CS 6363.005.19S Lecture 21—April 9, 2019

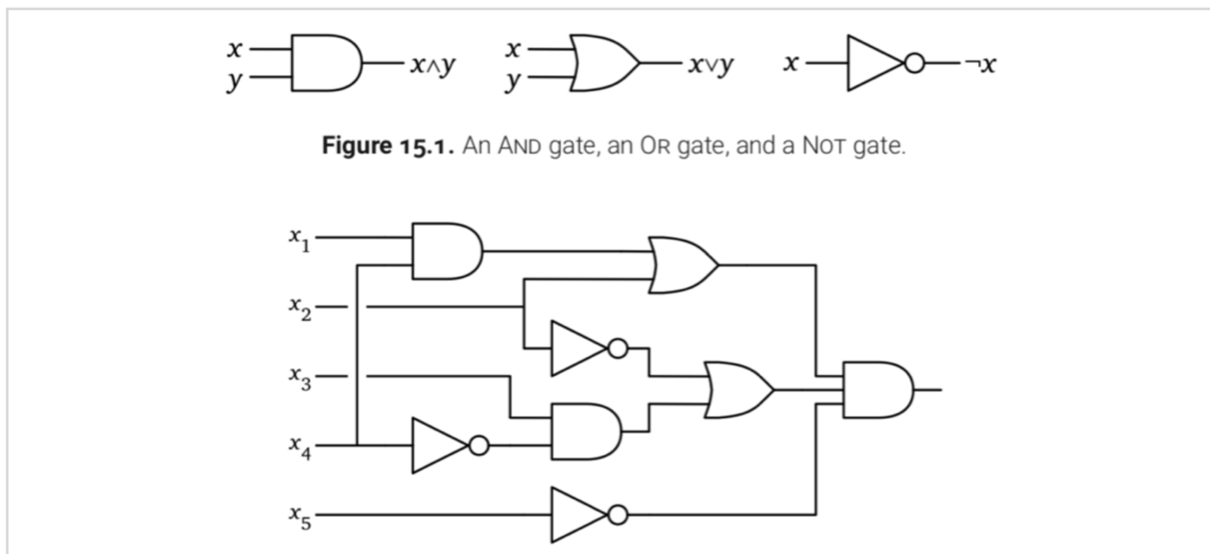
Main topics are #NP-hardness.

Prelude

- Homework 4 is due today; grace period ends 10am Thursday.
- Midterm 2 is Tuesday April 16.

Circuit Satisfiability

- Let's start with a puzzle.
- I've drawn a boolean circuit with AND, OR, and NOT gates. If I treat the gates as vertices in a graph, it's organized as a DAG.



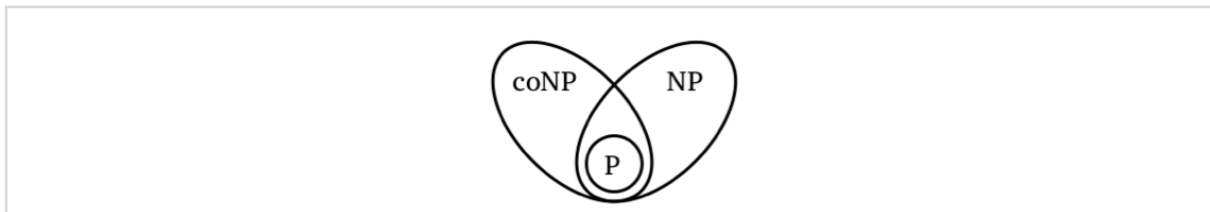
- There are five inputs, and one output represented as a lightbulb. Is there an assignment of True and False values to the inputs to turn on the lightbulb at the end? Go ahead, I'll wait.
- It turns out, yes, we can turn on that lightbulb. You just need to use these inputs:
  - x_1 = False
  - x_2 = True
  - x_3 = True
  - x_4 = False
  - x_5 = False
- This is an instance of a problem called *circuit satisfiability* or CircuitSAT. You have gates arranged as a DAG and n inputs. Can you output True?
- It's really easy to check a proposed setting of the inputs, in linear time even. Just evaluate the output of each gate in topological order.
- In particular, it's easy to convince somebody that you *yes, you can* turn on the bulb just by telling them how to set the inputs.

- But actually finding a way to set the inputs is really hard. The only algorithm I know of is just to try all 2^n ways to set the inputs, and that's really slow!
- Maybe there's a better algorithm, but nobody has found it yet. But for reasons I'll explain today, computer scientists have been trying really really hard to solve this problem faster for half a century. I personally don't believe we can do much better.

P versus NP

- So far this semester, we've been discussing efficient algorithms for a variety of problems and some of the algorithm design techniques we know of for finding those algorithms.
- A minimal requirement for an algorithm to be*efficient* is to have a running time of O(n^c) for some constant c; i.e., polynomial time.
- There's a deep theory concerning *problems* that have polynomial time algorithms.
- This theory is mostly about *decision problems*. A decision problem takes its input and outputs a single value, Yes or No. CircuitSAT is a decision problem. Can we turn on the lightbulb?
- There are three *classes* of decision problems that we really care about.
  - P: Decision problems we can solve in polynomial time.
    - We can solve these "quickly".
    - For example: Does the minimum spanning tree have total weight at most k?
  - NP: Decision problems where *if the answer is yes*, there is a proof that you can verify in polynomial time. You can also discount bad proofs in polynomial time.
    - So somebody can convince you the answer is Yes in polynomial time.
    - For example: CircuitSAT.
  - co-NP: Decision problems where *if the answer is no*, there is a proof that you can verify in polynomial time.
    - Essentially the opposite of NP.
    - For example: Prime: Given an n-bit number, is it prime?
- There's a common misconception that I want to address about what these acronyms stand for. Obviously, P stands for Polynomial. The NP stands for Non-deterministic Polynomial.
- If you've seen non-deterministic Turing machines, say in Theory of Computation, then this acronym may make some sense. It's the set of decision problems decidable by a non-deterministic Turing machine where *every* computation path take polynomial time. The "proof" I mentioned before is the choice of non-deterministic state transitions.
- If you've not seen Turing machines, maybe think about "non-deterministic" meaning you can try every possible polynomial length proof, non-deterministically, in polynomial time.
- To further emphasize what NP doesn't stand for, it's easy to see that every problem in P is also in NP. You can verify Yes answers in polynomial time by just solving the problem from scratch. So MST < k? is in NP also.

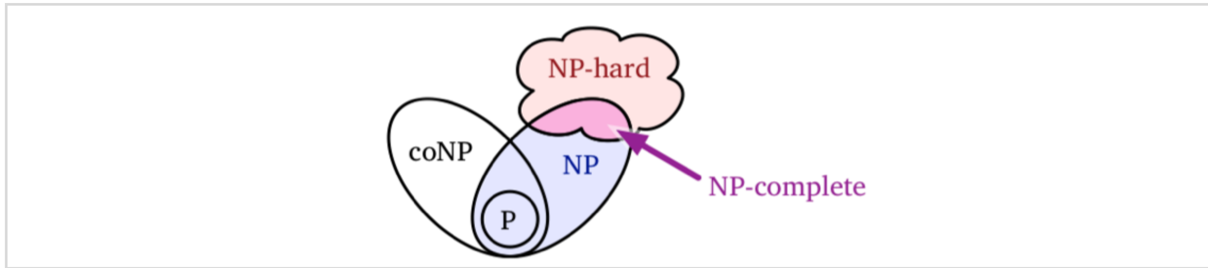
- And more generally, P is in NP. One of the most important problems in computer science, if not all of *science* is whether or not P is actually *equal* to NP. If I can *verify* an answer, or mathematical proof, or efficacy of a drug, etc. easily, can I actually *find* the answer, proof, or drug to begin with?
- I believe P and NP are probably different. Take your homework for example. Many of these problems are *hard*. They take a long time to solve. But once you see the solution, I hope they much easier to understand.
- So most people think P and NP aren't the same set, but we have no mathematical proof. Maybe there are fast algorithms for every problem in NP, but we just haven't found them yet. And it would be a big deal if we found them. Science, and especially math, would progress faster if P = NP. On the other hand, internet commerce would be more dangerous, because most cryptography schemes work under the assumption that you can't solve certain problems in polynomial time.
- P vs. NP is such an important question that the Clay Mathematics Institute lists P versus NP as the first of its seven Millennium Prize Problems, only one of which have been solved so far. If somebody finds a proof that P = NP or P ≠ NP, then they win a $1,000,000 reward.
- There's one more subtle but open problem here. Is co-NP equal to NP? If I can prove a Yes answer, can I also prove a No answer? Probably not, but we don't know for sure!
- So again, here's what we think the world looks like.



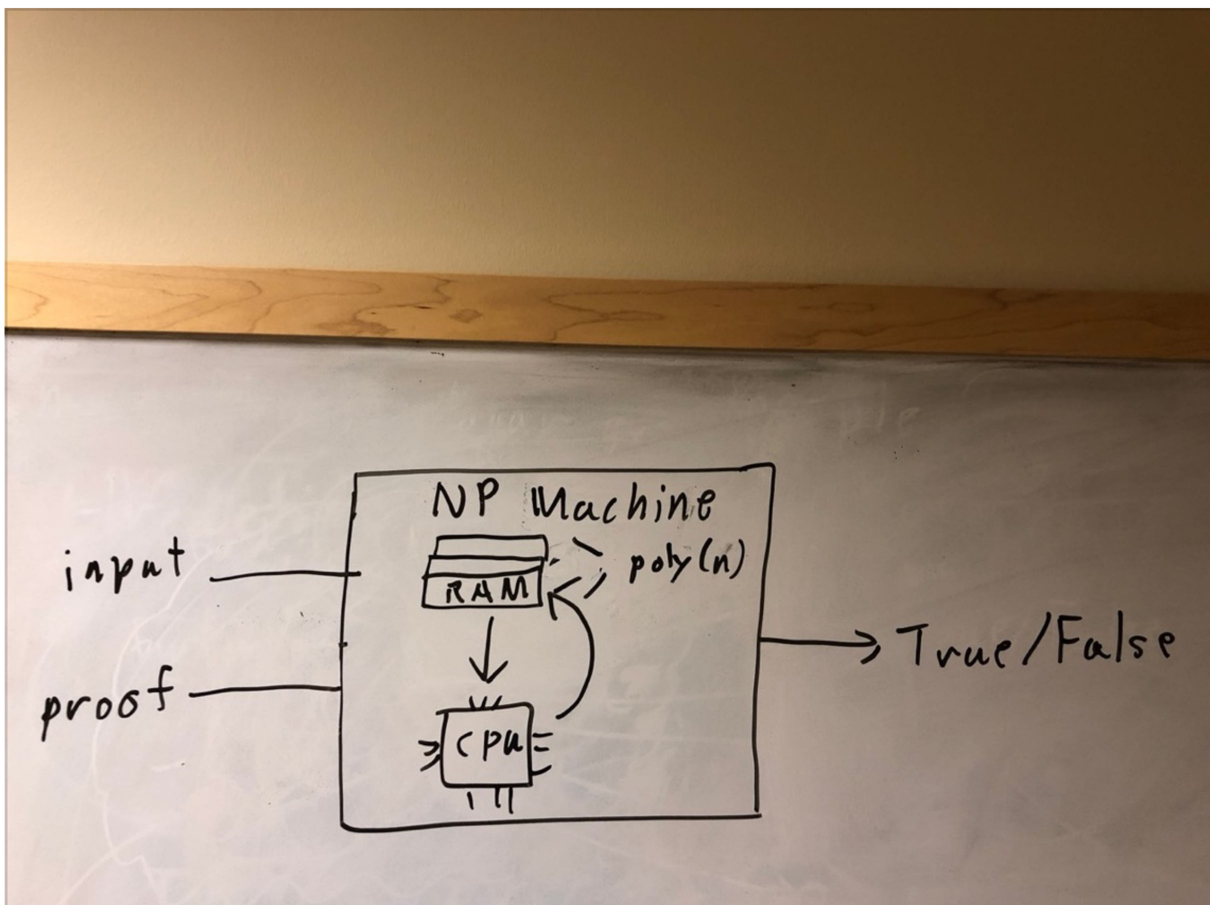
NP-hard and NP-complete

- So P is the set of easy problems. But the rest of today and next week, I want to focus on the *hard* problems.
- A problem A (decision or not) is NP-hard if we can reduce any problem in NP to A with polynomial time overhead.
- In other words, if we can solve A in polynomial time, we can solve any problem in NP in polynomial time as well. A is *as hard* as every problem in NP.
- So a polynomial time algorithm for A would imply P = NP.
- Now, most people don't believe P = NP, so a problem being NP-hard means the problem probably most likely most certainly *doesn't* have a polynomial time algorithm. So don't bother trying to find one.
- Finally, a decision problem is NP-complete if it is in NP *and* it is NP-hard. So it's as hard as every other problem in NP. There are thousands of known NP-complete problems, and we've yet to find polynomial time algorithms for any of them. So again, P probably doesn't

equal NP.



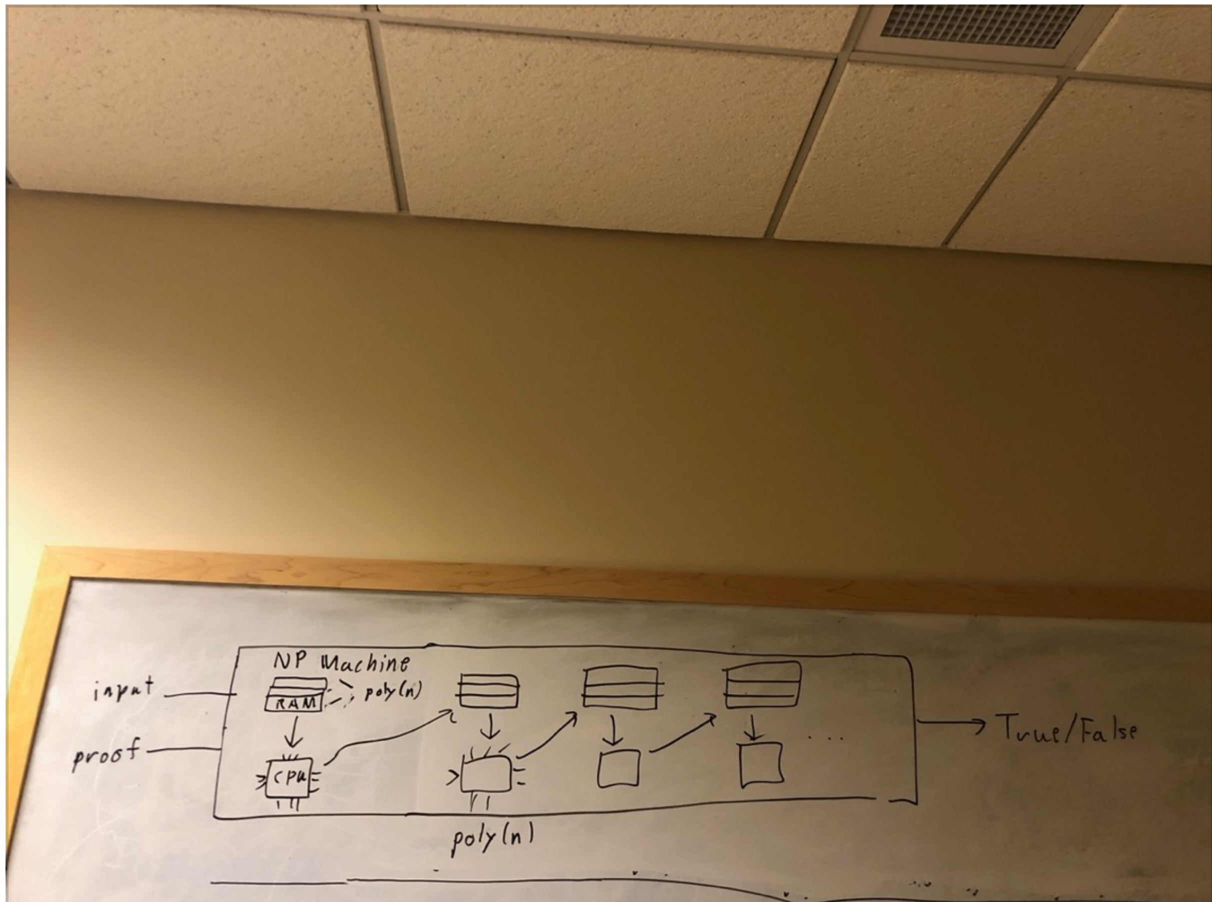
- But how do we know all these problems are NP-complete? It all comes down to a famous theorem by Steven Cook ('71) and Leonid Levin ('73).
- The Cook-Levin Theorem: CircuitSAT is NP-complete.
- I've already argued that CircuitSAT is in NP. A formal proof that it's NP-hard is beyond the scope of this course (that's what Theory of Computation is for), but I want to give a hint as to why you can use it to solve problems in NP. This proof sketch is super sketchy.
- 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 the result back in.
- The machine only has polynomial time to run, so it can only use a polynomial amount of

RAM and the loop happens a polynomial number of times.

- 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.



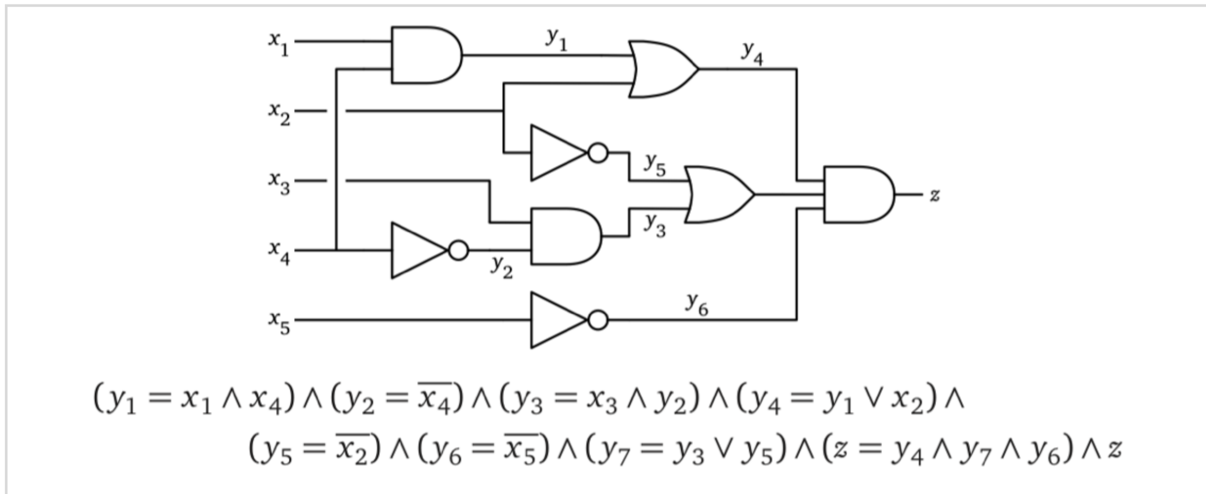
- The original machine uses only poly(n) clock ticks, so we *only* need a polynomial number of chips.
- We've removed the loop and essentially turned our custom machine into one big acyclic boolean circuit. As the final step, we hardcode the input and use the proof wires as our *circuit's* only inputs. We can get the circuit to output true if and only if there was some Yes proof for the NP machine.
- If there's a polynomial time algorithm for CircuitSAT, then it will take polynomial time in the original problem size when given this giant circuit. So we'd have a polynomial time algorithm for our original NP problem.

Reductions and SAT

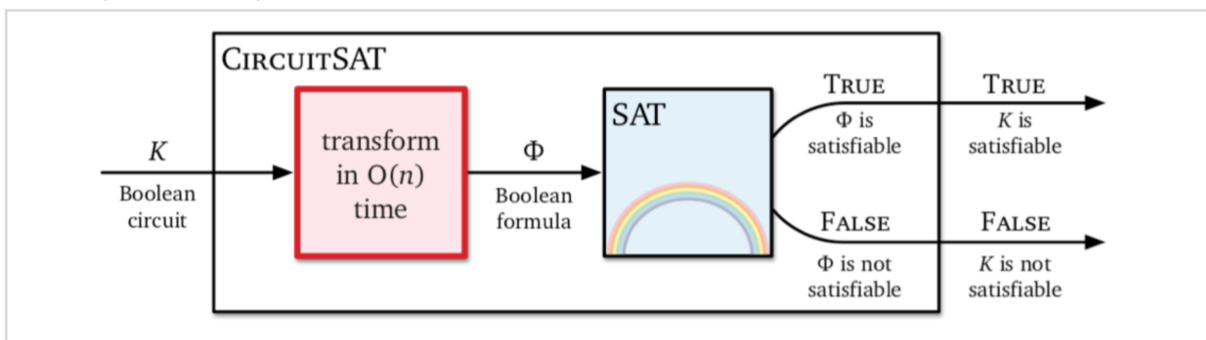
- Remember, there are *a lot* of NP-hard problems.
- Fortunately, it's a lot easier to show these other problems are NP-hard.
- We use something called a *reduction argument*.
- Remember, a reduction from problem A to B is an algorithm for A that assumes an algorithm for B exists.
- To prove that problem B is NP-hard, reduce a known NP-hard problem A **to** B in

polynomial time.

- The direction here is essential. You solve the **known hard problem** using the new problem as a subroutine. THE OTHER DIRECTION DOES NOT WORK.
- 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.



- 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 satisfy the formula.
  - Given a way to satisfy the formula, just grab its x values to satisfy the circuit.
- We just need to consider the gates in topological order to compute the formula, so the reduction takes linear time, which is polynomial.
- To summarize, suppose we have a magic polynomial time algorithm for SAT. We can take an input to CircuiSAT, reduce it to SAT in linear time, and then output the answer given by the magical SAT algorithm.



- The total running time of our CircuitSAT algorithm is O(n) + however long it takes the magical SAT algorithm to run on an O(n) size input. So polynomial time total.

- So if we have a polynomial time algorithm for SAT, then we have a polynomial time algorithm for Circuit-SAT, so we have a polynomial time algorithm for every other problem in 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.

3SAT

- Let's look at another NP-complete problem. This one is a special case of SAT called 3SAT.
- First, some definitions you may have seen.
- A *literal* is a boolean variable or its negation (a or not(a)).
- A *clause* is a disjunction (OR) of several literals (b or not(c) or not(d))
- A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several clauses ((a or b or c or d) and (b or not(c) or not(d)) and (not(a) or c or d) and (a or not(b)).
- A 3CNF formula is a CNF formula with exactly three literals per clause. So this example is not a 3CNF formula since the first and last clauses have the wrong number of literals.
- 3SAT: Given a 3CNF formula, is there an assignment of the variables that makes the formula evaluate to True?
- 3SAT looks like it should be easier than general SAT since I'm restricting what types of inputs you get, but it turns out the problem is still NP-hard.
- Remember: To prove NP-hardness, you need to reduce *from* a known NP-hard problem *to* your new problem.
- We'll use a reduction directly from CircuitSAT to show 3SAT is NP-hard. This should be the last time we reduce directly from CircuitSAT.
- Given a boolean circuit:
  1. Change it so every AND and OR gate has only two inputs. If a gate has k > 2 inputs, replace it with a binary tree of k - 1 two-input gates.
  2. Write down the circuit as a formula with one clause per gate. Just like in the reduction to SAT.
  3. Change every gate clause into a CNF formula.

$$\begin{aligned} a = b \wedge c &\quad\longmapsto\quad (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\quad\longmapsto\quad (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\quad\longmapsto\quad (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

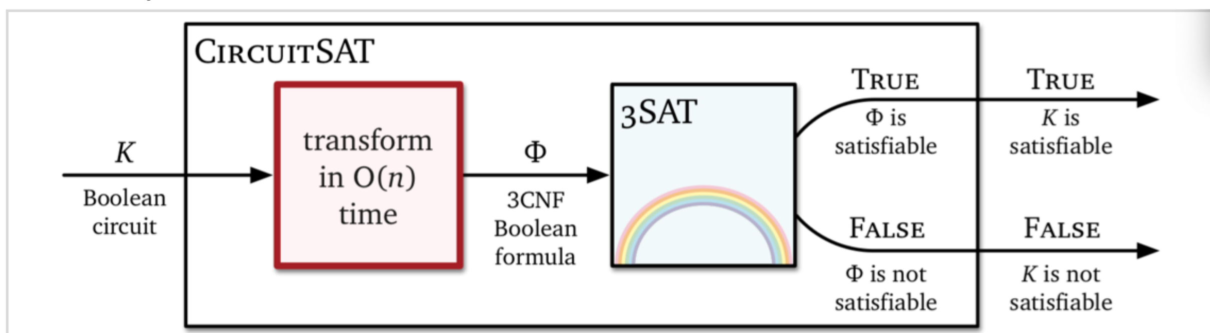
  4. Make sure every clause has exactly three literals by introducing new literals for every one and two-literal clause and expanding them into new clauses.

$$
\begin{aligned}
a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\
a &\longmapsto (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y})
\end{aligned}
$$

- Here's the 3CNF formula you get from our favorite circuit.

$$
\begin{gathered}
(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\
\wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\
\wedge (y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\
\wedge (\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\
\wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\
\wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\
\wedge (\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\
\wedge (y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\
\wedge (y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \\
\wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}})
\end{gathered}
$$

- Yeah, that's gross, but it's only a constant factor larger than the original circuit, and you can compute it in polynomial time.
- In summary, here is what our reduction looked like:



- So a polynomial time algorithm for 3SAT gives a polynomial time algorithm for CircuitSAT and therefore any problem in NP. 3SAT is NP-hard.
- It is also in NP, so 3SAT is NP-complete.
- Next time, we'll see some NP-hard problems that aren't based on circuits!