CS 6363.003.21S Lecture 21—April 20, 2021

Main topics for #lecture include #NP-hardness.

Efficient Algorithms and P vs. 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.
- But there are some problems for which we simply don't know of a polynomial time algorithm!
- For example, here is a boolean circuit with AND, OR, and NOT gates. If you treat the gates as vertices in a graph, it's organized as a DAG.

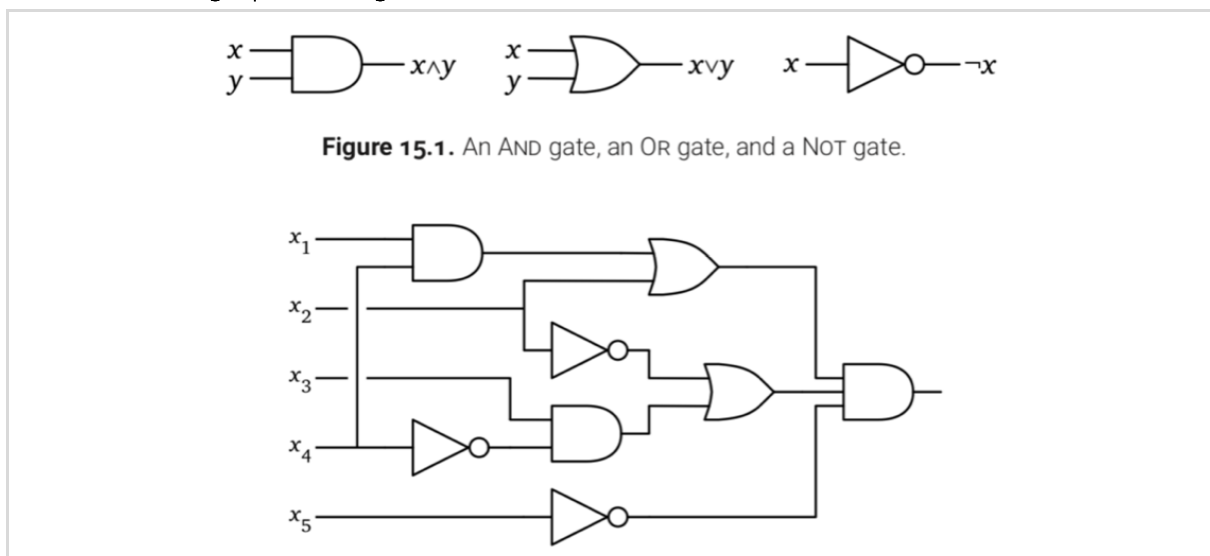


Figure 15.1. An AND gate, an OR gate, and a NOT gate.

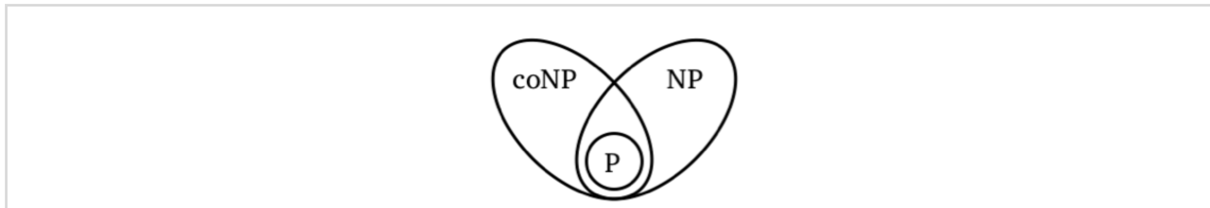
- There are five inputs, and one output. I want to know: is there an assignment of True and False values to the inputs to set the output to True?
- It turns out, yes, we can make the output True for this particular circuit. We can use these inputs:
  - x_1 = False
  - x_2 = True
  - x_3 = True
  - x_4 = 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?
- Strangely, it's really easy to check a proposed setting of the inputs, in linear time even: Find a topological ordering of the gates and evaluate their outputs in that order.
- In particular, it's easy to convince somebody that *yes, you can* output True 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. And this is in spite of there being a fast way to verify assignments.
- CircuitSAT is just one example from a deep theory concerning *problems* that have polynomial time verification procedures but not necessarily polynomial time algorithms.
- This theory is mostly about *decision problems*. A decision problem takes its input and outputs a single value, True or False. CircuitSAT is a decision problem. Can we have the circuit output True?
- 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 true*, 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 True in polynomial time, but they can't trick you.
        - For example: CircuitSAT.
    - co-NP: Decision problems where *if the answer is false*, 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? You can prove the answer is false by showing me a divisor other than 1 or the input number.
- Naturally, P in all these acronyms stands for Polynomial. The N is less obvious, and it actually stands for Non-deterministic.
- If you've seen non-deterministic Turing machines in a theory of computation course 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 takes 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, in parallel, in polynomial time per proof.
- Being in NP *does not* imply exclusion from P. In fact, every problem in P is also in NP. You can verify True answers in polynomial time by just solving the problem from scratch. In other words, the "proof" may as well be empty. No, don't worry about showing me a light spanning tree. I can find one myself if it exists.
- So P is a subset of NP. One of the most important problems in computer science, if not mathematics, if not all of *science* is whether or not P is actually *equal* to NP. If we can *verify* an answer, or mathematical proof, or efficacy of a drug, etc. easily, can we actually *find* the

answer, proof, or drug to begin with?

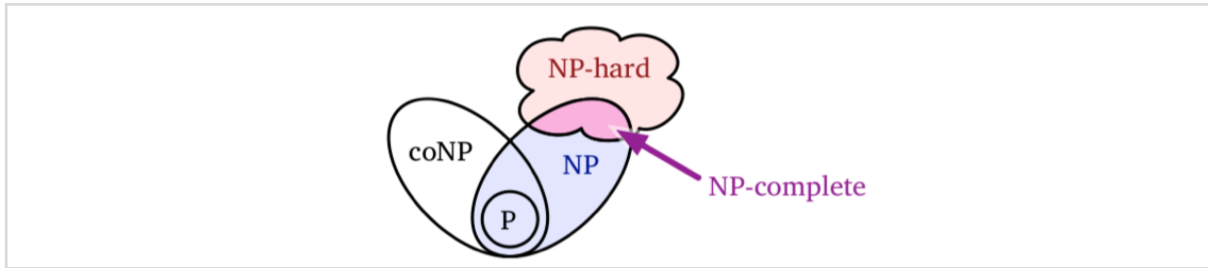
- I believe P and NP are probably different. Take your homework for example. Many of these problems are *hard*. They take a long time to solve. But once you see the solution, I hope they become much easier to understand.
- Most computer scientists think P and NP aren't the same set, but we have no mathematical 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 and private communications might suffer, because most classical 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 provide a proof when the answer is True, can I also provide a proof whenever the answer is False? Probably not, but we don't know for sure!
- So again, here's what most computer scientists think the world looks like. But there's no proof that all three sets aren't the same after all.



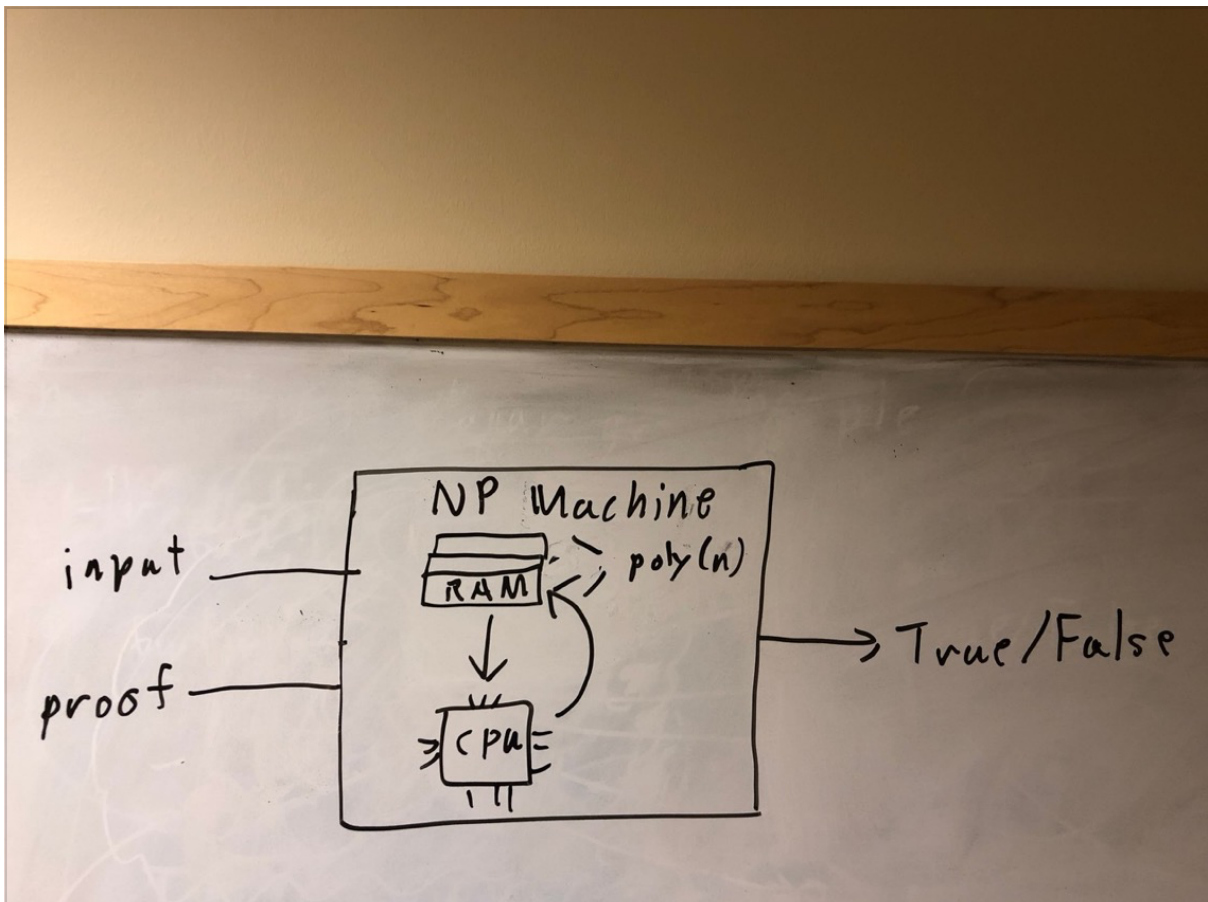
NP-hard and NP-complete

- So P is the set of easy problems. But the rest of this week, I want to focus on the *hard* problems.
- Problem B (decision or not) is NP-hard if we can reduce *every* problem A in NP to problem B with polynomial time overhead.
- Which implies if we can solve B in polynomial time, we can solve every problem in NP in polynomial time as well. B is *as hard* as every problem in NP.
- So a polynomial time algorithm for B 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 a proof that B is NP-hard can be used as evidence that you shouldn't bother trying to find a polynomial time algorithm.
- Finally, a decision problem is NP-complete if it is in NP *and* it is NP-hard. So despite being in NP, it is 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.

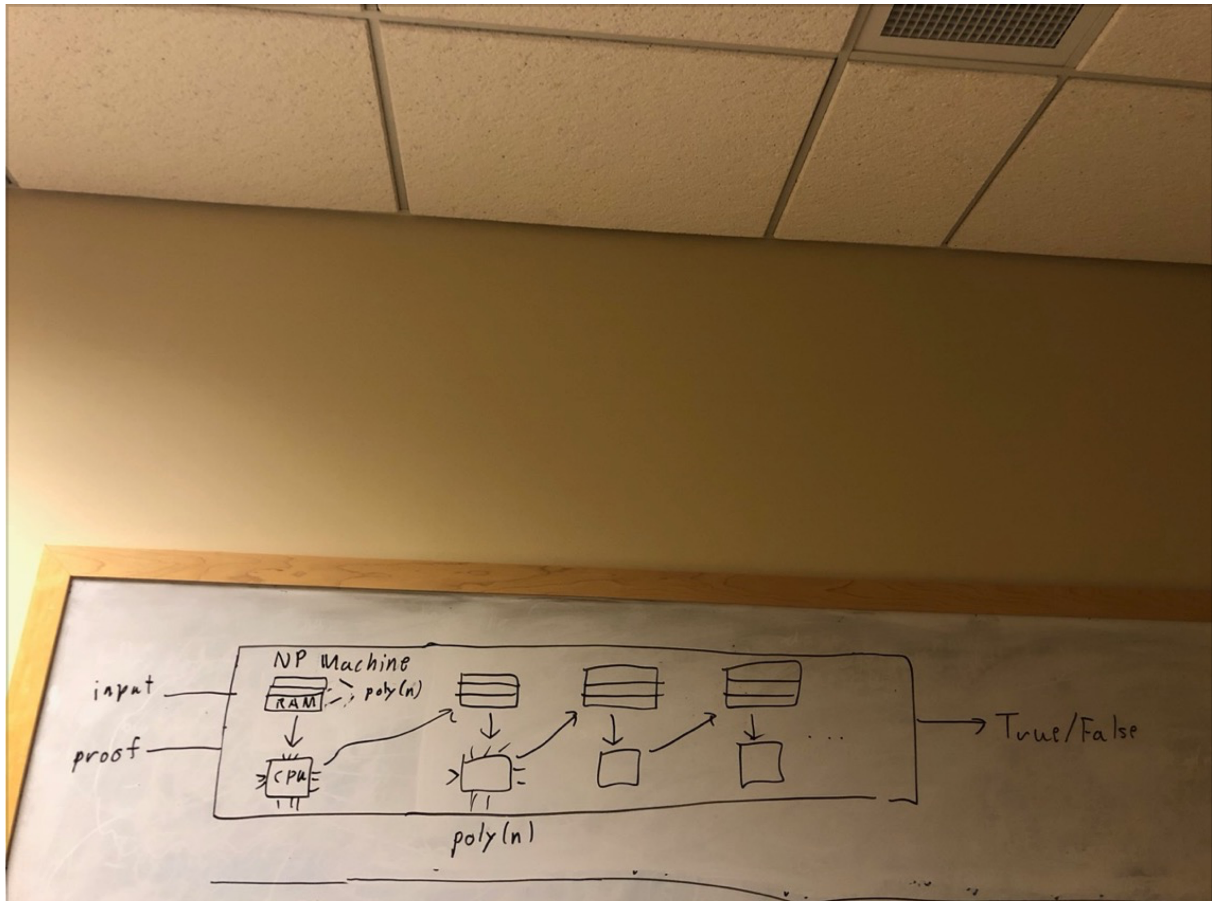


- But how do we know about all these NP-complete problems? It all comes down to a famous theorem by Steven Cook ('71) and Leonid Levin ('73).
- The Cook-Levin Theorem: CircuitSAT is NP-complete.
- I've already argued that CircuitSAT is in NP. A formal proof that it's NP-hard is too tedious to bother with as part of this course, but I want to give a hint as to why you can reduce problems in NP to CircuitSAT.
- Let A be an arbitrary problem in NP. You can verify inputs to A where the answer is True in polynomial time given an appropriate proof, so imagine we have a custom built computer just for seeing if a given proof matches up with a given input to A. We'll call this computer the *verifier*.



- The verifier takes in the original input to A along with a proof. It outputs True on the right if and only if we give it a legitimate proof that the instance of A leads to a True output.
- 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 take advantage of a polynomial amount of RAM a polynomial number of clock cycles.
- Now, suppose somebody only gives us an input to problem A. We want to decide in polynomial time if the answer is True, which is equivalent to deciding if there exists a proof that our machine can verify. Our algorithm modifies the verifier machine.
- Instead of running a clock, we'll buy a ridiculous number of CPUs and RAM chips and link them in sequence like this.

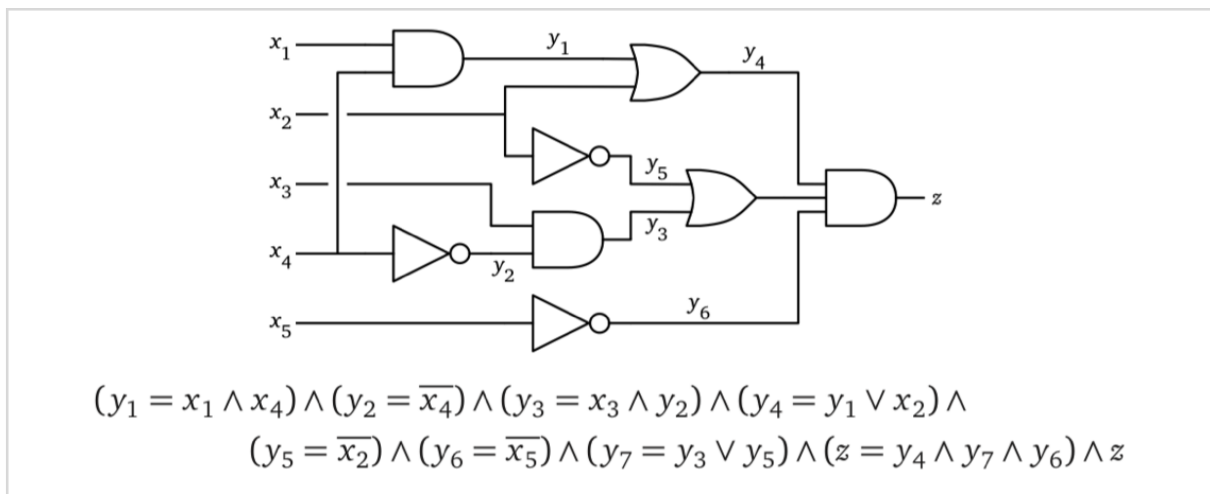


- The original machine uses only poly(n) clock ticks, so we *only* need a polynomial number of chips.
- We've removed the loop and essentially turned our verifier machine for A into one big acyclic boolean circuit. As the final step, we hardcode the input into this giant circuit.
- At this point, the proof wires are the only input to our circuit. We can get the circuit to output True if and only if there was some True proof for the original verifier machine and the original given input.
- 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

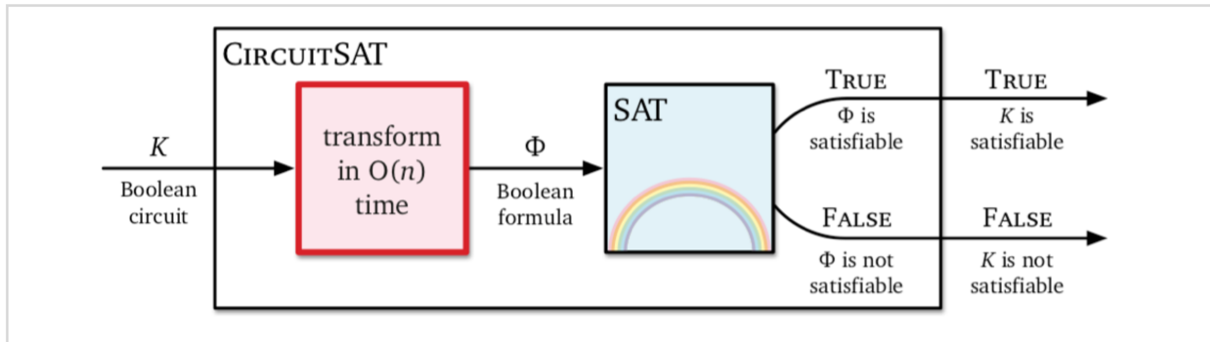
- Remember, there are *a lot* of NP-hard problems.

- Fortunately, showing other problems to be NP hard does not require nearly as much work or having to think about an abstract problem A from NP.
- We use a more direct *reduction argument*.
- Remember, a reduction from problem A to B is an algorithm for A that runs an algorithm for B.
- To prove that problem B is NP-hard, we reduce a **known** *NP-hard* problem A **to** problem B in polynomial time.
- The direction here is essential. You show how to solve the **known hard problem** using the **new problem** as a subroutine. In other words, you end up showing the new problem must be at least as difficult as the old problem, because it can be used to solve the old problem.
- THE OTHER DIRECTION DOES NOT WORK, and performing the reduction backwards is a common mistake, even for those taking the QE exam.
- Let's try our first example. Consider *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 a list of equations separated by ^s describing each gate, and put the final output at the end of the list, because 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.
- We can summarize this process using a diagram: suppose we have a magic polynomial time algorithm for SAT and we want to build an algorithm for CircuitSAT. We can take an

input to CircuitSAT, reduce it to SAT in linear time, and then output the answer given by the magical SAT algorithm.



(the SAT box has a rainbow, because it's magic!)

- The total running time of the CircuitSAT algorithm is O(n) + however long it takes the magical SAT algorithm to run on an O(n) size input.
- If we have a polynomial time algorithm for SAT, then we have a polynomial time algorithm for CircuitSAT. But that means we have a polynomial time algorithm for every other problem in NP.
- In other words, we can solve *every* problem in NP by first reducing to CircuitSAT and that reducing to SAT. SAT is NP-hard.
- Finally, we can verify a True answer to SAT by just evaluating an assignment of the variables in linear time, so SAT is in NP.
- SAT is NP-hard and in NP, so SAT is NP-complete.

3SAT

- Let's look at another NP-complete problem. This one is a special case of SAT called 3SAT or sometimes 3CNF-SAT.
- 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.

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

- 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 heavily 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.

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

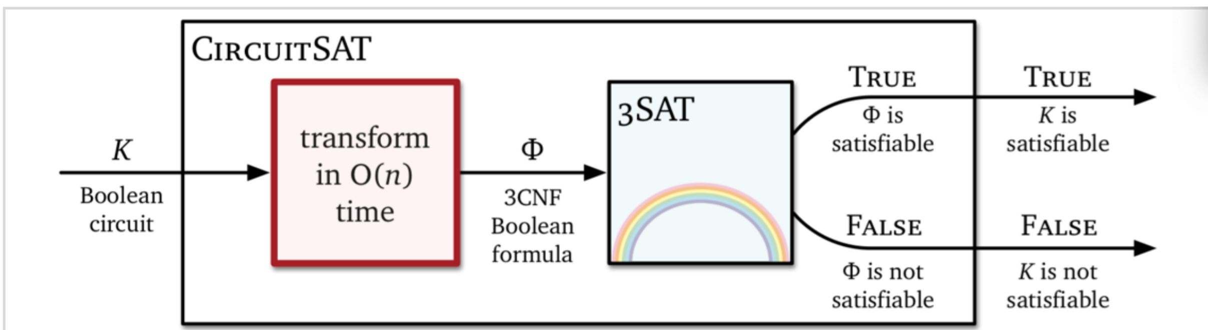
  4. Make sure every clause has exactly three literals by introducing new literals for every one and two-literal clause and expanding them into new clauses.

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

- Here's the 3CNF formula you get from our favorite example 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 every problem in NP. 3SAT is NP-hard.

- 3SAT is a special case of SAT, so it must be in NP also. Together with its hardness, we see 3SAT is NP-complete.
- Next week, we'll see some NP-hard problems that aren't based on booleans!