CS 4349 Lecture—November 15, 2017

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

Prelude

- Homework 10 due.
- Homework 11 will be assigned the Monday after Fall Break.

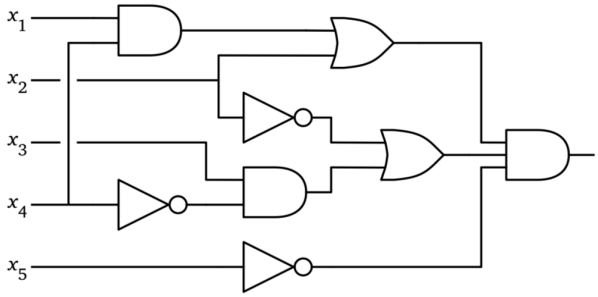
3SAT

- Monday we saw two NP-hard problems that were pretty similar, CircuitSAT and SAT.
- In the first, you're given a description of a boolean circuit. In the second you're given a boolean formula. The goal for both problems is to determine if you can assign the input variables so that the circuit or the formula output or evaluate to true.
- It was probably not that surprising then that we could reduce CircuitSAT to SAT by just writing out the boolean formula that describes the circuit.
- Today, we're going to see that there are a lot of other NP-hard problems, some of which look very different from these SAT problems.
- First, though, we'll show that even a very special case of SAT is NP-hard. This special case is 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 *to* your new problem *from* a known NP-hard 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.

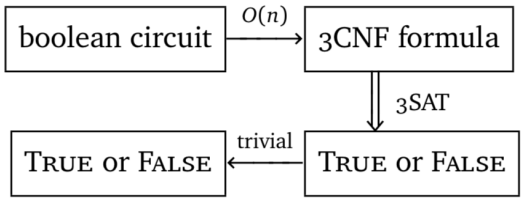
2. Write down the circuit as a formula with one clause per gate. Just like in the reduction to SAT.
3. Change every gate clause into a CNF formula.
    - a = b and c ➔ (a or not(b) or not(c)) and (not(a) or b) and (not(a) or c)
    - a = b or c ➔ (not(a) or b or c) and (a or not(b)) and (a or not(c))
    - a = not(b) ➔ (a or b) and (not(a) or not(b))
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.
    - a ➔ (a or x or y) and (a or not(x) or y) and (a or x or not(y)) and (a or not(x) or not(y))
    - a or b ➔ (a or b or x) and (a or b or not(x))

- Here's that circuit from Monday and the 3CNF formula you get for it.



$$
\begin{aligned}
&(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_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \\
&\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{aligned}
$$

- 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 short, here is what our reduction looked like:

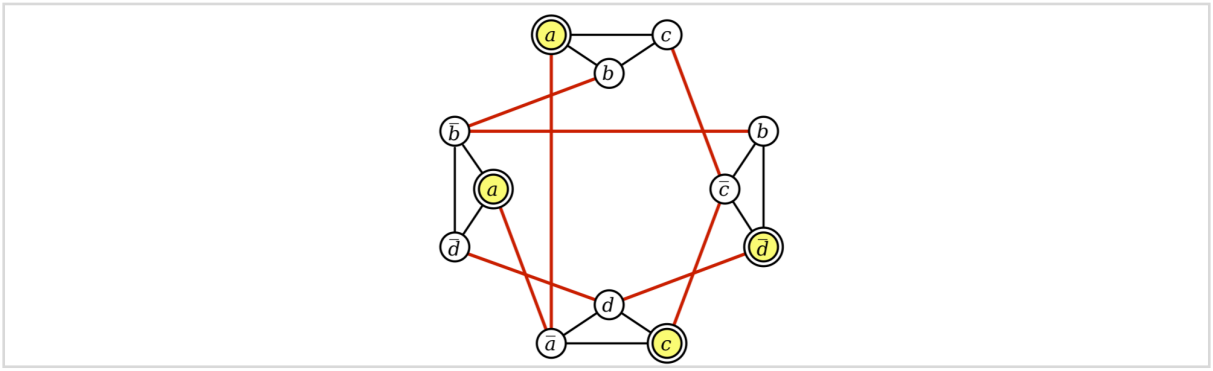


- So a polynomial time algorithm for 3SAT gives a polynomial time algorithm for CircuitSAT and therefore any problem in NP. 3SAT is NP-hard.

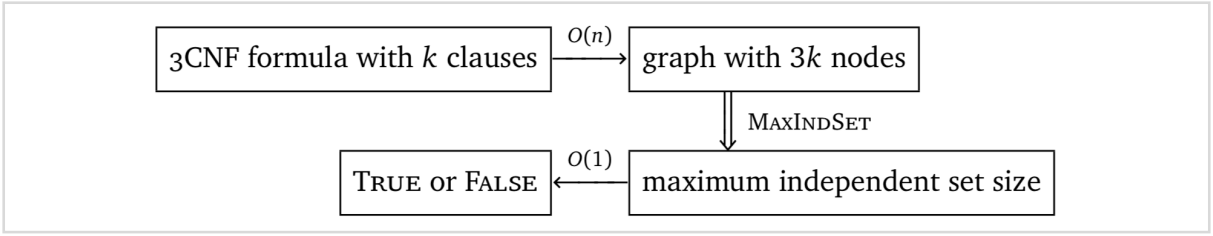
- It is also in NP, so 3SAT is NP-complete.

Maximum Independent Set

- Like I said earlier, lots of very different looking problems are NP-hard.
- Let G be an undirected unweight graph.
- An *independent set* of G is a subset of vertices with no edges between them.
- The *maximum independent set* problem (MaxIndSet) asks for a largest independent set in a given graph.
- MaxIndSet is NP-hard! To show that, we reduce *from* a known NP-hard problem. It turns out 3SAT is very useful for reducing to unexpected settings like graph problems, so we'll use that.
- Given a 3CNF formula with k clauses:
  1. Create a graph G with one node per literal.
  2. Add edges between (a) literals of the same clause and (b) between literals corresponding to a variable and its inverse.
     - So the formula (a or b or c) and (b or not(c) or not(d)) and (not(a) or c or d) and (a or not(b) or not(d)) becomes:



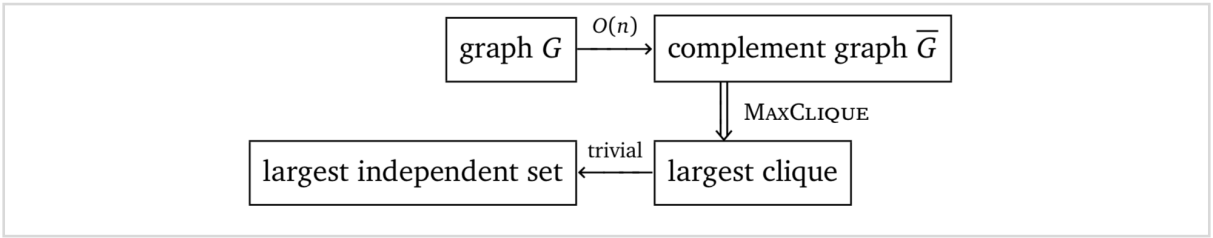
  3. Return True if and only if the maximum independent set has size k.
- We need to prove there is a way to satisfy the formula if and only if the maximum independent set has size k.
  - If the maximum independent set has size k, assign True to each literal in the independent set. Choose arbitrary assignments for variables without literals in the independent set. Contradicting literals are connected by an edge, so we'll get a consistent assignment of truth values. We can pick at most one literal per each of the k clauses, so k chosen literals means we'll be assigning True to some literal in each clause. The formula can be satisfied.
  - If there is a satisfying assignment, choose one literal per clause that is True. We get k choices, and they form an independent set.



- The reduction takes polynomial time, so a poly time algorithm for MaxIndSet gives a poly time algorithm for 3SAT and every problem in NP. MaxIndSet is NP-hard.
- One thing to note, maximum independent set as I defined it is *not* a decision problem since you return a set of vertices instead of a boolean. So it does not belong to NP and is not NP-complete.

Maximum Clique

- A *clique* is another name for a complete graph.
- The *maximum clique problem* (MaxClique) asks for the largest complete subgraph in a given undirected graph.
- One way to think about it is if Facebook published a graph where friends are connected by edges. The maximum clique would be the largest group of mutual friends.
- We can prove MaxClique is NP-hard by reducing *from* MaxIndSet.
- The *edge-complement* bar(G) of a graph G has the opposite set of edges: uv is an edge in bar(G) if and only if uv is *not* an edge in G.
- A set of vertices in G is independent if and only if they form a clique in bar(G).
- So, given a graph G for which we want the maximum independent set, we compute its complement bar(G) in O(V^2) time and compute a maximum clique in the complement.

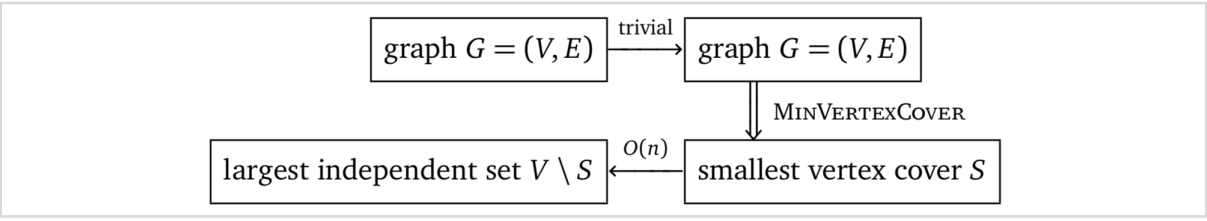


- The reduction takes polynomial time, so a poly time algorithm for MaxClique gives a poly time algorithm for MaxIndSet and every problem in NP. MaxClique is NP-hard.

Vertex Cover

- A *vertex cover* of a graph is a set of vertices that touches every edge in the graph.
- The MinVertexCover problem is to find the smallest vertex cover in a given graph.
- We can prove MinVertexCover is NP-hard by reducing *from* MaxIndSet.
- The reduction relies on a simple fact: If I is an independent set in a graph G = (V, E), then V \ I is a vertex cover.
    - One of the two vertices on each edge is outside the independent set.

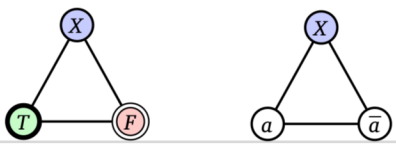
- So, given a graph in which we want to compute MaxIndSet, we just find the smallest vertex cover and return the vertices outside of it.



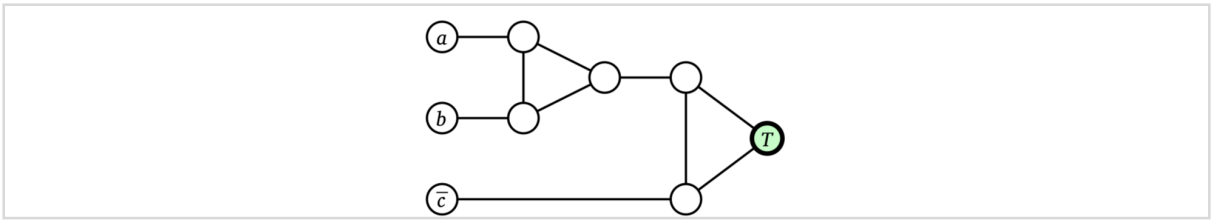
- Again, the reduction takes polynomial time, so a poly time algorithm for MinVertexCover gives a poly time algorithm for MaxIndSet and every problem in NP. MinVertexCover is NP-hard.

Graph Coloring

- A *k-coloring* of a graph is a map C : V ➡ {1, 2, …, k} that assigns one of k 'colors' to each vertex so that every edge has two different colors at its endpoints.
- The graph coloring problem is to find the smallest number of colors needed to legally color a graph.
- It turns out the special case of 3Colorable is already NP-hard: Given a graph, does it have a 3-coloring?
- We can prove it NP-hard by reducing *from* 3SAT. Like in a lot of NP-hardness proofs for more interesting problems, we'll create a bunch of *gadgets* that capture what needs to happen when you satisfy a boolean formula.
- Given a 3CNF formula with k clauses, we create a graph G.:
  1. G contains a single *truth gadget*: a triangle with vertices labeled T, F, and X. They all get different colors in a valid 3-coloring, so we may as well refer to those vertices' colors as True, False, and Other.
  2. G contains one *variable gadget* per variable a, which is a triangle connecting nodes labeled a, not(a), and X. We'll make sure X gets the color Other so that exactly one of a or not(a) is assigned True.

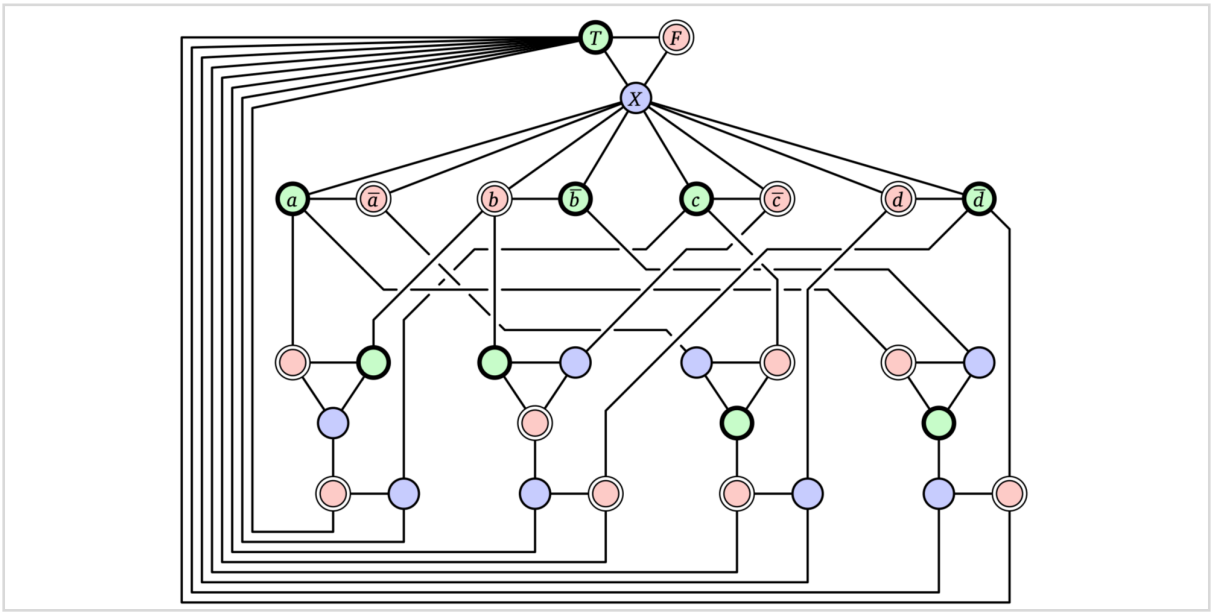


  3. G contains one *clause gadget* per clause drawn like so. You can do a case analysis to verify you cannot color each of the literal vertices False and still color that T vertex True. In particular, if you guarantee to color the literal vertices True or False, then one of them will have to be colored True in a legal 3-coloring.

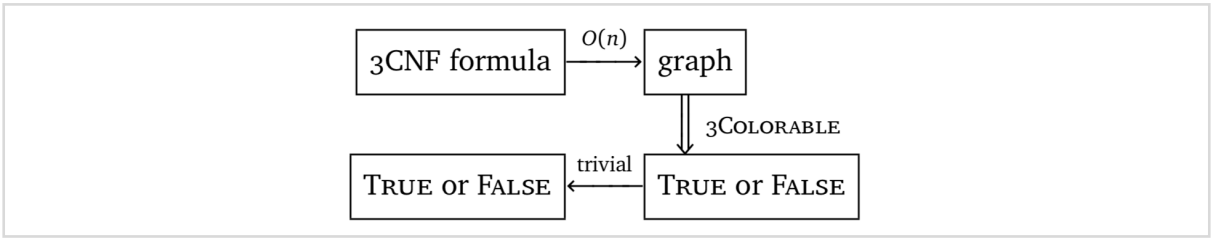


4. Finally, we identify all the T, F, X, and literals nodes with the same label and return whether or not G has a 3-coloring.

- So the formula (a or b or c) and (b or not(c) or not(d)) and (not(a) or c or d) and (a or not(b) or not(d)) becomes:



- To prove the reduction correct, we do the same thing we've done before. If there's a way to satisfy the formula, we can use the variable assignments to choose colors for our vertices. If there's a legal 3-coloring, we can use the colors to choose assignments for our variables.



- Again, the reduction takes polynomial time, so a poly time algorithm for 3Colorable gives a poly time algorithm for 3SAT and every problem in NP. 3Colorable is NP-hard.

So Many Problems

- I could keep doing these reductions for a while still, but let's stop.
- There are A LOT of NP-hard problems. Here are some examples that you may run into in practice or that are particularly interesting. I'll also give a problem from which to reduce to prove NP-hardness. Some of these reductions appear in Jeff Erickson's lecture notes. He also have a lot more example problems.
    - HamiltonianCycle: Given a graph, is there a cycle that visits every vertex exactly once

except the first/last one? From MinVertexCover.

- HamiltonianPath: Given a graph, is there a path that visits every vertex exactly once? From HamiltonianCycle.
- Subset Sum: Given a set X of positive integers and an integer t, does X have a subset summing to t? From MinVertexCover.
    - This problem is called *weakly NP-hard* because there are polynomial time algorithms if the numbers are written in unary. These algorithms take exponential time if the numbers are written in binary, though. The reduction produces exponentially large numbers.
- NotAllEqual3SAT: Given a 3CNF formula, is there an assignment to the variables so every clause contains at least one True *and* at least one False literal? From 3SAT.
- Partition: Given a set S of integers, can we partition them into two disjoint subsets A and B so the sums in a and b are equal? From SubsetSum.
- SetCover: Given a collection of sets S_1, S_2, …, S_m, find the smallest sub-collection containing (covering) every element in the union of the sets. From VertexCover.
- LongestPath: Given a graph G with non-negative edge weights and two vertices u and v, what is the length of the longest path from u to v visiting each vertex *at most* once? From HamiltonianPath.
- SteinerTree: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum weight subtree containing every *marked* vertex. From VertexCover.
    - If *every* vertex is marked, this becomes the minimum spanning tree problem, which we saw has several polynomial time algorithms.
- Generalizations of many games and puzzles, like Tetris, Sudoku, and Super Mario Brothers.