The State of P vs. NP

Taylor Hornby
taylor@defuse.ca

Term Paper, CS 860,
Prof. Jonathan Buss, Fall 2016,
University of Waterloo.

November 13, 2017

Abstract

The **P** vs. **NP** question is perhaps one of the most important open problems in mathematics. In this paper, we place somewhat-recent circuit lower bound results, proof complexity results, and satisfiability lower bound results in context of the **P** vs. **NP** question. This paper leaves out lots of topics that deserve mention like Geometric Complexity Theory (GCT), approximation algorithms for **NP**-complete problems, and the link between derandomization and circuits. Also included is a discussion of why **P** vs. **NP** is fundamentally hard to answer. Throughout the paper, we suggest ideas for future research.

1 Introduction

Informally stated, the **P** vs. **NP** problem asks if computers can efficiently *solve* all the problems they can efficiently *verify* the solutions to. It is one of the most important open questions in mathematics, being one of the seven Millennium Prize Problems set out by the Clay Institute of Mathematics. It is reasonable to think it is the most important one on the list, since if $\mathbf{P} = \mathbf{NP}$ then the proofs of all the others can probably be found quickly by a computer [9].

Being such an important and central problem, **P** vs. **NP** is poorly understood and progress toward resolving it is slow. One promising strategy for tackling it, Geometric Complexity Theory (GCT), is described as potentially taking "about 100 years... if it works at all" [9]. This paper won't talk about GCT, and it doesn't include an explanation of other important things like the approximation algorithms and heuristics we use to solve **NP**-complete problems in practice, or how an assumption a bit stronger than $\mathbf{P} \neq \mathbf{NP}$ makes randomized computation and deterministic computation equivalent [9]. It will talk about some results in circuit lower bounds, proof length complexity, lower bounding circuit satisfiability, and proving that $\mathbf{P} \neq \mathbf{NP}$ is hard to prove. The goal isn't to be comprehensive, nor to be completely up-to-date. Rather, the goal is to help you—the reader—understand how the theorems relate to the **P** vs. **NP** question.

We'll state most of the theorems without giving their proofs. Our goal is to place them into a broader context: for each one, we want to be clear about why the theorem was worth proving, and how its proof has helped us get closer to resolving **P** vs. **NP**. After we've motivated the importance of the theorems and surveyed the context they live in, we'll explain a select few of the techniques that were used to prove them.

We're assuming the reader has a good understanding of basic complexity theory, and can look up the definitions of complexity classes, machine models, and languages that we don't formally define. If not, we recommend referring to the excellent textbook by Arora and Barak [5] and the Complexity Zoo [1].

Throughout the paper, you'll find ideas for future work in boxes like this one:

Idea 1. Prove $\mathbf{P} = \mathbf{NP}$ and then take over the world.

The next section explains the relationship between some theorems and **P** vs. **NP**. Section 3 presents some of their proof techniques. Section 4 concludes.

2 Results and Directions

One way to understand the entire **P** vs. **NP** ecosystem is to think about how you might prove a hard math theorem yourself. First, in the *understanding-building phase* you'll try to prove the statement itself or a weaker version of it as a warm-up exercise. If you get stuck, then you'll enter the *problem-transformation phase* where you'll think of some things that, if they were true, would be convenient for proving the thing you're trying to prove. After that you'll go through the understanding-building phase again with all of the statements you just came up with in the problem-transformation phase, and so on, recursively, until you've solved the problem.

The point of the problem-transformation phase is to change the problem: make it look different. If you're making no progress at all towards resolving one formulation of the problem, maybe asking the question differently will lead to new insights. After you've made the problem look different, you can try to build up some intuition for the new formulation in a new understanding-building phase. Hopefully you'll find a nice progression from weaker statements to stronger statements: the exercise of proving a weaker one will give you the intuition you need to prove a stronger one. Rinse and repeat until you have enough understanding prove what you actually set out to prove.

The same thing is happening on a large scale with the **P** vs. **NP** problem. This is illustrated in the tree of Figure 1. The root of the tree is $\mathbf{P} \neq \mathbf{NP}$, the statement we'd ultimately like to prove. In the first level of the tree, you can see that we've transformed $\mathbf{P} \neq \mathbf{NP}$ into different-looking statements. The different-looking statements are actually harder to prove, but maybe we'll finally be able to make progress because they look so different. Moving down the tree towards the leaves are weaker statements implied by the ones above. If we follow the branches down far enough, we eventually arrive at statements that are weak enough to prove. This is the understanding-building phase in action: the complexity theorist community is trying to climb up the branches of this tree to attain $\mathbf{P} \neq \mathbf{NP}$. All it takes to prove $\mathbf{P} \neq \mathbf{NP}$ is to find *one* branch that can be incrementally climbed, but so far all the branches seem to have ceilings we can't cross. There are two ways to make progress: climb one step further up a branch, or add an entirely new branch.

Idea 2. It would be interesting to do a sociological study on how groups of mathematicians self-organize to solve hard problems that no one of them can solve on their own. Is the status quo optimal, or is there a better algorithm?

What's illustrated in Figure 1 is just a tiny fraction of what's known about **P** vs. **NP**, and the tree can be drawn in lots of different ways. So this tree isn't necessarily historically accurate, it's just a convenience for placing individual theorems into context. In the next sections, we'll explain each of the branches shown in Figure 1 in more detail and then talk a little bit about why we're having a hard time climbing any higher up the branches.

Idea 3. It would be nice to have an up-to-date and complete version of the tree in Figure 1. Perhaps it would be worthwhile to host an interactive version of this tree on a website and let the community update it as new papers come out.

2.1 Circuit Lower Bounds

The first major branch on the left of Figure 1 corresponds to finding circuit lower bounds. For any polynomial-time deterministic algorithm, there is a family of equivalent polynomial-size boolean circuits (one circuit for each input length). So if $\mathbf{P} = \mathbf{NP}$, then SAT has polynomial-size circuits. One technique for showing $\mathbf{P} \neq \mathbf{NP}$ is to show that SAT doesn't have polynomial-size circuits, or in other words, that $\mathbf{NP} \not\subseteq \mathbf{P}/\mathsf{poly}$.

We've transformed a problem about Turing machines and polynomial running times into a problem about circuit sizes. $\mathbf{NP} \not\subseteq \mathbf{P}/\mathsf{poly}$ is stronger than $\mathbf{P} \neq \mathbf{NP}$, so there's no hope of proving $\mathbf{NP} \not\subseteq \mathbf{P}/\mathsf{poly}$ directly. But now that we are talking about circuits, there are several obvious ways to weaken the statement.

The first way is to show that classes even larger than **NP** aren't contained in $\mathbf{P}/\mathsf{poly}$. For example, we might make progress by first showing $\mathbf{NEXP} \not\subseteq \mathbf{P}/\mathsf{poly}$, then $\mathbf{EXP} \not\subseteq \mathbf{P}/\mathsf{poly}$, then $\mathbf{PH} \not\subseteq \mathbf{P}/\mathsf{poly}$, until finally we have the expertise required to show $\mathbf{NP} \not\subseteq \mathbf{P}/\mathsf{poly}$.

The other way we can weaken the statement is by restricting what kinds of circuits are allowed. Instead of allowing general polynomial-size boolean circuits, we might try bounding the fan-in of the gates and restricting them to logarithmic or even constant

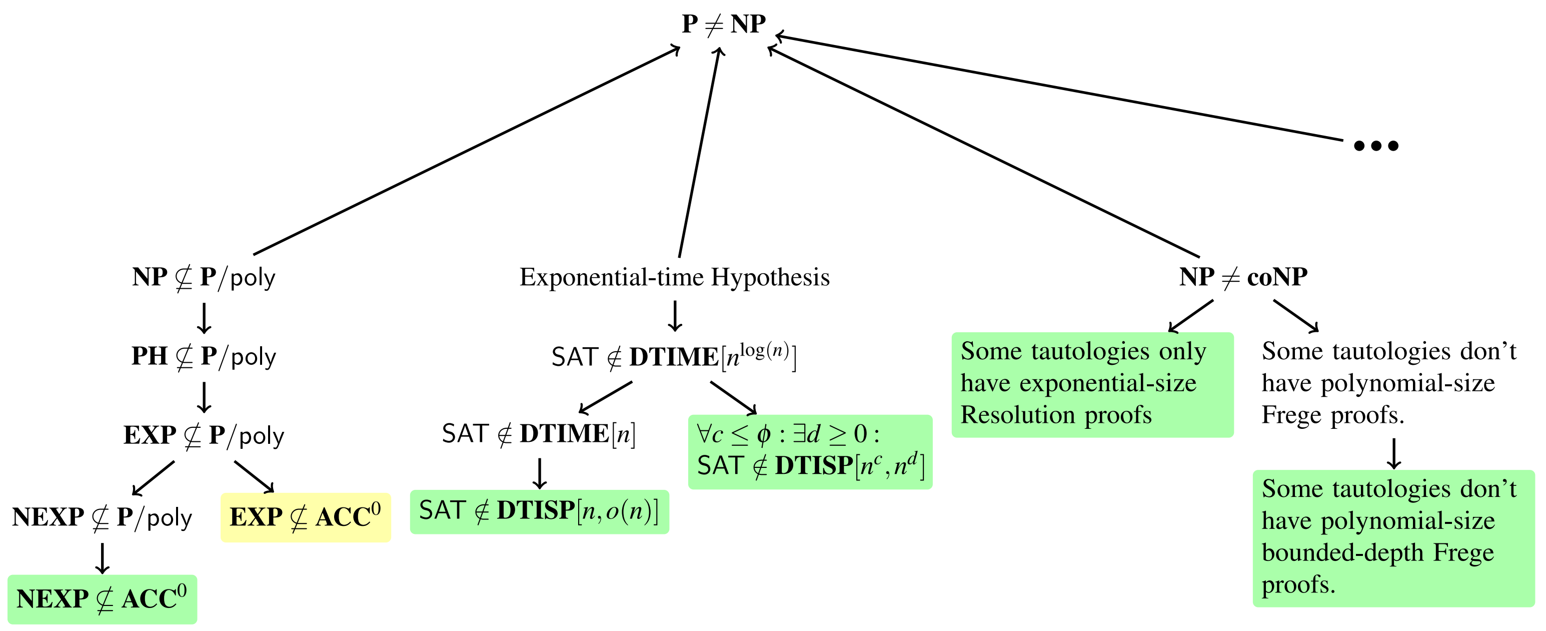
$\mathbf{P} \neq \mathbf{NP}$
$\mathbf{NP} \not\subseteq \mathbf{P}/\mathsf{poly}$
$\mathbf{PH} \not\subseteq \mathbf{P}/\mathsf{poly}$
$\mathbf{EXP} \not\subseteq \mathbf{P}/\mathsf{poly}$
$\mathbf{NEXP} \not\subseteq \mathbf{P}/\mathsf{poly}$
$\mathbf{EXP} \not\subseteq \mathbf{ACC}^0$
$\mathbf{NEXP} \not\subseteq \mathbf{ACC}^0$
Exponential-time Hypothesis
$\mathsf{SAT} \notin \mathbf{DTIME}[n^{\log(n)}]$
$\mathsf{SAT} \notin \mathbf{DTIME}[n]$
$\mathsf{SAT} \notin \mathbf{DTISP}[n, o(n)]$
$\forall c \leq \phi : \exists d \geq 0 :$ $\mathsf{SAT} \notin \mathbf{DTISP}[n^c, n^d]$
$\mathbf{NP} \neq \mathbf{coNP}$
Some tautologies only have exponential-size Resolution proofs
Some tautologies don't have polynomial-size Frege proofs.
Some tautologies don't have polynomial-size bounded-depth Frege proofs.
...

Figure 1: Results and directions toward proving $\mathbf{P} \neq \mathbf{NP}$. Each node of this tree is a statement, and the arrows are logical implication. Near the top of the tree close to $\mathbf{P} \neq \mathbf{NP}$ we're creating *stronger* statements that are *different*. Going down the tree, we're *weakening* the statements until we know how to prove them. A green background means the statement is proved. A yellow background means the statement has unlikely consequences (i.e. we're close to proving it). To prove $\mathbf{P} \neq \mathbf{NP}$, we have to climb all the way up at least one of the branches to root of the tree.

depth. The classes $\mathbf{NC}^k$ are the languages accepted by bounded fan-in polynomial-size circuits of depth $\log^k(n)$. The classes $\mathbf{AC}^k$ are the same but allowing unbounded fan-in. The best we've done in this direction is to show that $\mathbf{NEXP} \not\subseteq \mathbf{ACC}^0$, where $\mathbf{ACC}^0$ is $\mathbf{AC}^0$ augmented with gates MOD_m gates that output 1 when the number of inputs which are 1 is a multiple of m [24]. We're close to showing $\mathbf{EXP} \not\subseteq \mathbf{ACC}^0$; making that assumption entails unlikely results [21].

The most obvious next step is to complete the proof that $\mathbf{EXP} \not\subseteq \mathbf{ACC}^0$. After that, we'll have to either strengthen the circuit model and show something like $\mathbf{NEXP} \not\subseteq \mathbf{AC}^1$, or use a smaller complexity class and show something like $\mathbf{PSPACE} \not\subseteq \mathbf{ACC}^0$.

There's an important connection between finding faster-than-brute force circuit-satisfiability algorithms and showing lower bounds, which is explained in Section 3.3.

2.2 Lower Bounding Deterministic SAT Solvers

The most obvious way to prove $\mathbf{P} \neq \mathbf{NP}$ is to prove that some $\mathbf{NP}$-complete problem like SAT has no polynomial-time algorithm. This is what's shown in the second branch of Figure 1.

The way to weaken this statement is obvious: start by showing there's no linear-time algorithm. Unfortunately, nobody has been able to prove that yet, so we have to weaken the statement even more. Decreasing the allotted running time any more than we already have makes the statement trivially true: the algorithm won't even have time to read the whole input. So, we have to decrease the amount of space allotted.

Kannan showed that SAT $\notin \mathbf{DTISP}[n, o(n)]$, meaning you can't solve SAT in linear time and sublinear space. Fornow et al. strengthened this to to show that for any $c < \phi$, where $\phi = 1.618\ldots$ is the golden ratio, there's some $d > 0$ such that SAT $\notin \mathbf{DTISP}[n^c, n^d]$ [10]. Furthermore if you take the limit as c approaches 1 from above, then d approaches 1 from below, so it's very close but doesn't quite show that SAT $\notin \mathbf{DTISP}[n^1, n^1] = \mathbf{DTIME}[n]$.

To take a step up this branch, it would be nice to finally show that SAT $\notin \mathbf{DTIME}[n]$, which surprisingly doesn't seem to be known despite how close Kannan's and Fortnow's results come [22]. It *is* known that $\mathbf{NTIME}[n] \not\subseteq \mathbf{DTIME}[n]$ [15], but this doesn't imply SAT $\notin \mathbf{DTIME}[n]$ because even if SAT $\in \mathbf{NTIME}[n]$ it might not be one of the languages that separates $\mathbf{NTIME}[n]$ from $\mathbf{DTIME}[n]$. It's strange that based on what we know today, the time it takes to check for satisfiability might be no more than the time it takes to evaluate the formula on a single assignment.

2.3 Proof Length

The third branch of Figure 1 corresponds to the proof complexity approach to $\mathbf{P}$ vs. $\mathbf{NP}$. If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{NP} = \mathbf{coNP}$, so $\mathbf{NP} \neq \mathbf{coNP}$ is stronger than $\mathbf{P} \neq \mathbf{NP}$. What we've done here is transformed **P** vs. **NP** into a different problem about the length of proofs in efficiently-checkable proof systems.

One **coNP**-complete language is TAUTOLOGY. If all tautologies have polynomial-size proofs in some polynomial-time-checkable proof system, then **coNP** = **NP**. So we want to show that there's some family of tautologies that don't have polynomial-size proofs. To weaken this statement, we can consider proof systems that are less powerful than "polynomial-time-checkable." A good place to start is formal logic, which is already in the business of proving tautologies.

A standard kind of proof system that gets used in logic is called a Frege system, and all tautologies can be proven in such systems. By stating the pigeonhole principle in predicate logic, it's possible to construct a family of tautologies whose bounded-depth Frege-system proofs are exponentially long [16]. Before that was proved, it was shown that the same kind of pigeonhole tautologies require exponentially-long proofs in the Resolution system.

To take a step up this branch, we need to consider proof systems that are stronger than Frege systems. Or we could take a step laterally and find a different language somewhere higher up in the polynomial hierarchy that has a natural notion of proof (maybe one related to validities in first-order logic).

2.4 Formal Independence and Proof Techniques that Won't Work

The last few sections have been about the various approaches—branches of the tree—to tackling **P** vs. **NP**. Most of the branches seem to have stalled out; we're having a hard time climbing any higher. Let's now discuss whether there could be a fundamental reason (beyond our own ignorance) that **P** vs. **NP** is hard or impossible to resolve.

Mathematics has its foundation in formal systems like Peano Arithmetic (PA) or Zermelo-Fraenkel set theory (ZF, or ZFC with the axiom of choice). All of the published proofs in mathematics today are thought to be a kind of shorthand for much longer proofs in one of those formal systems. For example, if we take ZFC to be the foundation of mathematics, then in principle we should be able to take any math paper and rewrite all of its mathematical statements and proof steps in terms of the set relation "$\in$" and the axioms of ZFC.

Gödel proved that any sufficiently powerful formal system (e.g. powerful enough to express number theory) is either incomplete or inconsistent. This means that if the formal system is consistent then there are statements which are true but cannot be proven within it. It's easy to get an intuition for why this is the case for formal systems powerful enough to talk about Turing machines. Let $L = \{x \in \{0,1\}^* \mid x \text{ is a Turing machine that halts on the blank-tape input}\}$. Obviously L is undecidable, but for any specific $x \in \{0,1\}^*$ you can write down the sentences "$x \in L$" and "$x \notin L$" in the formal system. One of them is true, so if all true statements had proofs, a Turing machine could just search through all possible proofs until it finds a proof of one or the other and decide accordingly. Thus L would be decidable, which is a contradiction. So there are at least some true statements without proofs.

Is it possible that neither $\mathbf{P} = \mathbf{NP}$ nor $\mathbf{P} \neq \mathbf{NP}$ have proofs in ZFC? It seems strange, but it could conceivably happen. Even if $\mathbf{P} = \mathbf{NP}$ is true, it might not be provable because it might not be possible to prove the polynomial-time algorithm for the **NP**-complete problems is actually correct or runs in polynomial-time. If this is the case, then maybe we can prove that **P** vs. **NP** is independent of the axioms. So far, there's no strong evidence that it is independent of ZFC, but it has been shown to be independent of weaker systems [3].

Another possibility is that $\mathbf{P} \neq \mathbf{NP}$ could be very hard (but not impossible) to prove for some fundamental reason. The shortest ZFC proof might be very long, perhaps too long to fit into the observable universe. Gödel proved his incompleteness theorem by constructing a sentence G referring to itself, which is usally presented informally as "This sentence has no proof." Gödel proved that G has no proof. The statement $\mathbf{P} \neq \mathbf{NP}$ (or more specifically $\mathbf{NP} \neq \mathbf{coNP}$) is a statement about all *efficient* proof systems, so maybe $\mathbf{P} \neq \mathbf{NP}$ or some family of $\mathbf{P} \neq \mathbf{NP}$-like statements have just the right kind of G-like "meta"-ness to require a non-efficient (rather than non-existent) proofs. Again, there's no strong evidence that this is the case. It's more likely that we're just missing some deep insight into computation that would let us prove $\mathbf{P} \neq \mathbf{NP}$.

What we can say for certain, though, is that some of our best techniques for proving things in complexity theory aren't powerful enough on their own to prove $\mathbf{P} \neq \mathbf{NP}$. There are three main barriers: relativization, natural proofs, and arithmetization [3, 4].

The relativization barrier [6] says that a proof of $\mathbf{P} \neq \mathbf{NP}$ or $\mathbf{P} = \mathbf{NP}$ must not relativize. A proof relativizes when, if we give all of the machines in the proof access to some oracle $\mathcal{O}$, the proof still works. The proof that the halting problem is undecidable is an example of a relativizing proof, as are the diagonalization arguments used to prove the time and space hierarchy theorems. We know that the proof resolving **P** vs. **NP** doesn't relativize because there are languages A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ but $\mathbf{P}^B \neq \mathbf{NP}^B$. If the proof of $\mathbf{P} \neq \mathbf{NP}$ relativized, then we could rewrite the proof for machines having an oracle for A and get a proof that $\mathbf{P}^A \neq \mathbf{NP}^A$, which would be a contradiction. Vice-versa if a proof of $\mathbf{P} = \mathbf{NP}$ relativized.

The natural proofs barrier [18] goes like this: If a "natural proof" can be used to prove $\mathbf{P} \neq \mathbf{NP}$, then some of the **NP** problems we think are hard—like breaking psuedorandom functions—are much easier than we think. If we're right that those problems are hard then we won't be able to prove $\mathbf{P} \neq \mathbf{NP}$ with a natural proof. But what's a natural proof and how do we use one to break psuedorandom functions?[1]

A natural proof is one that follows this strategy:

1. Define some complexity measure C on boolean functions.
2. Show that polynomial-time algorithms can only compute boolean functions that have low C-complexity.

[1]This explanation is based heavily on Aaronson's presentation in [3]

3. Show that SAT has high C-complexity.

The only condition on C is that it is computable in polynomial time given the entire 2^n-sized truth table of the boolean function as input.

Now, suppose we have a natural proof of $\mathbf{P} \neq \mathbf{NP}$. Suppose also, as we suspect is the case, that pseudorandom functions (PRFs) exist. Informally, a pseudorandom function is an easy-to-compute function that takes a "seed" parameter s and an input parameter x. The idea is, if someone picks s randomly, then gives you black-box access to either the pseudorandom function's x parameter or a truly-random function, you shouldn't be able to distinguish which is the case. The existence of pseudorandom functions is important to cryptography, since they're the foundation for using a short random key to encrypt huge amounts of data.

But let's suppose there's a family of PRFs that pretend to be random functions from $\{0,1\}^n \mapsto \{0,1\}$ using a seed of size n^c, for some large c. If this PRF family is secure, it can't be broken—distinguished from a truly random function—in less than 2^{n^c} operations, corresponding to brute-force guessing the seed.

Here's how the natural proof lets us break this PRF: Truly random boolean functions will have high complexity in any complexity measure, and in particular C-complexity. But the function that arises by fixing a random seed s for the PRF must have low C-complexity since it's efficiently computable. We can distinguish the two cases—either we're given black-box access to a truly random function from $\{0,1\}^n \to \{0,1\}$ or to the x parameter of the PRF, with the s parameter fixed—by computing its C-complexity. We can make 2^n queries to the black box to learn the entire truth table, and then compute its C-complexity in polynomial time on that table, a total running time of $2^{O(n)}$. But if the PRF family was secure, it should have taken at time at least 2^{n^c}, which is a lot larger. So we've broken the PRF.

What this means is that natural proofs will only work to show SAT lower bounds for classes that don't have secure PRFs. We think $\mathbf{P}$ has secure PRFs, so we can't use a natural proof to prove $\mathbf{P} \neq \mathbf{NP}$.

The algebrization barrier [4] is like a stronger kind of relativization, which I won't explain here because I haven't read the paper and don't understand it.

There's one more kind of barrier. If $\mathbf{P} \neq \mathbf{NP}$ really is independent, then we know there's a barrier to proving *that*, too! See Section 5 of [3]. Let's *hope* that that there's at least a finite n for which the statement $S(n)$ below is provable![2]

$$S(k) = \begin{cases} \text{“}\mathbf{P} \neq \mathbf{NP}\text{”} & \text{if } k = 0 \\ \text{“}S(k-1) \text{ is independent of ZFC”} & \text{if } k > 0. \end{cases}$$

Idea 4. Prove that $S(n)$ is provable for some n.

[2] $S(k)$ for $k \geq 1$ are statements in ZFC by using the Turing machine that checks ZFC proofs.

Idea 5. Determine whether there exists (or doesn't exist) a statement G such that when you replace "$\mathbf{P} \neq \mathbf{NP}$" with G in the definition of S above, $S(k)$ is unprovable for all $k \geq 0$.

3 Interesting Proof Techniques

In this section, we'll present some of the techniques that were used to prove the results we talked about in the last section. Due to time constraints[3], we're focusing on proof techniques that haven't already been explained and re-explained many times over in complexity-theory textbooks.

You won't find complete proofs here; we're only going to present a few small nuggets that are key to understanding the entire proof. The complete proofs can be found in the original papers. The goal here is to help you understand some of the important tricks or the high-level structure without making the whole thing unfun by adding too many details.

3.1 Indirect Diagonalization

As mentioned earlier, we haven't made much progress towards lower bounding the amount of time a deterministic machine needs to decide SAT. If we're willing to constrain the amount of space too, we can actually prove a lower bound. This is done using a technique called indirect diagonalization:

1. Assume the opposite for contradiction, e.g. that $\mathsf{SAT} \in \mathbf{DTISP}[t, s]$, where t and s are the time and space bounds respectively.
2. Use SAT-solving algorithm we just assumed exists to speed up computation enough to contradict a time hierarchy theorem. Usually, this involves speeding up a computation by allowing the machine to alternate [7] (between nondeterminism and conondeterminism) some number of times, and then using the SAT solver to remove the alternations without slowing it down more than it was sped up.

The technique is called indirect diagonalization because the time hierarchy theorems that get contradicted are proved by diagonalization.

For example, here's Kannan's argument (Section 5.1 of [10]) that $\mathbf{NTIME}[n] \not\subseteq \mathbf{DTISP}[n, o(n)]$.

The hierarchy theorem we're going to contradict is that for certain τ, $\mathbf{NTIME}[\tau] \not\subseteq \mathbf{coNTIME}[o(\tau)]$. Let $L \in \mathbf{NTIME}[\tau]$ and $L \notin \mathbf{coNTIME}[o(\tau)]$. Now, for contradiction, suppose there's a deterministic SAT solver that runs in time $t(n) = n$ and $s \in o(n)$. Call this deterministic machine M_{DSAT}.

[3]More correctly referred to as "procrastination."

1. Using M_{DSAT} and the fact that SAT is complete for nondeterministic linear time, we know there is a deterministic machine M_{DL} deciding L in time $t(\tau)$ and space $s(\tau)$.
2. By exchanging the accepting/rejecting states of M_{DL} we get $M_{D\overline{L}}$, a deterministic machine deciding $\overline{L}$ in the same time and space.
3. Using equation (4) from [10] we get a Σ_2 machine $M_{\Sigma_2\overline{L}}$ deciding accepting $\overline{L}$ in time $(t(\tau)s(\tau))^{\frac{1}{2}}$.
4. By Lemma 4.2 from [10], using the existence of M_{DSAT} to satisfy the antecedent of the lemma, we get a nondeterministic machine $M_{N\overline{L}}$ deciding $\overline{L}$ in time $t(n + (t(\tau)s(\tau))^{\frac{1}{2}})$.
5. We can turn that into a conondeterministic machine for L running in the same amount of time by swapping the accepting and rejecting states.

This reduction tells us it's possible to build a conondeterministic machine accepting L that runs in time $T(t,s) = t(n + (t(\tau)s(\tau))^{\frac{1}{2}})$ if a deterministic machine can decide satisfiability in time t and space s. Plugging in $t(n) = n$ and $s \in o(n)$ gives $T(t,s) = n + (\tau o(\tau))^{\frac{1}{2}} = o(\tau)$ when τ is superlinear. That means we're deciding L on a conondeterministic machine in time $o(\tau)$, which contradicts the hierarchy theorem.

Idea 6. [Library Reductions] Each step in the proof above is of the form, "If we have algorithms for deciding these languages in these machine models, then we can make an algorithm for this other language in this other machine model." The way it's presented in [10] goes through statements which are just complexity class containments, so explicit expressions for the time and space used by the new machines are lost. Perhaps it would be useful to give a name to this kind of reduction, and publish them with explicit time and space expressions to make indirect diagonalization proofs easier.

Idea 7. Does this proof technique work for other kinds of SAT solvers, like randomized ones, or quantum ones? Can we use it to prove the same kind of SAT lower bounds for those machines?

The true cleverness of the proof is hidden in steps (3) and (4) above, so what are they? Step (3) is a way of speeding up computation by adding alternations, and step (4) is a way of removing alternations without slowing the computation down too much (assuming you have a fast SAT solver). Step (4) is straightforward, just remove all levels of alternation by implementing the deepest one using the SAT solver until they're all gone. We'll explain step (3).

What we're going to do is speed up deterministic computation by letting the machine alternate. We'll use nondeterminism to guess the configuration that the computation finishes in, as well as the configuration at several evenly-spaced points in time along the way.

Say that we've divided a T-step configuration into b equal blocks of time, so we have the starting configuration C_0, the configuration C_1 at the end of the first block, C_2 at the end of the second block, and so on, until C_b at the end of the last block. After nondeterministically guessing $C_0, \ldots, C_b$, it remains to verify that the computation starting with configuration C_0, actually goes into those configurations along the way, and then finishes in C_b. Without alternations, we'd have to separately check for each pair $(C_0, C_1), (C_1, C_2), (C_2, C_3), \ldots, (C_{b-1}, C_b)$ that the second in the pair is the result of executing the first in the pair for T/b steps. There are b of them, and checking each one takes T/b time; so far we haven't got any speedup. But notice that we do the exact same sort of check for each pair: We take the starting configuration, execute it for T/b steps, and then check that the result matches the finishing configuration. We can use conondeterminism (a $\forall$ quantifier) to check each pair in parallel. So, using nondeterminism and then alternating once to conondeterminism, we've sped up the computation from T to T/b. But, because we have to write down (guess) the b different C_i's, there's an increase in time by bS, where S is the space (configuration size) of the original computation. See Algorithm 3.1.

Algorithm 3.1. Using alternations to speed up computation

```
1: function RunComputation($C_0$, $T$, $S$, $b$)
2: Nondeterministic phase:
3:     for $i = 1$ to $b$ do
4:         $C_i \leftarrow$ guess a configuration of size $S$
5: Conondeterministic phase:
6:     for $i = 1$ to $b$ do                    ▷ To be evaluated with a $\forall$ quantifier
7:         Check that $C_i$ follows from $C_{i-1}$ in $T/b$ steps
8:     return $C_b$
```

Now notice that line 7 of Algorithm 3.1 is a deterministic computation itself, so we can repeat the same thing to speed it up, too. If we fix b, then each layer adds two new quantifiers ($\exists$ then $\forall$), adds bS time steps to guess the configurations, and divides the length of the deterministic computation by b. So if we do it k times, the new running time of the Σ_{2k} machine is $kbS + T/(b^k)$. This is minimized by picking $b = (T/S)^{1/(k+1)}$, which gives

$$\mathbf{DTISP}[T,S] \subseteq \Sigma_{2k}\mathbf{TIME}[(TS^k)^{1/(k+1)}],$$

which is what we used in step (3) of the proof above.

Idea 8. [SAT $\notin$ **DTIME**[n]] By a more precise analysis of the constant and polylogarithmic factors in in [10], we might be able to lower bound the constant factor in a linear-time satisfiability algorithm. Here's a rough sketch of how that might work.

By (12) of [10] and picking $b = (T/S)^{1/(k+1)}$ as is done to obtain (4) of [10], we get

$$\mathbf{DTISP}[T,S] \subseteq \Pi_{k+1}\mathbf{TIME}[(TS^k)^{1/(k+1)}].$$

Note that k was assumed to be constant in the context of [10], so this may hide a factor of k. Now suppose we can solve SAT in time cn, where we'll pick c later. Set $T = 2^n$ and $S = 2^{n/(c+k)}$ for some k that we'll pick later. We have

$$\begin{aligned}\mathbf{DTISP}[2^n, 2^{n/(c+k)}] &\subseteq \Pi_{n+1}\mathbf{TIME}[(TS^n)^{1/(n+1)}]\\ &\subseteq \Sigma_{n+2}\mathbf{TIME}[(TS^n)^{1/(n+1)}]\\ &\subseteq \Sigma_{n+2}\mathbf{TIME}[2^{n/(c+k)}].\end{aligned}$$

(Note that we're now hiding *logarithmic* factors in T because of the hidden constant factor in k). By applying a variant of Lemma 4.3 of [10] for deterministic SAT solvers instead of conondeterministic SAT solvers n times, we might be able to get

$$\begin{aligned}\mathbf{DTISP}[2^n, 2^{n/(c+k)}] &\subseteq \mathbf{DTIME}[c_2^{n+2}c^{n+2}2^{n/(c+k)}]\\ &\subseteq \mathbf{DTIME}[c_2^n c^n 2^{n/(c+k)}]\\ &= \mathbf{DTIME}[2^{n(\log c_2 + \log c + 1/(c+k))}].\end{aligned}$$

The constant c_2 is the one hidden in the statement of the lemma, which we must include because we're applying the lemma a nonconstant number of times. Let's suppose that we analyse the lemma and find that $c_2 < \sqrt{2}$, or that we can force it to be using the linear speedup theorem [2]. Then what we have is

$$\mathbf{DTISP}[2^n, 2^{n/(c+k)}] \subseteq \mathbf{DTIME}[2^{n(\frac{1}{2} + \log c + 1/(c+k))}].$$

We can get an unlikely consequence by, for example, picking $k = 1000$ then solving the inequality $1/2 + \log c + 1/(c+1000) < 2/3$. We find that if we can solve SAT in time cn for any $c < 1.12169$ then

$$\mathbf{DTISP}[2^n, 2^{n/1002}] \subseteq \mathbf{DTIME}[2^{2n/3}],$$

which seems unlikely. Using the linear speedup theorem we can turn a SAT solver running in linear time with *any* constant factor into, say, a $1.10n$-time SAT-solver, so this unlikely consequence would follow from any linear-time SAT algorithm. Note that this is going to be highly sensitive to the machine model; a single logarithmic factor popping up in the application of Lemma 4.3 or in the running time of the SAT solver kills the whole idea.

If we set $T = 2^n$ and $S = p$ where p is a polynomial and proceed the same way, then we get

$$\mathbf{DTISP}[2^n, p] \subseteq \Sigma_{n+2}\mathbf{TIME}[(2^n p^n)^{1/(n+1)}] \subseteq \Sigma_{n+2}\mathbf{TIME}[p].$$

After removing the alternations we have

$$\mathbf{DTISP}[2^n, p] \subseteq \mathbf{DTIME}[c_2^n c^n p] = \mathbf{DTIME}[p 2^{n(\log c_2 + \log c)}].$$

Assuming $c_2 < \sqrt{2}$ is attainable, then if a cn-time SAT solver exists for any $c < \sqrt{2}$, there exists a $\delta < 1$ such that

$$\mathbf{DTISP}[2^n, p] \subseteq \mathbf{DTIME}[p 2^{\delta n}],$$

i.e. exponential-time **PSPACE** algorithms can be sped up.

3.2 Nondeterministic Exponential Time Reduces to Succinct-3SAT

An important result for proving lower bounds like $\mathbf{NEXP} \not\subseteq \mathbf{ACC}^0$ is that Succinct-3SAT is **NEXP**-complete, and that the reduction is very efficient. In the next section, we'll see how to use it to prove a result like $\mathbf{NEXP} \not\subseteq \mathbf{ACC}^0$, but for now, let's focus on how the reduction itself works.

First of all, what is Succinct-3SAT? It's a language of *circuits* which encode exponentially-large 3SAT instances in the following way. A string x is in Succinct-3SAT if x encodes a circuit C with n inputs and $3n+3$ outputs, and when you build a 3CNF formula ϕ with 2^n variables and 2^n clauses according to C—for clause j, the variable numbers and whether or not they are negated are encoded in the $3n+3$ bits $C(j)$—then ϕ is satisfiable.

The first step of the reduction is from nondeterministic time T to a nondeterministic RAM machine running in time $O(T \log T)$. A RAM machine is a machine which has a read-only input tape and oracle-like access to a large bank of random-access memory. The way this works is that the machine has k register tapes and an address tape. All of the tapes except for the read-only input tape are limited to size $O(\log(n))$. The machine can write a memory address a to the address tape and then "read" to branch based on the a-th bit of RAM. Or, it can write an address a onto the address tape and "write" to set the a-th bit of RAM to either 0 or 1. Given a nondeterministic Turing machine that runs in time T, we can make an equivalent nondeterministic RAM machine that's at most a logarithmic factor slower: encode the original machine's configuration into the RAM, and use the address tape to keep track of where its head is. One time step of the original machine can be performed in time $\log(T)$ by reading the symbol under the original machine's head from RAM, writing its new contents to RAM, and then updating the address tape with the new head position. It's the last step—adding or subtracting 1 from the address tape—that takes $\log(T)$ time.

Random access machines are useful for our reduction to Succinct-3SAT because we can separate the $O(\log(n))$-size internal configuration of the machine from the much larger configuration of the machine's random access memory. The internal configuration consists of the machine's finite control's state, the $O(\log(n))$-size contents of the address tape, the $O(\log(n))$-size contents of the k register tapes, and the bit in RAM at the address currently written on the address tape. The key insight is this: If you have a sequence of T internal configurations, supposedly corresponding to a valid computation on some input, you can check that they describe a valid computation without ever having to write down all of the RAM's contents. Following the ideas in [12], that can be done as follows. Starting with a sequence of T timestamped configurations which are initially in the order they occur in the computation,

1. For each sequential pair, check that the second follows from the first, assuming all of the bits that get read from memory are correct.

2. Now check that the bits that were read from memory are correct by sorting according to the contents of the address tape (most significantly), and then by timestamp. Each internal configuration contains a bit b which is a claim for what the bit at the current address (written on the address tape) is. Check this claim for the second in the pair by ensuring either (1) it matches the claim for the first in the pair or (2) it doesn't match the claim for the first in the pair but the first in the pair is performing a write operation to the address with the expected value. Additionally, if the second in the pair is performing a read operation, check that what's read is the same as the claim bit in the same configuration[4].

For the next step of the reduction, we implement the procedure above in a circuit. The input to the circuit is the sequence of internal configurations, and it outputs 1 if the checks pass and the machine accepts or 0 otherwise. The whole circuit has size $O(T\log^{O(1)}T)$ but it can be succinctly described. Just like how Succinct-3SAT are instances of 3SAT whose parts are computed by small circuits, a circuit has a succinct description if its parts (gate type and wiring relationships) can be computed by small circuits. As shown in [12], if we give the circuit some more information in its input to help it do the sorting, the big configuration-checking circuit can be succinctly described by short circuits of a kind even weaker than $\mathbf{NC}^0$.

For the last step of the reduction, we turn the succinctly-described circuit to a succinctly-described 3CNF that's satisfiable if and only if the circuit is, using the usual techniques. It turns out that, because the reduction from circuit satisfiability to 3CNF is so local, we don't need to strengthen the type of circuit we're using for the succinct descriptions.

Altogether, this implies that nondeterministic time $T \geq 2^n$ reduces to 3SAT of size $T\log^{O(1)}(T)$, and the 3SAT instance can be succinctly described by a weak kind of circuit [12].

[4]It's important to include the claim bit in the internal configuration, otherwise information about what was last written to address a could be lost if the machine writes a on its address tape without reading or writing (e.g. if a is a prefix of another address the machine is writing down).

3.3 ACC Circuit Lower Bounds

Here we're going to see how a fast satisfiability algorithm for a kind of circuit helps give a lower bounds for that class. Let's be abstract and call the class of circuits we have a fast satisfiability algorithm for $\mathscr{C}$. Let's call the running time of the satisfiability algorithm (in the number of inputs, not the size of the circuit) T_s. Let $\mathscr{P}$ be the complexity class of languages that are recognized by $\mathscr{C}$-circuits.

Like the results in [21], we'd like to assume something like $\mathbf{EXP} \subseteq \mathscr{P}$ and derive some unlikely consequence. The "unlikely consequence" will be to speed up computation "too much" by adding nondeterminism. To make the same techniques work, let's assume our circuits of interest are not any more powerful than regular ones, i.e. $\mathscr{P} \subseteq \mathbf{P}/\mathsf{poly}$.

The first thing we need is a reduction from deterministic time to Succinct-3SAT. We'll also need that the Succinct-3SAT instances generated by the reduction have succinct witnesses. That means, when it's satisfiable, there has to be some circuit that encodes a satisfying assignment: on input i it returns the truth value to assign to variable i.

In the last section we presented a reduction from nondeterministic time to Succinct-3SAT, but it also works deterministic time to Succinct-3SAT. Unfortunately, it doesn't guarantee its output instances have succinct witnesses. But fortunately, it's shown in [21] that there is a reduction which does under the assumption $\mathbf{EXP} \subseteq \mathbf{P}/\mathsf{poly}$, which is implied by our assumption that $\mathbf{EXP} \subseteq \mathscr{P}$. Let's call the running time of that reduction T_r.

Idea 9. Do all of the Succinct-3SAT instances produced by the reduction in [12] have succinct witnesses under an assumption like $\mathbf{EXP} \subseteq \mathbf{P}/\mathsf{poly}$? The correct values of the extra circuit inputs that help the routing networks sort might not be describable by succinct circuits. If we could show that they always can be, then we could use the reduction in [12] to tighten the results in [21].

Suppose we're given an input x ($n = |x|$) to a T_D step deterministic computation that we'd like to speed up. Then in time $O(T_r)$ we can get a general boolean circuit C_x on n inputs of size $O(T_r)$ describing a large $\mathsf{3SAT}$ instance equivalent to whether or not the T_D-step deterministic computation accepts x.

To speed up the T_D-step deterministic computation, we'll do the following:

1. Use the reduction to get C_x, a circuit of size T_r. (T_r steps)
2. Nondeterministially guess a $(T_r)^d$-size $\mathscr{C}$-ciruit C that's equivalent to C_x. This has to exist because since general circuit evaluation is in $\mathbf{P} \subseteq \mathbf{EXP}$, and by assumption, $\mathbf{EXP} \subseteq \mathscr{P}$. This takes $(T_r)^d$ nondeterministic guessing steps.
3. Check that C is actually identical to C_x using the $\mathscr{C}$ satisfiability algorithm. This takes T_s steps.
4. Nondeterministically guess a $\mathscr{C}$-circuit succinct witness D of size $((T_r)^c)^d$ to C_x. This exists by the same reasoning as in (1). This takes $((T_r)^c)^d$ nondeterministic

guessing steps.

5. Check that D actually describes a satisfying assignment to C using the $\mathscr{C}$ satisfiability algorithm. This takes T_s steps.

The interesting steps are (2) and (4)—we have to encode the two different kinds of checks into the satisfiability of some $\mathscr{C}$ circuit, see [21] for the details. The whole algorithm uses $O(((T_r)^c)^d + T_s)$ steps and $O(((T_r)^c)^d)$ nondeterministic choices. Concretely, for the results in [21], where $\mathscr{C}$ are the $\mathbf{ACC}^0$ circuits, $\mathscr{P} = \mathbf{ACC}^0$, we have $T_r(n) = n^5$ and $T_s(n) = 2^{n-n^\delta}$ for some $\delta > 0$. If we set $T_d = 2^n/n^{10}$ then we get their result that if $\mathbf{EXP} \subseteq \mathbf{ACC}^0$ then there exist constants $\delta > 0$ and c' such that $\mathbf{DTIME}[2^n/n^{10}] \subseteq \mathbf{NTIMEGUESS}[2^{n-n^\delta}, n^{c'}]$. Dividing by n^{10} is necessary to make their reduction to Succinct-3SAT produce a circuit with n input variables (instead of $n + c\log(n)$ input variables).

More information on the link between satisfiability algorithms and lower bounds can be found in [23].

Idea 10. What's the most powerful kind of circuit for which a polynomial-time satisfiability algorithm is known? Circuits with efficient satisfiability algorithms are ones for which important runtime properties can be checked efficiently. For example, if we implemented a parser for a language using such circuits, then it would be possible to determine whether the implementation had a bug for any input of length n. Maybe we could design the languages used in future internet protocols to be parsed by these circuits, so that the most security-critical parts of our computer programs can be automatically checked for exploitable bugs without extra effort on the part of the programmer.

4 Conclusion

We've presented some approaches that have been tried to tackle **P** vs. **NP**, and we've organized some recent results according to which approach they're a part of. We've looked at some of the techniques that were pivotal in proving these results and we've stated several ideas for improving on this work.

This paper is necessarily incomplete. There are many approaches we didn't even try to cover, like Geometric Complexity Theory, and we left out many of the proof techniques that are central to our collective understanding of how computation works. But maybe this paper will be helpful to newcomers getting started in the specific areas we've covered.

References

[1] Complexity zoo. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`. Accessed: 2016-12-18.

[2] Wikipedia: Linear speedup theorem. `https://en.wikipedia.org/wiki/Linear_speedup_theorem`. Accessed: 2016-12-19.

[3] Scott Aaronson. Is P versus NP formally independent? *Bulletin of the EATCS*, 81: 109–136, 2003.

[4] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM Transactions on Computation Theory (TOCT)*, 1(1):2, 2009.

[5] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

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

[7] Ashok K Chandra and Larry J Stockmeyer. Alternation. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 98–108. IEEE, 1976.

[8] Stephen A Cook and Robert A Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(01):36–50, 1979.

[9] Lance Fortnow. The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86, 2009.

[10] Lance Fortnow, Richard Lipton, Dieter Van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *Journal of the ACM (JACM)*, 52(6): 835–865, 2005.

[11] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 59–68. ACM, 1986.

[12] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Local reductions. In *International Colloquium on Automata, Languages, and Programming*, pages 749–760. Springer, 2015.

[13] Richard M Karp and Richard Lipton. Turing machines that take advice. *Enseign. Math*, 28(2):191–209, 1982.

[14] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[15] Wolfgang J Paul, Nicholas Pippenger, Endre Szemeredi, and William T Trotter. On determinism versus non-determinism and related problems. 1983.

[16] Toniann Pitassi, Paul Beame, and Russell Impagliazzo. Exponential lower bounds for the pigeonhole principle. *Computational complexity*, 3(2):97–140, 1993.

[17] Alexander A Razborov. Feasible proofs and computations: Partnership and fusion. In *International Colloquium on Automata, Languages, and Programming*, pages 8–14. Springer, 2004.

[18] Alexander A Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, 1997.

[19] Adi Shamir. IP=PSPACE. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.

[20] Alexander Shen. IP=PSPACE: simplified proof. *Journal of the ACM (JACM)*, 39 (4):878–880, 1992.

[21] Holger Spakowski. On limited nondeterminism and ACC circuit lower bounds. In *International Conference on Language and Automata Theory and Applications*, pages 320–329. Springer, 2016.

[22] Dieter Melkebeek Van. *A survey of lower bounds for satisfiability and related problems*, volume 7. Now Publishers Inc, 2007.

[23] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. *SIAM Journal on Computing*, 42(3):1218–1244, 2013.

[24] Ryan Williams. Nonuniform ACC circuit lower bounds. *Journal of the ACM (JACM)*, 61(1):2, 2014.

[25] Ryan Williams. Strong ETH breaks with merlin and arthur: Short non-interactive proofs of batch evaluation. *arXiv preprint arXiv:1601.04743*, 2016.