of F_n'' is set 0 (yielding $F_n''[x = 0]$), or set 1 (yielding $F_n''[x = 1]$), A reports "UNSAT" in both cases and we abort.
   Then, A must be wrong on at least one of the three. In this case, $R_A^1(1^n)$ reports F_n'', $R_A^2(1^n)$ reports $F_n''[x = 0]$, and $R_A^3(1^n)$ reports $F_n''[x = 1]$.[4]

Similarly to the previous refuter against algorithms printing SAT assignments, there are a few cases to check.

Case 1: If A outputs a valid SAT assignment on F_n for infinitely many n, we are done.

Case 2: If A reports "UNSAT" on F_n for all but finitely many n, we are done (same analysis as the refuter from the previous subsection).

Case 3: In the remaining case, A only outputs a SAT assignment on F_n for finitely many n, yet A reports "SAT" on F_n for infinitely many n. Thus there are infinitely many n such that A reports the wrong answer on at least one of

$$F_n'', \quad F_n''[x = 0], \quad F_n''[x = 1].$$

Since R_A^1 always reports F_n'', R_A^2 always reports $F_n''[x = 0]$, and R_A^3 always reports $F_n''[x = 1]$, there must be an $i \in \{1, 2, 3\}$ such that for infinitely many n, $R^i(1^n)$ outputs a formula on which A is incorrect!

This concludes the proof of Theorem 3.1.

A Universal Refuter. In fact, for separations like that of SAT against n^k-time algorithms that print SAT assignments, one can construct a *single* $n^{O(k^2)}$-time refuter R which works against all n^k-time algorithms A, infinitely often. The idea is to simply perform "Levin search" [Lev73]: we consider a "meta-algorithm" A' which on a formula F of length n, runs each of the first $\log(n)$ algorithms with n^k-time alarm clocks, and if *any* of the first $\log(n)$ algorithms outputs a SAT assignment to F, then A' outputs it. Now, the refuter $R_{A'}$ for A' is a refuter for all n^k-time algorithms.

3.3 An Aside: Finding Hard Instances for Practical SAT Solvers

Although the above theorem is rather complexity-theoretic in nature, we believe that the ideas could be useful in finding "hard instances" for practical SAT solvers. Let t and m, n be positive integer parameters. For any given solver S, in principle one can build a SAT instance $F_{S,n,t}$ which is satisfiable if and only if there exists a 3CNF G with m clauses and n variables such that G is satisfiable, yet S does not conclude that G is satisfiable within t decisions/backtracks/seconds (whatever notion of time is easiest to encode). If S can solve $F_{S,n,t}$ (indeed, if *any* solver can solve $F_{S,n,t}$) then the solution will produce a hard instance. If no solver can solve $F_{S,n,t}$, then the formula $F_{S,n,t}$ is itself a good candidate for a hard instance. One can imagine holding a "tournament" between a host of practical SAT solvers, feeding various formulas $F_{S',n,t}$ into various solvers S'', to produce many interesting hard instances.

4 Constructive Separations

Theorem 3.1 of the previous section showed that it's possible to efficiently produce "hard inputs" for claimed SAT solvers, assuming $\mathsf{P} \neq \mathsf{NP}$. To generalize this notion, we propose the following definition, letting f be a decision problem and $\mathcal{A}$ be a class of algorithms.

Definition 4.1. We say there is a **P-constructive separation of $f \notin \mathcal{A}$** if for all algorithms $A \in \mathcal{A}$, there is a polynomial-time algorithm R_A such that, for infinitely many n, $A(R_A(1^n)) \neq f(R_A(1^n))$.

Thus, a P-constructive separation means that for every "weak" algorithm, we can concoct a polynomial-time algorithm that produces bad inputs for the weak algorithm. Theorem 3.1 can then be expressed as:

If $\mathsf{P} \neq \mathsf{NP}$, then there's a P-constructive separation of SAT $\notin \mathsf{P}$.[5]

[5]Here, we are conflating the class of polynomial-time algorithms with the class of decision problems solvable in polynomial time. My apologies if this bothers you.

So, we know that proving $\mathsf{P} \neq \mathsf{NP}$ will also require us to be able to efficiently construct hard SAT instances for polynomial-time algorithms. A natural question arises:

> *Which complexity lower bound problems* **require** *constructive separations, and which do not?*

Our question is an algorithmic one about the nature of a lower bound. What algorithms are implied by complexity lower bounds? In our paper [CJSW21], we try to make a case that constructive separations are a key to proving major separations between complexity classes. We prove that:

1. Essentially all major separation problems regarding polynomial time will require constructive separations.
2. Making many known lower bounds constructive requires resolving *other* major lower bound problems.

That is, we believe that the property of constructivity (the ability to efficiently refute weak algorithms) lies in the "gap" between lower bounds we know how to prove, and major lower bounds that we'd like to prove. Constructivity is a property we *want* of lower bounds; it is in a sense the opposite of a barrier.

In the next two subsections, we explain our results in more detail.

4.1 Major Complexity Class Separations Will Require Constructive Separations

One of our main theorems is that for *many* choices of complexity classes C and $\mathcal{D}$, a separation $C \neq \mathcal{D}$ implies a *constructive* separation of $f \notin C$ for some function $f \in \mathcal{D}$.

Theorem 4.2 (Informal, see [CJSW21] for details)**.** *For all classes $C \in \{\mathsf{P}, \mathsf{ZPP}, \mathsf{BPP}\}$ and all classes $\mathcal{D} \in \{\mathsf{NP}, \Sigma_2\mathsf{P}, \mathsf{PP}, \mathsf{PSPACE}, \mathsf{EXP}, \mathsf{NEXP}, \mathsf{EXP}^{\mathsf{NP}}\}$, if $C \neq \mathcal{D}$, then there is a C-constructive separation of $f \notin \mathcal{D}$, for a "natural" function $f \in \mathcal{D}$.*

The above theorem is informal, in that (a) we have not defined "natural" (but the properties needed hold of most $\mathcal{D}$-complete problems), and (b) we have not defined what it means to be ZPP-constructive or BPP-constructive (but it is a natural randomized notion of constructive separation; see the paper for details).

The above Theorem 4.2 generalizes Theorem 3.1 to hold for many different classes C and $\mathcal{D}$. A couple of these other cases were known prior to our paper [GST07, DFG13], but most were not. See our paper [CJSW21] for more details.

Theorem 4.2 says that, if we manage to prove a good separation against (randomized) polynomial time, then we are also going to get a constructive separation (in which we can efficiently produce bad inputs). So, we might as well think about constructive methods for proving lower bounds!

In Section 5, we will give one particular example of such a result, proving that if $\mathsf{P} \neq \mathsf{PSPACE}$, then there is a P-constructive separation that true quantified Boolean formulas (TQBF) is not in P.

4.2 Making Known Lower Bounds Constructive Implies Strong Circuit Lower Bounds

In the second part of the paper [CJSW21], we show how constructive separations for several different well-known lower bounds (based on information-theoretic arguments) would turn out to imply breakthrough lower bounds. That is, "constructivizing" any one of many known lower bounds would have actually have rather significant lower bound consequences. The three regimes we consider are:

- (randomized) streaming lower bounds,
- query complexity lower bounds, and
- superlinear-time one-tape Turing machine lower bounds.

Streaming lower bounds and query complexity lower bounds are generally considered to be well-understood, and certain superlinear-time lower bounds against one-tape Turing machines have been known for decades [Hen65, Maa84]. Surprisingly, we show in [CJSW21] that making these separations constructive would imply breakthrough separations such as $\mathsf{EXP}^{\mathsf{NP}} \neq \mathsf{BPP}$, or even $\mathsf{P} \neq \mathsf{NP}$ (if the algorithm producing bad inputs is restricted enough). Here, we briefly outline our results in more detail.

Constructivizing Streaming Lower Bounds Implies Breakthroughs. In the streaming algorithm setting, an algorithm (storing little space) must pass through all bits of the input stream exactly once, and output a good answer when the stream ends. A host of problems are well-known to be *unconditionally* hard for randomized streaming algorithms that use a small amount of working space, and these lower bounds typically follow from communication complexity lower bounds. For a canonical example, in the Set-Disjointness (DISJ) communication problem, Alice is given an n-bit string x, Bob is given an n-bit string y, and the goal is to determine whether or not the inner product $\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i$ is nonzero, with minimal communication. We can think of DISJ as a function that takes (x, y) and outputs a Boolean value, in the natural way. The randomized communication complexity lower bounds for DISJ [KS92, Raz92, BJKS04] directly imply that

no $n^{1-\varepsilon}$-space randomized streaming algorithm can correctly decide the simple language

$$L_{\mathsf{DISJ}} = \{xy \in \{0,1\}^\star \times \{0,1\}^\star \mid |x| = |y|, \mathsf{DISJ}(x,y) = 1\}.$$

Therefore, every (randomized) streaming algorithm using only $n^{o(1)}$ workspace must fail to correctly decide L_{DISJ} on some inputs. We show that efficient refuters against streaming algorithms attempting to solve *any* NP problem (not just L_{DISJ}) would imply a breakthrough separation against *general* randomized algorithms.

Theorem 4.3 (Informal)**.** *For every language $L \in \mathsf{NP}$, a P^{NP}-constructive separation of L from uniform randomized streaming algorithms using $O(\log n)^{\omega(1)}$ space implies $\mathsf{EXP}^{\mathsf{NP}} \neq \mathsf{BPP}$.*

Essentially every lower bound proved against streaming algorithms in the literature holds for some problem whose decision version is in NP. Theorem 4.3 shows if *any* such lower bound can be made constructive, even if it takes P^{NP} to produce the bad inputs, then $\mathsf{EXP}^{\mathsf{NP}} \neq \mathsf{BPP}$ follows, a longstanding (embarassing) open problem in complexity theory. And if we could replace "P^{NP}-constructive" with "P-constructive", we would prove that $\mathsf{EXP} \neq \mathsf{BPP}$. The upshot is that the counterexample inputs printed by any such refuter algorithm must encode a function that is actually hard for *general* randomized algorithms.

Constructivizing Lower Bounds for One-Tape Turing Machines Implies Breakthroughs. It has been known at least since Maass [Maa84] that nondeterministic one-tape Turing machines require $\Omega(n^2)$ time to decide even simple problems such as PALINDROMES. However, the lower bounds (and others like it) are proved by non-constructive counting arguments. (One can also use $\Omega(n)$ lower bounds on the nondeterministic communication complexity of the EQUALITY function, to prove such lower bounds.) Similarly to the previous setting, we show that if there is a P^{NP} refuter that can produce bad inputs for a given one-tape (nondeterministic) Turing machine, then E^{NP} ($2^{O(n)}$ time with an NP oracle) requires exponential-size circuit complexity. This in turn would imply $\mathsf{BPP} \subseteq \mathsf{P}^{\mathsf{NP}}$, a breakthrough simulation of randomized polynomial time. The theorem we prove is very general, and applies to many more problems than just PALINDROMES:

Theorem 4.4. *For every language L computable by a nondeterministic $n^{1+o(1)}$-time RAM, a P^{NP}-constructive separation of L from nondeterministic $O(n^{1.1})$-time one-tape Turing machines implies $\mathsf{E}^{\mathsf{NP}} \not\subset \mathsf{SIZE}[2^{\delta n}]$ for some constant $\delta > 0$.*

Let us demonstrate how Theorem 4.4 works, with the specific example of PALINDROMES. Our approach can be readily generalized to a proof of Theorem 4.4.

Proof of Theorem 4.4. (Sketch) Let $\varepsilon > 0$ be arbitrarily small. Our goal is to construct a nondeterministic one-tape Turing machine M that takes $N^{1+c\varepsilon}$ time for a universal constant $c \geq 1$, such that M has the following properties:

- For every n, M **accepts** every palindrome $xx^R \in \{0,1\}^{2N}$ of length $2N = 2^{n+1}$ such that x, construed as the length-2^n truth table of a function from n bits to 1 bit, has circuit complexity at most $2^{\varepsilon n}$.
- M **rejects** every string $y \in \{0,1\}^{2N}$ that is not a palindrome and has circuit complexity at most $2^{\varepsilon n}$, when construed as a function from $n+1$ bits to 1 bit.
- M **rejects** every string of odd length (for simplicity, we only consider even-length palindromes, of the form xx^R).

That is, M correctly decides PALINDROMES on all strings of circuit complexity at most $2^{\varepsilon n}$. This Turing machine M exists unconditionally: its correctness does not rely on any assumptions, and we will describe how to construct M later.

Given that such an M exists, for sufficiently small $\varepsilon > 0$, M runs in time $o(N^2)$. Therefore it must fail to correctly solve PALINDROMES on infinitely many inputs. Consider any P^{NP} algorithm R that, on the input 1^N, prints an input $z_N \in \{0,1\}^N$ on which M fails to decide PALINDROMES, for infinitely many N.

The properties of M imply that the string z_N does *not* have circuit complexity at most $2^{\varepsilon n}$. Therefore, there is a P^{NP} procedure R that, on infinitely many 1^N, prints the truth table of a function with circuit complexity greater than $2^{\varepsilon n}$. We can produce a function $f \in \mathsf{E}^{\mathsf{NP}}$ with circuit complexity greater than $2^{\varepsilon n}$, as follows.

> $f(x) :=$ Let $n = |x|$. Run $R(1^{2^n})$, producing a string z_{2^n} of length 2^n.
> Let $y_1, \ldots, y_{2^n}$ be the list of all n-bit strings in lex order.
> Output the i-th bit of z_{2^n}, where $x = y_i$.

Observe that the truth table of f, on inputs of length n, is precisely the string z_{2^n}. Therefore, f requires circuit complexity greater than $2^{\varepsilon n}$, for infinitely many n.

Now we turn to the construction of the desired nondeterministic Turing machine M; here, we just sketch how it works. First, M rejects its input z immediately if $|z|$ is odd; this can be easily checked in linear time in a standard way. Note that M can also compute $|z|$ directly in $N \cdot \mathrm{poly}(\log N)$ time, by "dragging along" an $O(\log n)$-bit counter on its single tape as it streams through the bits of z, using a larger alphabet to store the content of the counter. Next, given that $|z| = 2N = 2^{n+1}$, M nondeterministically guesses a circuit C of size at most N^{ε}, using $O(N^{\varepsilon} \log N)$

bits of nondeterminism. The intention is that $z = xx^R$ and C is a circuit such that $C(i)$ outputs the i-th bit of x. The machine M can check this as follows. First, by "dragging along" the description of C as it reads the bits of x, M can verify that $C(i)$ outputs the i-th bit of x. Evaluating C on an input takes no more than $O(N^{c\varepsilon})$ time per bit of x for a fixed constant $c \geq 1$, which is in total $O(N^{1+c\varepsilon})$ time over all bits of x. Finally, M can verify that z is a palindrome by using C to check that the first bit of z matches the last bit, the second bit of z matches the next-to-last bit, and so on, using evaluations of C to check the first half of z. All this takes no more than $O(N^{1+c\varepsilon})$ time, and M accepts if and only if all bit checks pass.

It is easy to see that if z has circuits of size at most N^{ε}, and z is a palindrome, then the computation path of M that guesses a small circuit correctly will indeed accept z. However, if z is not a palindrome, then no computation path of M will accept z. □

Constructivizing Certain Query Lower Bounds Implies Breakthroughs. Even obtaining efficient refuters for query lower bounds on the "coin problem" [BV10] would imply strong lower bounds. We define the problem $\mathsf{Promise\text{-}MAJORITY}_{n,\varepsilon}$ for a parameter $\varepsilon < 1/2$, as follows:

> $\mathsf{Promise\text{-}MAJORITY}_{n,\varepsilon}$: *Given an input* $x \in \{0,1\}^n$*, letting* $p = \frac{1}{n}\sum_{i=1}^{n} x_i$*, distinguish between the cases* $p < 1/2 - \varepsilon$ *or* $p > 1/2 + \varepsilon$*.*

It is well-known that every randomized query algorithm needs $\Theta(1/\varepsilon^2)$ queries to solve $\mathsf{Promise\text{-}MAJORITY}_{n,\varepsilon}$ with constant success probability (uniform random sampling is the best one can do). That is, any randomized query algorithm making $o(1/\varepsilon^2)$ must make mistakes on some inputs, with high probability. We can show that constructing efficient-enough refuters for this simple sampling lower bound would imply $\mathsf{P} \neq \mathsf{NP}$. Please see the paper [CJSW21] for details.

Theorem 4.5 (Informal)**.** *A "uniform* AC^0*" constructive separation of the problem* $\mathsf{Promise\text{-}MAJORITY}_{n,\varepsilon}$ *from all randomized query algorithms that make only* $o(1/\varepsilon^2)$ *queries and run in* $\mathrm{poly}(1/\varepsilon)$ *time, implies* $\mathsf{P} \neq \mathsf{NP}$*.*

Finally, we also show that constructive separations for the Minimum Circuit Size Problem (MCSP) against AC^0 circuits would also imply unexpected breakthrough lower bounds. (Informally, the Minimum Circuit Size Problem (MCSP) is the problem of determining the circuit complexity of a given 2^n-bit truth table.) As above, it is known that MCSP does not have small AC^0 circuits [ABK+02], and the question is whether there is a *constructive* separation. We refer the reader to the paper [CJSW21] for details.

5 Proving Polynomial-Time Lower Bounds for TQBF Requires Constructivity

To give one example of how Theorem 3.1 can be extended beyond $\mathsf{P} \neq \mathsf{NP}$ to other major open problems about polynomial time, I will demonstrate here how $\mathsf{P} \neq \mathsf{PSPACE}$ implies a P-constructive separation of TQBF $\notin \mathsf{P}$. That is, assuming $\mathsf{P} \neq \mathsf{PSPACE}$, given any poly-time algorithm A that attempts to decide the PSPACE-complete problem TQBF (true quantified Boolean formulas), we can *efficiently* find QBFs on which A fails. Notice, since we are now talking about a lower bound on a PSPACE-complete problem, there is no way we can definitively *check* all answers to the algorithm A in polynomial time: unless $\mathsf{NP} = \mathsf{PSPACE}$, we cannot even give short proofs that a QBF is true or false. Nevertheless, we can still locate bad inputs for A. I have chosen the particular example of refuters for $\mathsf{P} \neq \mathsf{PSPACE}$ with TQBF, because one can apply similar ideas as in the refuter for $\mathsf{P} \neq \mathsf{NP}$ with SAT. In the paper [CJSW21], we use more sophisticated ideas to obtain refuters for many more complexity class separation problems.

In particular, assume $\mathsf{P} \neq \mathsf{PSPACE}$, and let A be an n^k-time that attempts to decide TQBF by outputting "true" or "false" on every encoding of a Boolean formula. For a quantified Boolean formula (QBF) F, let "F'' denote its encoding in binary. Say that A is *inconsistent* on F if:

- Either $F = (\forall x)G(x)$ for a QBF G with one free variable, and $A(\text{“}F'') \neq A(\text{“}G(0)'') \wedge A(\text{“}G(1)'')$,
- or $F = (\exists x)G(x)$ for a QBF G with one free variable, and $A(\text{“}F'') \neq A(\text{“}G(0)'') \vee A(\text{“}G(1)'')$.

Of course, $(\forall x)G(x)$ is true if and only if $G(0)$ and $G(1)$ are true, and $(\exists x)G(x)$ is true if and only if $G(0)$ or $G(1)$ is true. Thus, A is inconsistent when it fails to satisfy this basic property of quantifier semantics, and A must be incorrect on at least one of three QBFs.

We can define a refuter against A, as follows. Let $i \in \{1, 2, 3\}$.

$R_A^i(1^n)$: Construct a formula F_n encoding the property

$$(\exists \text{ QBF } \text{“}G'', |G| = n)[A \text{ is inconsistent on } G].$$

If $A(F_n)$ outputs "false", then output F_n.
Otherwise, $A(F_n)$ outputs "true". As in Theorem 3.1, use search-to-decision to try to construct a QBF G on which A is inconsistent.
If the search-to-decision fails (we reach a formula that A declared "true" but its

two subformulas are declared "false"), then as in Theorem 3.1, we have a set of three QBFs such that A is incorrect on at least one of them; R^i will output the ith such formula.
If the search-to-decision succeeds, then A is inconsistent on G, and we obtain three QBFs (G and two other "subformulas") such that A is incorrect on at least one of them; R^i will output the ith such formula.

Analogously as in Theorem 3.1, we can argue that there is always some $i \in \{1, 2, 3\}$ such that for infinitely many n, $R^i_A(1^n)$ outputs a QBF on which A is incorrect.

6 Conclusion

We hope this article has encouraged the reader to think more about how algorithmic methods are actually necessary for proving strong complexity lower bounds.

Our work leaves open several interesting directions. For example, it is not entirely clear how to extend our results to separations with complexity classes within P. For example, let L be a decision problem which is complete for P under logspace reductions. If L is not decidable in $\mathsf{LOGSPACE}$, does a "logspace-constructive" separation of $L \notin \mathsf{LOGSPACE}$ follow? What about constructive separations for non-uniform complexity classes, such as $\mathsf{P/poly}$? Should we expect a constructive separation of $\mathrm{SAT} \notin \mathsf{P/poly}$, and if so, what properties should the algorithms/circuits have?

Finally, in this invited article, I have not provided an overview of all relevant prior work. Such an overview can be found in our paper [CJSW21].

Acknowledgments

I am grateful to my co-authors, Lijie Chen, Ce Jin, and Rahul Santhanam, for collaborating with me on this project. I also thank Michal for inviting me to write about this work, and his patience with me while I was finishing this article.

References

[AB18] Baris Aydinlioglu and Eric Bach. Affine relativization: Unifying the algebrization and relativization barriers. *ACM Trans. Comput. Theory*, 10(1):1:1–1:67, 2018.

[AB19] Baris Aydinlioglu and Eric Bach. Corrigendum to affine relativization: Unifying the algebrization and relativization barriers. *ACM Trans. Comput. Theory*, 11(3):16:1, 2019.

[ABK+02] Eric Allender, Harry Buhrman, Michal Koucký, Dieter van Melkebeek, and Detlef Ronneburger. Power from random strings. *SIAM J. Comput.*, 35(6):1467–1493, 2006. Preliminary version in FOCS'02.

[Ajt02] Miklós Ajtai. Determinism versus nondeterminism for linear time rams with memory restrictions. *Journal of Computer and System Sciences*, 65(1):2–37, 2002.

[AW09] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM Trans. Comput. Theory*, 1(1):2:1–2:54, 2009.

[BC82] Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982.

[Bea91] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991.

[BGS75] Theodore P. Baker, John Gill, and Robert Solovay. Relativizations of the P =?NP question. *SIAM J. Comput.*, 4(4):431–442, 1975.

[BJKS04] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.*, 68(4):702–732, 2004.

[BJS01] Paul Beame, Thathachar S Jayram, and Michael Saks. Time-space tradeoffs for branching programs. *Journal of Computer and System Sciences*, 63(4):542–572, 2001.

[BSSV03] Paul Beame, Michael Saks, Xiaodong Sun, and Erik Vee. Time-space trade-off lower bounds for randomized computation of decision problems. *JACM*, 50(2):154–195, 2003.

[BV10] Joshua Brody and Elad Verbin. The coin problem and pseudorandomness for branching programs. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 30–39, 2010.

[CHO+20] Lijie Chen, Shuichi Hirahara, Igor Carboni Oliveira, Ján Pich, Ninad Rajgopal, and Rahul Santhanam. Beyond natural proofs: Hardness magnification and locality. In *11th Innovations in Theoretical Computer Science Conference, ITCS*, pages 70:1–70:48, 2020.

[CJSW21] Lijie Chen, Ce Jin, Rahul Santhanam, and R. Ryan Williams. Constructive separations and their consequences. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 646–657. IEEE, 2021.

[DFG13] Shlomi Dolev, Nova Fandina, and Dan Gutfreund. Succinct permanent is *NEXP*-hard with many hard instances. In *Algorithms and Complexity, 8th International Conference, CIAC 2013. Proceedings*, volume 7878 of *Lecture Notes in Computer Science*, pages 183–196. Springer, 2013.

[GST07] Dan Gutfreund, Ronen Shaltiel, and Amnon Ta-Shma. If NP languages are hard on the worst-case, then it is easy to find their hard instances. *Computational Complexity*, 16(4):412–441, 2007.

[Hen65] F. C. Hennie. Crossing sequences and off-line turing machine computations. In *6th Annual Symposium on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, USA, October 6-8, 1965*, pages 168–172. IEEE Computer Society, 1965.

[IK20] Christian Ikenmeyer and Umangathan Kandasamy. Implementing geometric complexity theory: on the separation of orbit closures via symmetries. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 713–726. ACM, 2020.

[IKK09] Russell Impagliazzo, Valentine Kabanets, and Antonina Kolokolova. An axiomatic approach to algebrization. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC*, pages 695–704. ACM, 2009.

[KN97] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.

[KS92] Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discrete Math.*, 5(4):545–557, 1992.

[Lev73] Leonid Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[Maa84] Wolfgang Maass. Quadratic lower bounds for deterministic and nondeterministic one-tape turing machines (extended abstract). In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 401–408. ACM, 1984.

[Mul07] Ketan Mulmuley. Geometric complexity theory VI: the flip via saturated and positive integer programming in representation theory and algebraic geometry. *CoRR*, abs/0704.0229, 2007.

[Mul10] Ketan Mulmuley. Explicit proofs and the flip. *CoRR*, abs/1009.0246, 2010.

[Mul12] Ketan Mulmuley. The GCT program toward the *P* vs. *NP* problem. *Commun. ACM*, 55(6):98–107, 2012.

[MW19] Dylan M. McKay and Richard Ryan Williams. Quadratic time-space lower bounds for computing natural functions with a random oracle. In Avrim Blum, editor, *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, volume 124 of *LIPIcs*, pages 56:1–56:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[Raz92] Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, 1997.

[RY20] Anup Rao and Amir Yehudayoff. *Communication Complexity: and Applications*. Cambridge University Press, 2020.

[Sip06] Michael Sipser. *Introduction to the theory of computation (2nd edition)*. Thomson Course Technology, 2006.

[Yao89] Andrew Chi-Chih Yao. Circuits and local computation. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*, pages 186–196. ACM, 1989.