CS 4349 Lecture—November 13, 2017

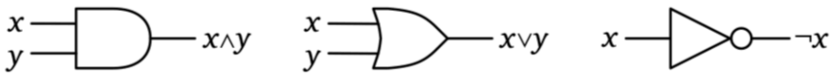
Main topics for #lecture include #PvsNP and #NP-hardness.

Prelude

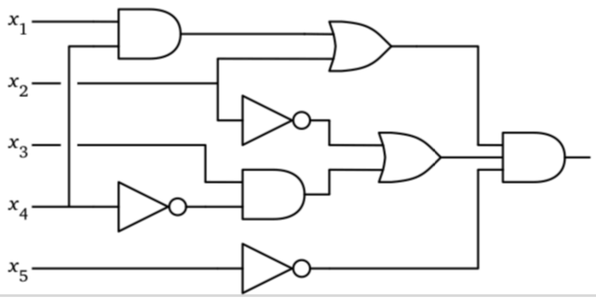
- Homework 10 due Wednesday, November 15th.

Circuit Satisfiability

- As usual, let's start the lecture by defining a problem, *circuit satisfiability* or *CircuitSAT* for short.
- You're given a *boolean circuit* consisting of AND, OR, and NOT gates. The circuit has several inputs, x_1 through x_n. The circuit has a single output. Imagine there's a light bulb at the end.



An AND gate, an OR gate, and a NOT gate.

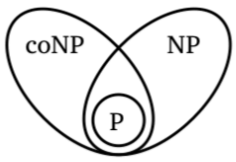


- The ultimate goal is to decide if there is a way to set the inputs so the lightbulb turns on.
- Now, if you *propose* a setting for the inputs, then checking if those turn the bulb on isn't too hard. For example, true true false false false leaves the bulb turned off.
- In short, you topologically sort the gates, and then evaluate their outputs in that order. It only takes linear time to check a setting of the inputs.
- So to solve our original problem, we could just try every combination of settings. But there are 2^n of them, and this algorithm would take exponential time.
- And here's the thing. I don't know of *any* efficient algorithm for this problem that's significantly better than trying every combination of settings! In fact, I don't believe there even *is* efficient algorithm for this problem.

P vs NP

- This whole semester, we've been talking about ways to design efficient algorithms for various problems, but now we're going to do something a bit different.

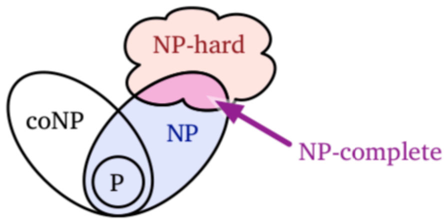
- This week, we're going to discuss what kinds of problems even have efficient algorithms.
- To make this precise, we need to make a few things concrete.
- First, we need a convenient definition of efficient: we'll consider algorithms that run in *polynomial time* or O(n^c) for some constant c.
- Next, we'll start by discussing a specific type of problem. A *decision problem* is a problem where the output is a single boolean value: True or False.
- There are three classes of decision problems that theorists particularly care about.
    - P: decision problems you can solve in polynomial time
        - In other words, we can solve these quickly.
    - NP: decision problems where if the answer is yes, there is a proof of the fact you can verify in polynomial time.
        - In other words, we can verify somebody else's solution quickly.
    - co-NP: decision problems where if the answer is *no*, there is a proof of the fact you can verify in polynomial time.
- CircuitSAT is in NP. You are trying to decide if there's a way to turn that bulb on. And if there is one, then you can prove it by telling me the settings for the inputs. I'll verify the settings in linear time.
- Every decision problem in P is also in NP, because we can verify True answers in polynomial time using an empty proof by just solving the problem from scratch. Similarly, every decision problem in P is also in co-NP.
- OK, so this point might seem strange to you if you've heard some misleading things about these classes before or you're confused on what NP stands for. The N doesn't stand for *not*. It stands for *non-deterministic*. If you've seen non-deterministic Turing machines, maybe in 4384, then these problems can be solved by a non-deterministic Turing machine where all computation paths take a polynomial number of rounds. If you haven't seen them, just remember: **the N doesn't stand for** *not*
- The most important question in theoretical computer science, if not all of computer science, is whether P and NP are actually different.
- So think about your homework for example. Hopefully, you find that the posted solutions make sense. Intuitively, this means solving homework problems is in NP. On the other hand, it's much harder to come up with the solutions from scratch, so it makes sense that P and NP would be different classes.
- Most theorists believe P does *not* equal NP, but we don't know how to prove it! In fact, the Clay Mathematics Institute lists P versus NP as one of its seven Millennium Prize Problems: Only one of these has been solved, and if you solve any of the others, including P versus NP, then there's a 1 million dollar reward for the solution.
- We don't even know if co-NP and NP are different classes. We think so, but we have no proof.



What we *think* the world looks like.

NP-hard and NP-complete

- So NP contains some easy problems, those in P, and some harder problems.
- We call a problem Pi NP-hard if a polynomial time algorithm for Pi implies there is a polynomial time algorithm for *every* problem in NP. These problems are *harder* than problems in NP.
- In other words, Pi is NP-hard if and only if a polynomial time algorithm for Pi implies P = NP.
- We don't think P = NP, so NP-hard problems *probably* don't have polynomial time algorithms.
- Finally, a decision problem is *NP-complete* if it is both NP-hard and in NP.



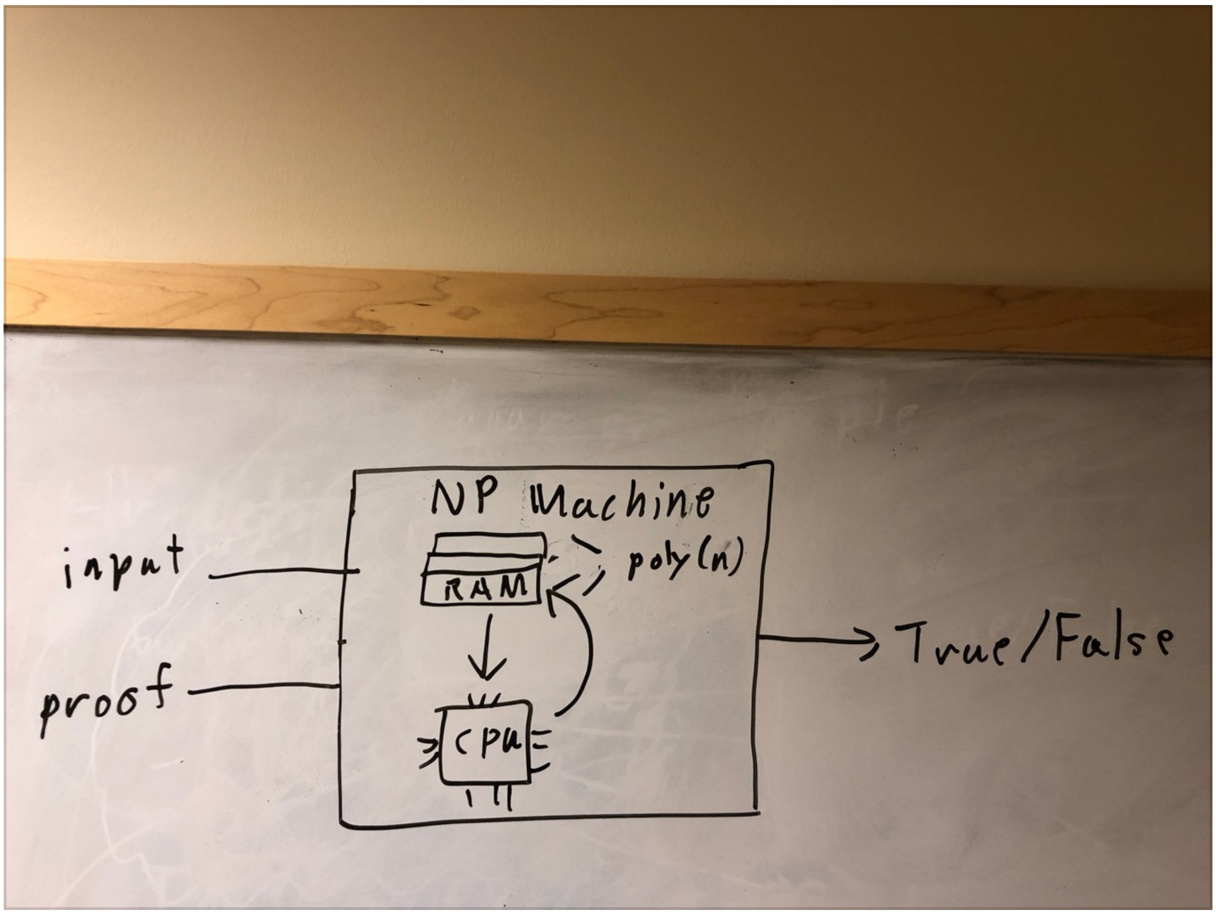
More of what we *think* the world looks like.

- These are the hardest problems in NP. If we can solve any of them in polynomial time, then we can solve *all* problems in NP in polynomial time.
- But it's not even clear there are NP-complete problems. We're asking for a problem so difficult that us solving it efficiently means we can solve all NP problems efficiently, but we also need there to be easily verifiable proofs to yes answers to that problem.
- But it turns out there are actually thousands of NP-complete problems that people really do care about.
- The first of these problems were found by Steven Cook and Leonid Levin in the early 1970's. In fact, we've already seen that first problem.
- Cook-Levin: Circuit satisfiability is NP-complete.

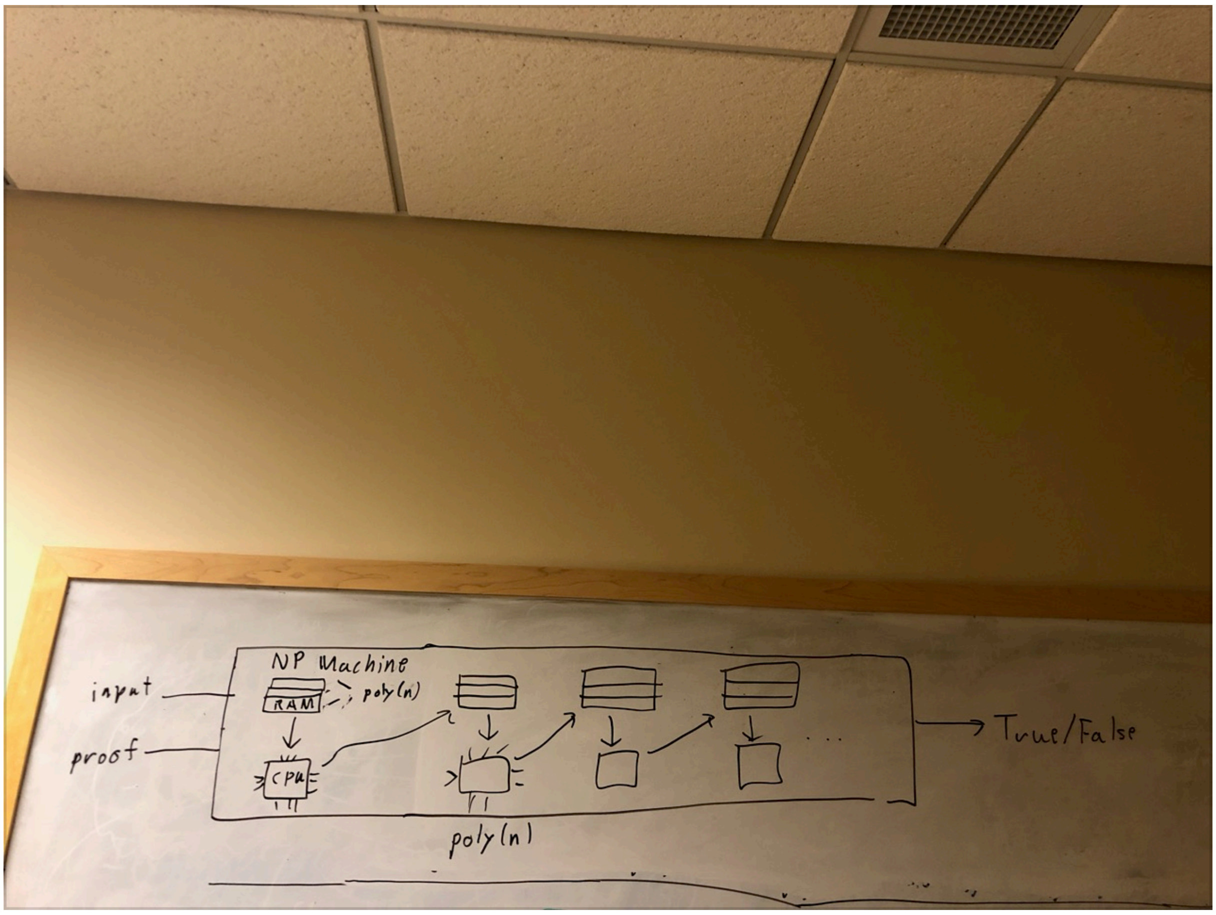
Sketchy Sketchy Proof Sketch of Cook-Levin:

- We already saw CircuitSAT is in NP.
- I'm going to show how a polynomial time algorithm for CircuitSAT gives you a polynomial time algorithm for any problem in NP.

- So, suppose we're trying to solve some problem in NP. You can verify yes answers to that problem in polynomial time given a proof, so imagine we have a custom built computer just for verifying those yes answers.



- It takes the input to the problem and the proof on the left and outputs True or False on the right depending on whether or not the proof is legit.
- Inside the computer we have a CPU and some RAM. Every clock cycle, the CPU pulls data from the RAM, computes something, and puts it back.
- The machine only has polynomial time to run, so it can only use a polynomial amount of RAM.
- Now, suppose instead of running a clock, we just buy a ridiculous number of CPUs and RAM chips and link them in sequence like this.



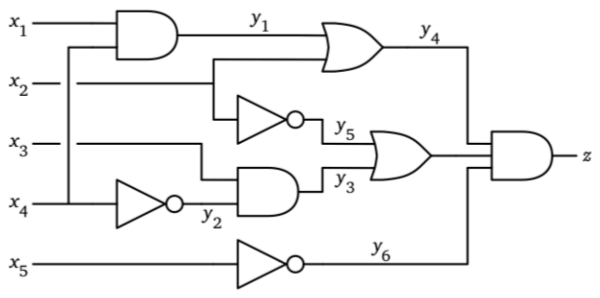
- The whole thing should finish in poly(n) clock ticks, so we *only* need a polynomial number of chips.
- We've essentially turned our custom machine into one big boolean circuit. As the final step, we hardcode the input and make the proof wires into our circuit's only inputs. We can get the circuit to output true if and only if the original NP problem had a yes answer.
- So figuring out whether or not we can output true with our circuit is the same as solving the NP problem.

Reductions and SAT

- Fortunately, it's a lot easier to show other problems are NP-hard.
- We use something called a *reduction argument*.
- Remember, a reduction from problem A to B is an algorithm for A that assumes an algorithm for B exists.
- To prove that problem Pi is NP-hard, reduce a known NP-hard problem **to** Pi in polynomial time.
- The direction here is essential. You solve the **known hard problem** using the new problem as a subroutine. Yes, this may seem backwards.
- But what you get is essentially like a proof by contradiction. If new problem Pi were easy, then the known hard problem would also be easy, because we can solve it using Pi.
- Here's an example. *formula satisfiability* or SAT: Given a boolean formula like (a v b v c v

not(d)) if and only if ((b = c) v not(not(a) ➜ d))), can you assign boolean values to the variables a, b, c, … so that the formula evaluates to True?

- We can show SAT is NP-hard by reducing from a known NP-hard problem.
- But, we only know of one NP-hard problem: CircuitSAT. So we reduce *from* CircuitSAT to SAT.
- We create a new variable for the output of each gate, write out the list of gates separated by ANDs, and put the last output at the end, since we want it to be true.



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

- This formula is satisfiable if and only if the circuit is satisfiable.
  - Given an assignment for the circuit, compute all the y and z values to find a way to satisfiable the formula.
  - Given a way to satisfy the formula, just grab its x values to satisfy the circuit.
- We just need to to a depth-first search from the inputs of the circuit to compute the formula, so the reduction takes linear time, which is polynomial.

$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{boolean formula}} \arrow[d, Rightarrow, "\text{SAT}"] \\
\boxed{\text{True or False}} & \boxed{\text{True or False}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

- So if we have a polynomial time algorithm for SAT, then we have a polynomial time algorithm for Circuit-SAT, which would imply P = NP. SAT is NP-hard.
- Finally, we can verify a yes answer to SAT by just checking an assignment of the variables in linear time, so SAT is in NP.
- It is NP-hard and in NP, so it is NP-complete.