# CS 6363.003.21S Lecture 22—April 27, 2021

Main topics for #lecture include #NP-hardness.

## 3SAT

- Last time, we discussed the complexity classes P and NP and how problems that are NP-hard problem probably have no polynomial time solution. Today, we'll look at a few more NP-hard problems.
- We'll start with a special case of SAT called 3SAT or sometimes 3CNF-SAT. 3SAT is a good example of how problems can remain NP-hard even if you put a bunch of restrictions on the input. In turn, these restrictions make it a useful problem from which to do further reductions.
- 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.

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

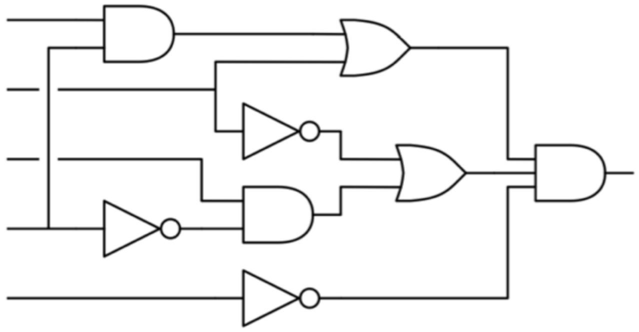
- A 3CNF formula is a CNF formula with exactly three literals per clause. So this example is not a 3CNF formula since the first and last clauses have the wrong number of literals.
- 3SAT: Given a 3CNF formula, is there an assignment of the variables that makes the formula evaluate to True?
- 3SAT looks 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.

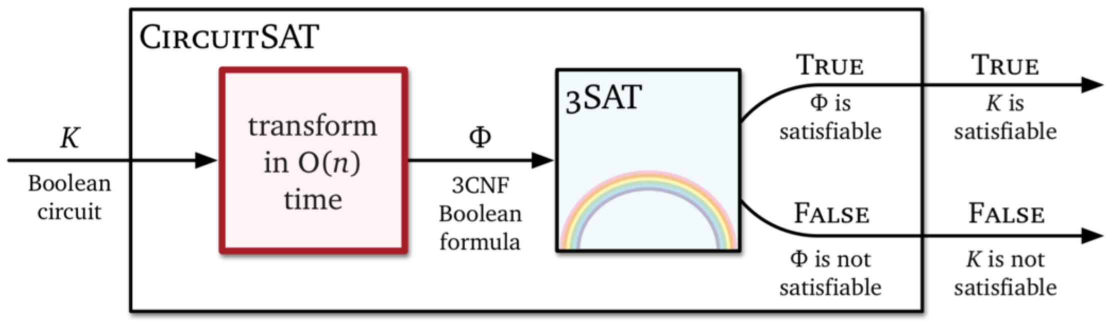
$$\begin{aligned} a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\ a &\longmapsto (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \end{aligned}$$

- 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}$$

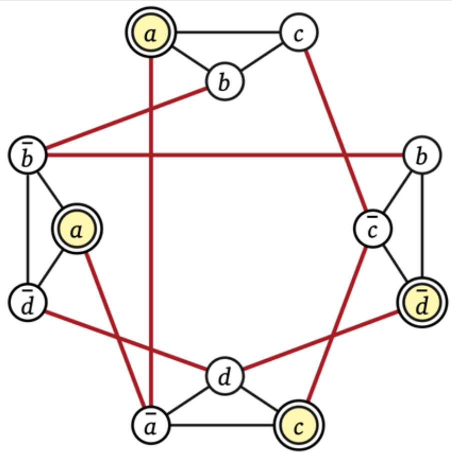
- Yeah, that's gross, but it's only a constant factor larger than the original circuit, and you can compute it in polynomial time.
- In summary, here is what our reduction looked like:




- 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.

## Maximum Independent Set

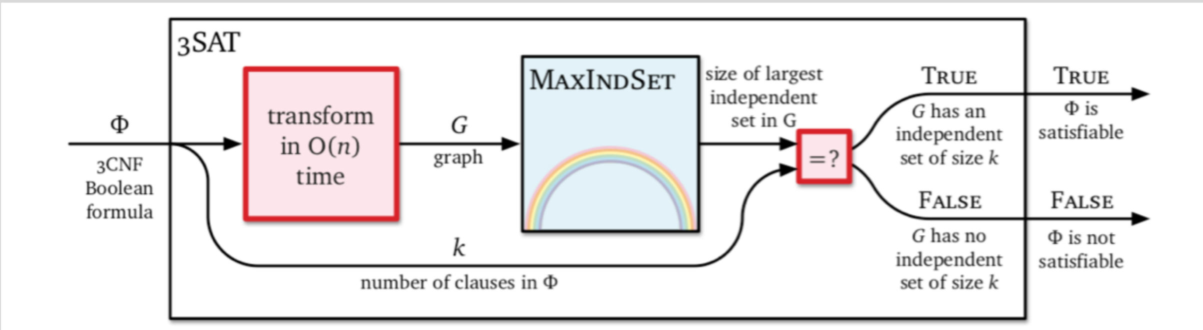
- 3SAT serves as a great jumping off point for a variety of problems that simply based on booleans.
- Suppose we're given a simple, unweighted graph G.
- An *independent set* in G is a subset of vertices with no edges between them.
- The *maximum independent set* problem (MaxIndSet) asks for the largest independent set in the graph. A couple months ago, we saw a linear time algorithm for when G is a tree. But what happens when G is allowed to be any graph?
- Claim: MaxIndSet is NP-hard.
- We'll do a reduction *from* 3SAT.
- Suppose we're given a 3CNF formula Phi. Let k be the number of clauses in Phi.
- We'll make a graph G with 3k vertices, one for each literal in Phi.
- Any two literals in the same clause get a "triangle" edge. Also, any two literals representing a variable and its inverse get a "negation" edge.
- For example, here's the graph you get from the formula below it.




$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

- I claim G contains an independent set of size exactly k if and only if Phi is satisfiable.
  - If Phi is satisfiable, fix a satisfying assignment. Each clause contains at least one true literal, so arbitrarily choose one per clause to make vertex set S. That's one choice per clause so no triangle edge has both sides chosen. And we only chose True literals, so no negation edge has both sides chosen. So S is an independent set of size k.
  - Suppose there is an independent set S of size k. Make each chosen literal True and assign arbitrary values to variables that weren't represented by S. Set S contains at most one vertex per each of the k clause triangles, so we must choose exactly one literal per clause. And we never set two contradictory literals to True because of the negation edges.
- The transformation itself takes O(n) time, so it is a polynomial time reduction.
- Here's our overall algorithm: Do the transformation, and return True if and only if the

maximum independent set has size k.




- So to solve any problem in NP, we can reduce to CircuitSAT and then reduce to 3SAT and then reduce to MaxIndSet, so MaxIndSet is NP-hard. In other words, a polynomial time algorithm for MaxIndSet implies P = NP, so there probably isn't one!
- We can also consider a *decision version* of this problem. Here, we're given the unweighted graph G along with an integer k. We want to *decide* whether or not G has an independent set of size at least k.
- The exact reduction given above implies the decision version is NP-hard as well. The decision version is also in NP, because True answers can be proven by just listing the k vertices in the independent set. So the decision version is actually NP-complete.

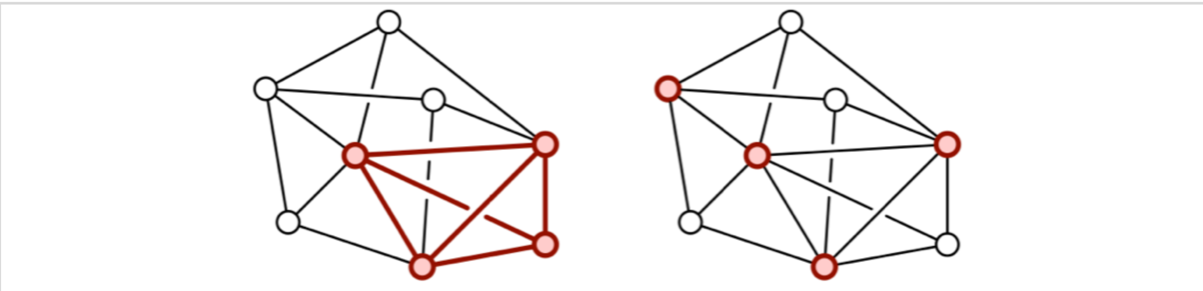
## The General Pattern

- We've now seen a couple examples, so let's talk about the general pattern behind most NP-hardness proofs.
- To prove problem B is NP-hard, we reduce NP-hard problem A to the new problem B. We usually need to do three things:
  1. Describe a polynomial time algorithm to transform an *arbitrary* instance a of A to a *special* instance b of B.
  2. Prove that if a is a "good" instance of A, then b is a "good" instance of B.
  3. Prove that if b is a "good" instance of B, then a is a "good" instance of A.
- You have to show both directions for the proof!
- And you normally think about all three things at once so they're all coherent.
- To address one point of possible confusion: We only have to do the reduction itself in one direction, from A to B. But the correctness proof itself goes in both directions, between the *instances* a and b.
- It may help to think about writing these proofs as algorithms themselves. Being in NP means you have some kind of *certificate* proving the answer is Yes or True. e.g., what inputs to the circuits, what settings of the variables, which vertices to include in a large independent set.
- Step 1 above is an algorithm to transform instances of A to instances of B in polynomial time.

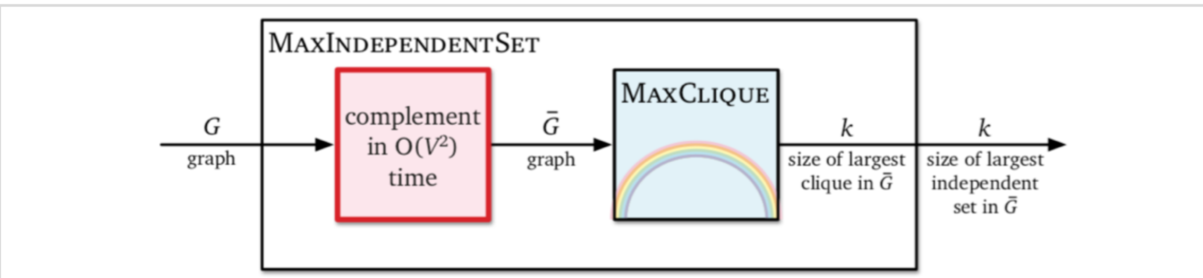
- Step 2 is transforming an arbitrary certificate for a to a certificate for b.
- Step 3 is transforming an arbitrary certificate for b to a certificate for a.
- For example, in 3SAT to MaxIndSet:
    1. We transformed an arbitrary 3CNF formula into a graph and a specific integer k in polynomial time.
    2. We transformed a satisfying assignment into an independent set of size k.
    3. We transformed an independent set of size k into a satisfying assignment.

## Clique and Vertex Cover

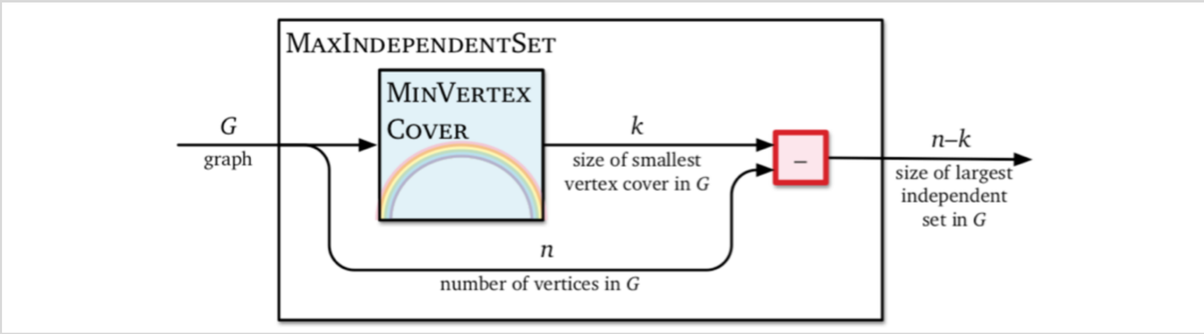
- Let's define a couple more problems. A *clique* is another name for a complete graph. The *MaxClique* problem asks for the number of vertices in the largest complete subgraph of G.
- A *vertex cover* is a set of vertices that touch every edge in the graph. *MinVertexCover* asks for the size of the smallest vertex cover in the graph.
- Below, we have a clique to the left and an vertex cover to the right.



- Claim: MaxClique and MinVertexCover are both NP-hard.
- For MaxClique, we define the edge-complement -G of G as the graph with the same vertices but the opposite set of edges so uv is an edge in -G if and only if it wasn't an edge in G.
- A set of vertices is independent in G if and only if it is a clique in -G, so we can solve MaxIndSet by solving MaxClique in the complement!




- For MinVertexCover, observe that I is an independent set in G = (V, E) if and only if V \ I is a vertex cover. So the largest independent set in G is the complement of the smallest vertex cover. If the smallest vertex cover has size k, the largest independent set has size n - k.




- Like before, the decisions versions of these problems are also hard. Given G and an integer k, the problems of deciding if there is a clique of size k and if there is a vertex cover of size k are both NP-complete.