P v NP

Jalex Stark

August 19, 2015

1 Ladner's Theorem

As a complexity theorist, when you see a new problem, you have two goals: to find the smallest complexity class it is in, and to find the largest complexity class you believe it's not in. Notice I said "is in" for the first statement but "believe it is in" for the second. There's a reason for that — upper bounds are much easier to prove than lower bounds. If I want to show that a problem is in EXPTIME, I can just write an exponential-time algorithm for it. If I want to show something is in NP, I just write a polynomial-time verifier for it. Proving that no fast algorithm exists is really hard.

Definition 1.1.

$$\mathsf{PRIMES} = \{x \,|\, x \text{ is a prime number } \}$$

Where does PRIMES fit in our hierarchy of complexity classes? Well, trial division easily gets us PRIMES $\in$ EXP. In fact, if we're not picky about what divisors we try, we get PRIMES $\in$ PSPACE. How about PRIMES $\in$ NP? PRIMES $\in$ coNP? In other words, do there exist short proofs of primality? Of compositeness?

A short proof of compositeness is easy — just give a nontrivial divisor. A short proof of primality requires a bit more cleverness. We'll show one called the *Pratt certificate*. It's based on the following converse to Fermat's Little Theorem:

Fact 1.2. *If there is an integer a such that:*

- *a is coprime to n,*
- *$a^{n-1} \cong 1 \pmod{n}$, and*
- *for every prime q dividing $n-1$, $a^{(n-1)/q} \ncong 1 \pmod{n}$,*

then n is a prime number.

So if if I want to quickly prove to you that n is prime, I'll start by giving you such an a. You can easily check the first two conditions (the first by claculating the gcd and the second by repeated squaring), but the last condition requires knowing the prime factorization of $n-1$, which isn't trivial. So a good short proof should also supply this factorization. Then we're done, right?

Example 1.3. Here's a Pratt certificate for 85: $(a = 4, 85 - 1 = 6 \cdot 14)$. We can quickly check that 4 is coprime to 85 and that $4^{85-1} \cong 1 \pmod{85}$. We can also check that $4^{(85-1)/6} \cong 16 \pmod{85}$ and $4^{(85-1)/14} \cong 16 \pmod{85}$. Therefore, 85 is prime!

Okay, so we also need a guarantee that our "prime factorization" is actually made of primes. How do we do this? Provide Pratt certificates for those, too! You might be worried that this recursion grows out of control. On the homework, you'll prove a precise bound on the number of sub-Pratt certificates appearing in a Pratt certificate. For now, though: the length of a prime factorization is basically polynomial in the length of the number. (The number of prime factors of n with multiplicity is bounded by $\log_2 n$, since each prime is at least 2.)

So now we know $\mathsf{PRIMES} \in \mathsf{NP} \cap \mathsf{coNP}$. It is widely conjectured that $P \neq \mathsf{NP} \cap \mathsf{coNP}$ — in other words, that there are problems in $\mathsf{NP} \cap \mathsf{coNP}$ outside of P. Could PRIMES be such a problem? No.

Theorem 1.4 (Agrawal–Kayal–Saxena, 2002)**.** $\mathsf{PRIMES} \in P$

The proof is by explicit construction of an algorithm that runs in roughly[1] $O(n^{12})$ time, and was improved within a few years to roughly $O(n^6)$ time. This algorithm is generally known as the "AKS primality test", in case you want to look it up.

Definition 1.5.

$$\mathsf{INTEGER\ FACTORIZATION} = \{(n, m) \mid \exists d < m[d \mid n]\}$$

Clearly, $\mathsf{INTEGER\ FACTORIZATION} \in \mathsf{NP}$: a short proof is just an appropriate factor. In fact, $\mathsf{INTEGER\ FACTORIZATION} \in \mathsf{coNP}$ by an argument we've already seen: a short disproof is just a prime factorization of n. (We don't need certificates of primality anymore — since $\mathsf{PRIMES} \in \mathsf{P}$, a prime is its own certificate!)

Unlike PRIMES, INTEGER FACTORIZATION is not known to be in P. In fact, it's strongly suspected not to be — many in-practice cryptographic protocols depend on the hardness of this problem. At this point in your complexity career, you're probably used to "hard" meaning something like "NP-complete". There are two good reasons we shouldn't expect such a notion of hardness to apply here:

- The class $\mathsf{NP} \cap \mathsf{coNP}$ probably[2] doesn't have complete problems.
- If a problem in coNP is NP-complete, then $\mathsf{NP} = \mathsf{coNP}$.

This gives us the following:

Proposition 1.6. *If* $\mathsf{NP} \neq \mathsf{coNP}$ *and* $P \neq \mathsf{NP} \cap \mathsf{coNP}$*, then there are problems in* NP *but not in* P *which are not* NP*-complete.*

[1]By "roughly", we mean "up to polylogarithmic factors"

[2]Ask me about "syntactic" and "semantic" classes if you want to know more.

This makes life harder for those trying to show that a problem is easy or hard, but it also makes the world more interesting!

Both of the hypotheses of Proposition 1.6 are generally regarded as true, but it would be nicer if we had a weaker hypothesis that was easier to believe.

Theorem 1.7 (Ladner, 1975)**.** *If* $\mathsf{P} \neq \mathsf{NP}$*, then there is a language* $A \in \mathsf{NP} \setminus P$ *which is not* NP*-complete.*

Proof idea Recall the proof that there exists B such that $P^B \neq \mathsf{NP}^B$. In that proof we started with this language:

$$L(B) = \{x \mid \exists w \in B[|x| = |w|]\}$$

and proved that no matter what B is, $L(B) \in \mathsf{NP}^B$. Then we *diagonalized* B against the languages in P. That is, we picked B cleverly so that for each $A \in P$, $L(B) \neq A$. Now we'll do the same kind of thing. Let's start with this language:

$$A_f = \{x \mid x \in \mathsf{SAT} \text{ and } f(|x|) \text{ is even}\}$$

So as long as f is poly-time computable, this language is in NP. A nondeterministic polynomial time algorithm for A_f is as follows: compute $f(x)$. If it's odd, reject. Otherwise, accept iff $x \in \mathsf{SAT}$.

Now our goal is to construct f cleverly to satisfy the following:

- $A_f \notin P$
- A_f is not NP-complete

Think of this as "blowing holes in SAT". (The "holes" are the strings of lengths n for which $f(n)$ is even.) The language L has "half" of SAT — enough of it that it can't be in P, but not enough of it that we can reconstruct all of SAT from it.

Unlike last time, we have two things to diagonalize against. Let's restate our conditions in a diagonalization-friendly form:

- For all $L \in \mathsf{P}$, $L \neq A$
- For all poly-time computable functions f, f does not reduce SAT to A.

Also unlike last time, our diagonalization needs to be efficent. Last time we set out to find an oracle, which is just a set of languages, so we defined it by induction and threw all worries of efficiency to the wind. This time, our diagonlization is going to be captured by the function f, so we need to be careful about how we go about it.

Set-up Let $\{M_i\}_{i\in\mathbb{N}}$ be an *effective* enumeration of all of the algorithms. By effective, we mean that there is a polynomial time function that, given i, produces the source code for algorithm M_i.

We'll use a fun fact from computability theory: not only does this enumeration exist, but every language that is decided by an algorithm in this enumeration is decided by infinitely many algorithms in this enumeration. More precisely: for all i, there exists $j > i$ such that $\forall x[M_i(x) \Leftrightarrow M_j(x)]$.

Similarly, we can enumerate all of the functions as $\{R_i\}_{i\in\mathbb{N}}$. Now, let's rephrase our conditions on A in terms of the M_i and R_i.

First, $A \notin \mathsf{P}$ iff A is distinct from every language in P. So if we had an enumeration of the polynomial time algorithms (call it $\{P_i\}_{i\in\mathbb{N}}$, we could say

$$A \notin \mathsf{P} \Leftrightarrow \forall i \exists x[A(x) \neq P_i(x)]$$

Problem: we can enumerate all of the algorithms, but we can't enumerate all of the algorithms that halt in polynomial time. (Well, we might be able to, but then our enumeration would take superpolynomial time to compute.)

Solution: we'll put a polynomial clock on each of the M_i so that it runs for no more than n^i steps. Then our enumeration exactly captures the languages in P: if $L \in P$, then $L \in \mathsf{TIME}(n^k)$ for some k. Since each language is represented infinitely often, there is some $i > k$ so that L is decided by M_i. M_i is not affected by the clock, so it decides L. Furthermore, every language decided by a clocked machine is a language in P. Let's do the same kind of clocking for the functions.

A is not NP-complete iff there is no polytime reduction from SAT to A. In other words, A is not NP-complete iff every polytime function fails to be a reduction:

$$A \text{ is not } \mathsf{NP}\text{-complete} \Leftrightarrow \forall i \exists x[A(x) \neq \mathsf{SAT}(R_i(x))]$$

The construction Forget for now about the definition A_f. We'll build A' in stages. We'll systematically go through every string in the language and decide whether each is in A' or not.

At stage $2i$, we'll ensure that A' is not equal to the language described by M_i. To do this, just set $A'(x) = \mathsf{SAT}(x)$ and then check whether $\mathsf{SAT}(x) = M_i(x)$. If so, stay in stage $2i$ and go to the next string. Otherwise, go to stage $2i+1$ with the next string-length, ie. the string $0^{|x|+1}$. **Claim:** stage $2i$ always terminates after considering only finitely many strings.

Proof: Otherwise, we'd have a polytime decidable language that was different in only finitely many places from SAT. But that gives us a polytime algorithm for SAT: memorize the finitely many differences, and apply the algorithm for the polytime language to any string not among those. This contradicts the assumption $P \neq \mathsf{NP}$.

At stage $2i+1$, we'll ensure that R_i is not a reduction from SAT to A'. To do this, set $R_i(x) \notin A'$ and then check whether $x \in \mathsf{SAT}$. If so, move on to step $2i+2$ with string 0^n, where n is greater than $|R_i(x)|$ for each x we considered in this stage. Otherwise, increment x and stay in step $2i+1$. **Claim:** this always terminates after considering only finitely many strings.

Proof: Otherwise, SAT is finite.

Now recall our definition from before

$$A_f = \{x \mid x \in \mathsf{SAT} \text{ and } f(|x|) \text{ is even}\}$$

We now have a clear choice for f: $f(|x|)$ is the stage of the above construction in which we consider strings of length $|x|$. Problem: is f polytime computable? No! If we're given x and want to compute $f(x)$, we need to do the whole construction up to x, which includes solving up to $2^{|x|}$ instances of SAT!

Lazy diagonalization This problem is solvable: let's just slow f down a lot. Let's slow f down enough that at each stage, we can see whether the previous stages have finished.

First, set $f(0) = f(1) = 2$. For $n \geq 1$, define $f(n+1) = f(n)$ if $n \leq (\log n)^{f(n)}$. Otherwise, if n is big enough, we interpret $f(n)$ as the current stage of our construction. We can think of n as a comptuation clock: the bigger n is, the more time we have to check that stage $f(n)$ has been completed.

- If $f(n) = 2i$, check whether there is some x with $|x| < \log n$ such that $x \notin M_i$ but $x \in A$. If there is, then we can say we've completed stage $2i$, so we set $f(n+1) = f(n) + 1$. Otherwise, set $f(n+1) = f(n)$.
- If $f(n) = 2i+1$, check whether there is some x with $|x| < \log n$ such that $x \in \mathsf{SAT}$ and $R_i(x) \notin A$.

We're asking to compute whether strings are in A, but A is defined in terms of f — isn't this circular? It's not circular if the strings we're asking about are of a length for which we've already computed f. Notice that $|R_i(x)| \leq |x|^i$, since R_i runs for only n^i many steps. By construction of f, we have

$$|x|^i \leq (\log n)^i \leq (\log n)^{f(n)} < n,$$

so $f(|x|)$ is always already defined when we ask for it. You should also be worried that we're asking about SAT instances. But we know $\mathsf{SAT} \in \mathsf{EXP}$, and the length of these SAT instances is logarithmic in the length of the input, so we can compute these in polynomial time.

2 Exercises

Exercise 2.1. The Pratt certificate for 229 ($a = 6, 229-1 = 2^2 \cdot 3 \cdot 19$) has 4 sub-Pratt certificates: one for itself, one for 3, one for 19, and one for 3 appearing in the certificate for 19.

Prove that a Pratt certficate for a prime p has at most $4 \log_2 p - 4$ Pratt certificates for primes other than 2. (Hint: strong induction.)

Exercise 2.2. We usually think of the problem of integer factorization as the *function* problem "given n, compute its prime factorization." Prove that our definition of integer factorization as a language (*decision* problem) captures all the hardness of that problem. More precisely, prove that:

- Given oracle access to the function problem, the decision problem can be solved in polynomial time. (Hint: You only need one oracle call)
- Given oracle access to the decision problem, the function problem can be solved in polynomial time. (Hint: Binary search! You only need $O(\log^2(n))$ oracle queries.)

The terminology we're using isn't very precise. Come talk to me if you don't understand what this problem means.

Exercise 2.3 (Actually, this is a hint for exercise 2.4)**.** Let L be a language in NP. Recall that there is a nondeterministic polynomial time machine M such that for each $x \in L$, there is some sequence of nondeterministic choices on which M accepts x, and that for each $x \notin L$, M rejects x on every sequence of nondeterministic choices.

Consider the machine M' obtained by flipping the result of M on each computation path — that is, M' rejects every pair of (input, sequence of nondeterministic guesses) on which M accepts and vice versa. What is the relationship between the language decided by M and the language decided by M'?

Exercise 2.4. Define a "strong" nondeterministic algorithm as follows: on each pair of (input, sequence of nondeterministic guesses) it does one of {accept, reject, return '?'}. Say that a strong nondeterministic algorithm M accepts x if on every sequence of nondeterministic guesses, it accepts or returns '?'. Similarly, say M rejects x if on every sequence of nondeterministic guesses, it rejects or returns '?'.

Prove that the class of problems decided by polynomial time strong nondeterministic algorithms is exactly $\mathsf{NP} \cap \mathsf{coNP}$.

3 Mahaney's Theorem

Definition 3.1. A language L is *sparse* if $c(n) = |\{x \in L \,|\, |x| \leq n\}|$, the number of strings in the language of length at most n, is bounded above by a polynomial.

We can think of a sparse language as having very little expressive power.

Theorem 3.2 (Mahaney, 1982)**.** *If a sparse language is* NP*-complete, then* $P = \mathsf{NP}$*.*

This is a hard proof, so we'll introduce the ideas in several steps. We'll start with the following puzzle.

Consider a full binary where some nodes are labeled with a dot and some nodes are labelled with arrows. Only leaves may be labelled with dots, and it is possible that no leaf has a dot. A node has an arrow pointing a child node iff that node has an arrow or a dot. (A node may have 0 or 1 or 2 arrows.)

Problem 3.1 (Easy)**.** You want to decide whether the tree has any dot-marked leaves in it. You have a marker which starts at the root of the tree. One computational step consists of looking at the label of the node you are currently at, and then choosing to move to one of the child nodes or the parent node. You can remember as much information as you like. Design a deterministic algorithm which halts in a number of steps polynomial in the depth of the tree that either produces a marked node or decides that none exist.

Okay, that was silly.

Problem 3.2 (For real this time)**.** Take the same set up as before, but now each node has a color which obscures the arrows. (You can still tell whether a marked leaf is marked.) Each node with an arrow has a different color from any node without an arrow. Design a deterministic algorithm which halts in a number of steps polynomial in the depth of the tree and the number of colors that either produces a marked node or decides that none exist.

Solution: Do a depth-first search of the tree. (That is, From each node, first recursively search th eleft subtree, then recursively search the right subtree.) At each node, note the color. If you've seen a color more than depth-of-tree many times, ignore that node.

Proof of correctness. When the depth-first search finds an x-leaf, it does so along the leftmost branch of arrows. Therefore, the algorithm never sees more than depth-many arrow nodes. Any color seen at least that many times may safely be assumed to not be an arrow-node color. We never have to search a subtree whose root is a non-arrow node, since we know such a subtree contains no x-leaves. Since we see each non-arrow color at most depth-many times and we see at most depth-many arrow colors, our algorithm is polynomial in the number of colors and the depth of the tree. □

Definition 3.3. A language L is *unary* if $L \subseteq 1^* = \{1^n \,|\, n \in \mathbb{N}\}$.

Theorem 3.4 (Berman, 1978)**.** *If a unary language is* NP*-complete, then* $P = \mathsf{NP}$*.*

Proof. Let ϕ be an instance of SAT. We aim to give a polynomial time algorithm to decide whether ϕ is satsfiable. We'll build an object known as the *self-reduction tree* of ϕ and then apply Problem 3.2.

Start with a full binary tree of depth n, where ϕ has n arguments. Label the root with $\phi\,(x_1, x_2, \ldots, x_n)$. Label the left child of the root $\phi(0, x_2, \ldots, x_n)$ and the right child with $\phi(1, x_2, \ldots, x_n)$. In general, the children of a node should be labeled with the formula of the parent node with one variable substituted by a constant. The leaf nodes of this tree should be constant Boolean expressions. Furthermore, they are all of the possible assignments to ϕ — ϕ is satisfiable iff any of the leaf nodes are labeled with a true formula.

Let R be a reduction from SAT to our unary language L. At each node, apply R. If the result is 1^n, interpret this node as being colored by the n^{th} color. If the result is not unary, then we know the formula at the ndoe is not satisfiable, so interpret this as a bad color. At a leaf, evaluate the formula. If it's true, treat it as a marked node. Otherwise, treat it as an unmarked node.

We've exactly recreated the situation from problem 3.2, so the same algorithm will find a satisfying assignment or prove none exist. $\square$

This can be strengthened:

Definition 3.5. If L is a language, $c_L(n) = \#\{x \in L \mid |x| \leq n\}$. L is *sparse* if there is a polynomial p such that $\forall n[c_L(n) \leq p(n)]$.

Theorem 3.6 (Mahaney, 1982)**.** *If S is sparse and* coNP*-complete, then* $\mathsf{P} = \mathsf{NP}$*.*

We'll give a sketch of the proof before we dive in:

1. Pad S, preserving NP-completeness and sparseness.
2. Construct an auxiliary language $\hat{S} \in \mathsf{NP}$ which is sort of like the complement of S.
3. By the NP-completeness of S, obtain reductions from $\hat{S}$ to S and $\overline{\mathsf{SAT}}$ to $\overline{S}$.
4. "Compose" these reductions in various ways to get many "candidate reductions".
5. Run the candidate reductions in parallel to get a real reduction from $\overline{\mathsf{SAT}}$ to S.
6. Apply Lemma 3.7.

Lemma 3.7 (Fortune, 1979)**.** *If S is sparse and* coNP*-complete, then* $P = \mathsf{NP}$*.*

Proof. Let R be a reduction from $\overline{\mathsf{SAT}}$ to S. We'll give a polytime algorithm for SAT in the same way as Theorem 3.4 — we'll traverse a self-reduction tree, using R to provide colors. Every node that is not an arrow node is labeled with a formula that is an instance of $\overline{\mathsf{SAT}}$, so all of the colors of nonarrow nodes lie in S. Since S is sparse, there are only polynomially many non-arrow colors, so the marked-node finding algorithm works. $\square$

Proof of Mahaney's Theorem. **Claim:** If p is any polynomial function and S_p is a padding of S by p, then S_p is sparse and NP-complete.

Proof: Define $S_p = \left\{x\#^k \,\middle|\, x \in S \text{ and } \left|x\#^k\right| \le p(|x|)\right\}$. The identity function is a reduction from S to S_p, so S_p is NP-hard. "Drop all of the #" is a reduction from S_p to S, so $S_p \in \mathsf{NP}$. Finally, $c_{S_p}(n) \le nc_S(n)$, so S_p is sparse.

Later in the proof, we're going to define a polynomial p and assume that S was actually S_p all along. Along the way, we'll only use the sparseness and NP-completeness of S, so this is a valid technique.

Claim: $\hat{S}$ defined below is in NP

$$\hat{S} = \left\{(x, 1^k) \,\middle|\, k < c_S(|x|) \text{ or } (k = c_S(|x|) \text{ and } x \notin S)\right\}$$

Proof: We'll describe an NP algorithm that decides the complement of S under the assumption that $k = c_S(|x|)$, and then it will accidentally decide exactly $\hat{S}$.

Algorithm: First, nondeterministically guess k distinct strings of length at most $|x|$, rejecting if any of them lie outside of S. Then accept if none of them are x. **Analysis:** If $k < c_S(|x|)$, then some branch successfully guessed k strings which are in S but not equal to x. Therefore, the algorithm accepts. If $k = c_S(|x|)$, then the branches which haven't rejected after the first stage have guessed exactly the strings in S. Then it accepts iff $x \notin S$. If $k > c_S(|x|)$, then no branch can guess k distinct strings in S, so the algorithm rejects.

Candidate reductions: S is NP-complete, so let U be a reduction from $\hat{S}$ to S and let T be a reduction from SAT to S. Let p be such that $|T(\phi)| \le p(|\phi|)$. Now replace S with the padded S_p, and modify T to pad its output such that $|T(\phi)| = p(|\phi|)$.

Define $R_k(\phi) = U(T(\phi), k)$. R_k acts like a reduction from $UN\mathsf{SAT}$ to S, but only on inputs of length ϕ. That is, if $k = c_S(|\phi|)$, then $R_k(\phi) \in S \Leftrightarrow \phi \in UN\mathsf{SAT}$.

A poly-time algorithm for SAT: Let $q(n)$ be a polynomial bound for $c_S(p(n))$. That is, let q be large enough so that after we hit our formulas with T, q is a bound on how many formulas we have of at most a given length.

Let ϕ be a formula. For each $k \le q(|\phi|)$, run the algorithm from the proof of the lemma using R_k as the reduction. All but one of these (the one with $k = c_S(|T(\phi)|)$) are not the true reduction, instead outputting useless garbage. However, if any of them find a satisfying assignment, then it's actually a satisfying assignment, so ϕ is actually satisfiable. If none of them find a satisfying assignment, then in particular, the true reduction didn't find a satisfying assignment, so ϕ is not satisfiable.

Therefore, this algorithm decides SAT in polynomial time. We conclude $\mathsf{P} = \mathsf{NP}$. $\square$