Computational Complexity Theory **Summer HSSP 2019**

Week 3: NP-Completeness

Dylan Hendrickson *MIT Educational Studies Program*

3.1 Definitions and General Facts

Last time, we defined several decision problems. Recall:

- A *circuit* has AND, OR, and NOT gates connected with wires, and computes a boolean value (True or False) from the input wires.
- A *boolean formula* is an expression of boolean variables connected with AND, OR, and NOT. A boolean formula *k-cnf* if it is an AND of *clauses*, each of which is an OR of at most k *literals*, which are variables or NOTs of variables.
- A *graph* has *vertices* connected in pairs by *edges*.

We introduced the following decision problems about these objects:

CircuitEval[uation]: given a circuit and input to the circuit, is the output True?
CircuitSat[isfiability]: given a circuit, is there an input such that the output is True?
FormulaEval: given a boolean formula and assignment, does the assignment satisfy the formula?
FormulaSat: given a boolean formula, is it satisfiable?
2SAT: given a 2-cnf boolean formula, is it satisfiable?
3SAT: given a 3-cnf boolean formula, is it satisfiable?
Path: given a graph, is there a path (along edges) from a specified vertex to another specified vertex?
HamPath: given a graph, is there a path from one vertex to another which visits each vertex exactly once?
2Color: given a graph, can you color the vertices with 2 colors with no same-color vertices sharing an edge?
3Color: given a graph, can you color the vertices with 3 colors with no same-color vertices sharing an edge?

Here are a couple more:
VertexCover: given a graph and a number k, is there a *vertex cover* of size k; i.e. k vertices such that every edge touches at least one of them?
Clique: given a graph and a number k, is there a *clique* of size k; i.e. k vertices with an edge between every pair?

We capture the notion of a "fast" algorithm by saying an algorithm takes *polynomial time* if, for some fixed c, it takes at most $O(n^c)$ on instances of size n, meaning at most n gates/variables/edges/etc.

P is the complexity class of decision problems which can be solved in polynomial time.

NP is the class of problems which can be *verified* in polynomial time. That means that there's a polynomial time algorithm such that when the answer is 'yes,' there's a *certificate* which makes the algorithm accept, and when the answer is 'no,' there isn't. **coNP** is the class of problems which can be verified false in polynomial time. Here we have anti-certificates, which convince a polynomial-time algorithm that the answer is 'no.'

Another way to think about **NP**: *nondeterministic* algorithms are allowed to make guesses, and are assumed to be perfectly lucky. If there's any sequence of guess which leads to accepting, the algorithm accepts. This

is where the term NP, which stands for 'Nondeterministic Polynomial time,' comes from. This is equivalent to the above definition: a nondeterministic algorithm can guess the certificate, and accept if there's any valid certificate. Going the other way, we can use the sequence of guesses which leads to accepting as a certificate. Similarly, **coNP** can be thought of in terms of *co-nondeterministic* algorithms which can make guesses and are perfectly *unlucky*.

Basic facts about the above definitions:

Lemma 3.1. *1. $\boldsymbol{P} \subset \boldsymbol{NP}$. (That is, if $A \in \boldsymbol{P}$, then $A \in \boldsymbol{NP}$.)*

2. $\boldsymbol{P} \subset \boldsymbol{coNP}$.

3. $A \in \boldsymbol{P}$ if and only if $\overline{A} \in \boldsymbol{P}$.

4. $A \in \boldsymbol{NP}$ if and only if $\overline{A} \in \boldsymbol{coNP}$.

Exercise 1. Prove Lemma 3.1. This should mostly consist of understanding the relevant definitions and notation.

Lemma 3.2. *All of the decision problems defined above are in $\boldsymbol{NP}$.*

CircuitEval, FormulaEval, *and* Path *are in $\boldsymbol{P}$.*

Exercise 2. Prove Lemma 3.2. For example, for 3Color we can use a valid 3-coloring as a certificate, and this can be checked in polynomial time by testing for each edge whether the two vertices on it are colored differently.

Lemma 3.3. 2SAT *and* 2Color *are in $\boldsymbol{P}$.*

Exercise 3. Prove Lemma 3.3. These are harder, and require some cleverness in finding a fast algorithm. This illustrates that sometimes there are fast algorithms for problems even if the most obvious algorithms are slower.

Next, we defined *reductions* between problems. Specifically, a *polynomial-time reduction* between decision problems A and B is an algorithm which converts an instance of A to an instance of B with the same answer. Think of this as saying "A is at least as easy as B" or equivalently "B is at least as hard as A."

A decision problem B is ***NP*-hard** if for every $A \in \mathbf{NP}$, there's a polynomial-time reduction from A to B; that is, B is at least as hard as every problem in **NP**. A decision problem is ***NP*-complete** if it's both **NP**-hard and in **NP**; that is, it's (one of) the hardest problem in **NP**.

Lemma 3.4. *If there are polynomial-time reductions $A \to B$ and $B \to C$, then there's a polynomial-time reduction $A \to C$.*

Lemma 3.5. *Suppose there's a polynomial-time reduction $A \to B$. Then*

1. If $B \in \boldsymbol{P}$, then $A \in \boldsymbol{P}$.

2. If $B \in \boldsymbol{NP}$, then $A \in \boldsymbol{NP}$.

3. If A is $\boldsymbol{NP}$-hard, then B is $\boldsymbol{NP}$-hard.

Exercise 4. Prove Lemmas 3.4 and 3.5.

Exercise 5. Show that if a problem is both **NP**-hard and in **P**, then $\mathbf{P} = \mathbf{NP}$.

Exercise 6. Suppose A and B are both in **P**. When is there a polynomial-time reduction $A \to B$?

Nobody knows for sure whether $\mathbf{P} = \mathbf{NP}$, and if you figure it out you can win a lot of fame and money. However, we're pretty confident $\mathbf{P} \neq \mathbf{NP}$. Hopefully this class will show you where that impression comes from. We'll be showing problems **NP**-complete; the above exercise says that if $\mathbf{P} = \mathbf{NP}$, such a problem is not in **P**. This is about is close as we can get to showing a problem in **NP** isn't in **P**.

3.2 Proving Problems NP-Hard

With all that out of the way, we'll show some problems are **NP**-hard. We start by sketching a proof of the following theorem; going through all the details is a pain and we'll skip it.

Theorem 3.6 (Cook-Levin)**.** CircuitSat *is **NP**-complete.*

Proof. We already know CircuitSat $\in$ **NP**, since we can use inputs to the circuit as certificates. The hard part is showing CircuitSat is **NP**-hard. To do this, let A be an arbitrary decision problem in **NP**, and we will show that there's a polynomial-time reduction $A \to$ CircuitSat.

Since $A \in$ **NP**, there's a polynomial-time algorithm f which validates certificates for A. That is, f takes as input an instance x and a certificate c, and accepts if c is a valid certificate for x. In addition, there is a valid certificate exactly when $x \in A$.

By expressing an algorithm's behavior extremely precisely, we can expand it into a sequence of logic operations—say AND, OR, and NOT. This converts an algorithm into a circuit. More accurately, we get a family of circuits, one for each input size, since the input to a circuit has a fixed size. The size of the circuit we get is approximately the same as the running time of the program: a polynomial-time algorithm gives a polynomial-size family of circuits. (Explaining exactly how to do this conversion from algorithm to circuit would require a precise definition of an algorithm.)

Here's our reduction $A \to$ CircuitSat: take an A-instance x of size n. Build the circuit corresponding to f on inputs of size n. This circuit has input wires for the instance x and certificate c, and outputs the same thing as f on the same input. Now we force the input wires for the instance to be the values of the actual instance x, and leave the inputs for the certificate as inputs to the circuit.

The result is a circuit which has an input which makes the output True exactly when there's a certificate which convince f to accept x. □

Now that we know about one **NP**-hard problem, Lemma 3.5.3 lets us prove more problems **NP**-hard more easily. All we need to do is find a reduction CircuitSat $\to B$; then B is **NP**-hard. If B is also in **NP**, then we know it's **NP**-complete.

Let's show 3SAT is **NP**-complete. by giving a reduction CircuitSat $\to$ **3SAT**. Given a circuit, we want to find a 3-cnf formula which is satisfiable if and only if the circuit can output True. Our formula will have a variable for each input wire, and also a variable for each gate in the circuit. We then need a bunch of clauses of size at most 3 which force the gate variables to be the values the circuit computes for them. That means that in order to satisfy the formula, you have to make it simulate the circuit, so it'll be satisfiable exactly when the circuit can output True.

For the actual clauses, it depends on what kind of gate we have:

- If x & $y = z$, use $(x \vee \neg z)$ & $(y \vee \neg z)$ & $(\neg x \vee \neg y \vee z)$.
- If $x \vee y = z$, use $(\neg x \vee z)$ & $(\neg y \vee z)$ & $(x \vee y \vee \neg z)$.
- If $\neg x = y$, use $(x \vee y)$ & $(\neg x \vee \neg y)$.

For the output gate z_O, we also need a clause (z_O) which forces the output of the simulated circuit to be True.

If the circuit is satisfiable, the formula is: we take the assignment which is the value of each input and gate when on an input to the circuit which outputs True. If the formula is satisfiable, the circuit is: take the

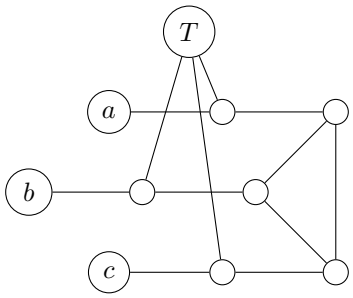
input to the circuit based on the variables corresponding to input wires; then the output of each gate is the same as the corresponding variable in the formula, so the output gate outputs True.

Since we now know 3SAT is **NP**-complete, we can show other problems **NP**-hard by a reduction from 3SAT, which is often easier to find than a reduction from CircuitSat. Let's give a reduction 3SAT $\to$ 3Color to show that 3Color is **NP**-hard.

Observe that 2SAT is in **P**, but 3SAT is **NP**-complete. Increasing the size of clauses suddenly makes the problem much harder.

Given a 3-cnf formula, we construct a graph for 3Color as follows:

- Three vertices in a triangle called T, F, and Z.
- For each variable x, a *vertex gadget*: two vertices x and $\neg x$ connected to each other and to Z.
- For each clause $(a \vee b \vee c)$, a *clause gadget*: 6 vertices connected as in the figure below.



This graph can be constructed in polynomial time. Let's see that it's in fact a reduction. Suppose the formula is satisfiable. Color T green, F red, and Z blue. Color x green and $\neg x$ red if x is true in the satisfying assignment, and vice-versa otherwise. For each clause $(a \vee b \vee c)$, at least one of a, b, and c is green. If a is green, we can color the clause gadget as follows: the vertex immediately right of a is red, and those immediately right of b and c are blue. The vertices two edges right of a, b, and c are blue, red, and green, respectively. This is a valid 3-coloring.

Now suppose the graph has a 3-coloring. Then T, F, and Z are all different colors; suppose they're green, red, and blue, respectively. We think of green as True and red as False. Because of the variable gadgets, for each variable x the vertices x and $\neg x$ are green and red in some order. Take the assignment which sets the literals corresponding to green vertices True. To see that this satisfies the formula, consider a clause gadget. For it to be colored in a valid way, the vertices in the right triangle are all different colors; one of the is blue. The vertex left of that one is connected to T and thus can't be green, so it's red. Then the literal connected to it can't be red, and must be green. That is, at least one literal in each clause is true. Thus this is a satisfying assignment.

Exercise 7. Show that the following problems are **NP**-complete: (listed in rough order of difficulty)

- FormulaSat
- VertexCover
- Clique
- HamPath