Main topics for the lecture include NP-hardness.

3SAT

- Last time, we discussed the complexity classes P and NP and how some problem that are NP-hard problem have no polynomial time solution. Today, we'll look at a few more NP-hard problems, many of which have nothing to do with booleans.
- The first 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.

\[
\begin{align*}
\text{clause} & : (a \lor b \lor c \lor d) \land (b \lor \tilde{c} \lor \tilde{d}) \land (\tilde{a} \lor c \lor d) \land (a \lor \tilde{b})
\end{align*}
\]

- 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 boolean satisfiability 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.

\[
\begin{align*}
a = b \land c & \quad \rightarrow \quad (a \lor \tilde{b} \lor \tilde{c}) \land (\tilde{a} \lor b) \land (\tilde{a} \lor c) \\
a = b \lor c & \quad \rightarrow \quad (\tilde{a} \lor b \lor c) \land (a \lor \tilde{b}) \land (a \lor \tilde{c}) \\
a = \tilde{b} & \quad \rightarrow \quad (a \lor b) \land (\tilde{a} \lor \tilde{b})
\end{align*}
\]

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.
Here is the example circuit we looked at yesterday and the 3CNF formula you get from it.

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:

\[
\begin{align*}
& (y_1 \lor \bar{x}_1 \lor \bar{x}_4) \land \neg (y_1 \lor x_1 \lor z_1) \land (\neg y_1 \lor x_1 \lor \bar{z}_1) \land (y_1 \lor x_4 \lor z_2) \land (\neg y_1 \lor x_4 \lor \bar{z}_2) \\
& \land (y_2 \lor x_4 \lor z_3) \land (y_2 \lor x_4 \lor \bar{z}_3) \land (\neg y_2 \lor \bar{x}_4 \lor z_4) \land (\neg y_2 \lor \bar{x}_4 \lor \bar{z}_4) \\
& \land (y_3 \lor x_3 \lor y_2) \land (\neg y_3 \lor x_3 \lor \bar{z}_5) \land (y_5 \lor x_3 \lor \bar{z}_5) \land (\neg y_5 \lor y_2 \lor z_6) \land (\neg y_5 \lor y_2 \lor \bar{z}_6) \\
& \land (y_4 \lor y_1 \lor x_2) \land (y_4 \lor \bar{x}_2 \lor z_7) \land (y_4 \lor \bar{x}_2 \lor \bar{z}_7) \land (y_4 \lor y_1 \lor z_8) \land (y_4 \lor \bar{y}_1 \lor \bar{z}_8) \\
& \land (y_5 \lor x_2 \lor z_9) \land (y_5 \lor x_2 \lor \bar{z}_9) \land (\neg y_5 \lor x_2 \lor z_{10}) \land (\neg y_5 \lor x_2 \lor \bar{z}_{10}) \\
& \land (y_6 \lor x_5 \lor z_{11}) \land (y_6 \lor x_5 \lor \bar{z}_{11}) \land (y_6 \lor \bar{x}_5 \lor z_{12}) \land (y_6 \lor \bar{x}_5 \lor \bar{z}_{12}) \\
& \land (\neg y_7 \lor y_3 \lor \bar{z}_5) \land (\neg y_7 \lor y_3 \lor z_{13}) \land (y_7 \lor \bar{y}_3 \lor \bar{z}_{13}) \land (y_7 \lor \bar{y}_3 \lor z_{14}) \land (y_7 \lor y_5 \lor \bar{z}_{14}) \\
& \land (y_8 \lor \bar{y}_4 \lor \bar{y}_7) \land (\neg y_8 \lor y_4 \lor \bar{y}_{15}) \land (\neg y_8 \lor y_4 \lor z_{15}) \land (y_8 \lor \bar{y}_7 \lor \bar{z}_{16}) \land (y_8 \lor \bar{y}_7 \lor z_{16}) \\
& \land (y_9 \lor \bar{y}_8 \lor \bar{y}_9) \land (\neg y_9 \lor y_8 \lor \bar{z}_{17}) \land (\neg y_9 \lor y_6 \lor \bar{z}_{18}) \land (y_9 \lor y_8 \lor \bar{z}_{17}) \land (\neg y_9 \lor \bar{y}_8 \lor z_{17}) \land (\neg y_9 \lor \bar{y}_8 \lor \bar{z}_{17}) \\
& \land (\neg y_9 \lor z_{19} \lor \bar{z}_{20}) \land (\neg y_9 \lor \bar{z}_{19} \lor \bar{z}_{20}) \land (y_9 \lor z_{19} \lor \bar{z}_{20}) \land (y_9 \lor \bar{z}_{19} \lor \bar{z}_{20})
\end{align*}
\]

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.

It is also in NP, so 3SAT is NP-complete.
Maximum Independent Set

- But not every NP-hard problem is based on circuits.
- 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.
- 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, \((a \lor b \lor c) \land (b \lor \neg c \lor d) \land (a \lor \neg c \lor d) \land (a \lor \neg b \lor d)\) becomes

![Diagram](image)

- 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. There’s one 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. Any independent set contains at most one vertex per clause triangle, so we chose one true literal per clause. And there are no contradictions, 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!

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.
Graph Coloring

- Let's look at yet another graph problem that’s a bit different.
- A proper k-coloring of $G = (V, E)$ is a function $C : V \rightarrow \{1, 2, \ldots, k\}$ assigning one of k “colors” to each vertex so each edge has distinct colors at its endpoints.
- The graph coloring problem is to find the smallest possible number of colors to get a proper k-coloring.
- It’s directly used for certain applications like compiler design. Can I store all my local variables for this function using only a few registers?
- 3Color is the “easier” problem where we simply ask, given a graph, does it have a 3-coloring?
- Claim: 3Color is NP-complete.
- It’s in NP, because you can just tell me the colors and I can verify its a proper coloring in polynomial time.
- We’ll do a reduction from 3SAT. If all else fails, we can maybe use 3SAT.
- Suppose we’re given a 3CNF formula $\Phi$. In many reductions, we build an input to the new problem by combining together a collection of useful subgraphs called gadgets.

There are three types for this reduction:

- A truth gadget: A triangle with vertices T, F, and X standing for True, False, and Other. These three vertices have to have different colors in a proper 3-coloring. For convenience, we’ll refer to the colors they get as True, False, and Other respectively.
- For each variable $a$, a variable gadget: a triangle joining two nodes $a$ and $\neg a$ to the node $X$ used in the truth gadget. Node $a$ must be colored True or False in proper 3-coloring, implying $\neg a$ gets False or True, respectively.

- For each clause in $\Phi$, a clause gadget: We join the three literal nodes for the clause to node T using five new nodes and ten new edges. If all three literals are colored False, then we’ll have a monochromatic edge in the clause gadget when using three colors. However, if at least one literal is colored True, we can get a proper coloring of
the clause gadget. The proof of this claim is just a tedious examination of all the cases.

- So here’s the whole graph for our earlier formula \((a \lor b \lor c) \land (b \lor \neg c \lor d') \land (a \lor \neg c \lor d) \land (a \lor \neg b \lor d').

- So now I claim the graph is 3-colorable if and only if \(\Phi\) is satisfiable.
  - If the graph is 3-colorable, exactly one of \(a\) or \(\neg a\) is assigned True for each variable. And as I already argued, at least one literal per clause is assigned true so we can use the True literals to satisfy \(\Phi\).
  - If \(\Phi\) is satisfiable, then we can use the literal assignments to get their colors and we can assign colors to the rest of the clause gadget vertices to properly color the whole graph.
- Here’s the whole algorithm:

- 3Color is NP-hard since we got a reduction, so it’s NP-complete.
- And the more general optimization version of graph coloring “how many colors do I need” must be NP-hard as well, since it naturally solves 3Color.
And More!

- There are a lot of NP-hard problems.
- Here’s a handful of examples:
  - Hamiltonian cycle: Given a graph, is there a cycle that contains every vertex?
  - Subset sum: Given a set of integers and a target integer T, is there a subset of the integers that sum to T?
  - Set Cover: Given a collection of sets, what is the smallest subcollection whose union is equal to the union of all of the sets?
  - Longest path: Given a graph, what is the longest path in the graph that doesn’t repeat any vertices?
  - Even some games: Sudoku: Given a partly filled out n x n Sudoku puzzle, can I fill in the rest of the numbers to find a solution?
- Sometimes you can still solve these problems quickly for the kinds of inputs you’ll see in practice.
- Sometimes, though, you’ll need to settle for suboptimal solutions, but maybe you can claim they’re still pretty close to being the best solution.