Lecture 20

NP-Completeness II

20.1 Overview

In the last lecture, we defined the class **NP** and the notion of **NP**-completeness, and proved that the Circuit-SAT problem is **NP**-complete. In this lecture we continue our discussion of NP-Completeness, showing the following results:

- CIRCUIT-SAT $\leq_p$ 3-SAT (proving 3-SAT is NP-complete)
- 3-SAT $\leq_p$ CLIQUE (proving CLIQUE is NP-complete)
- NP-completeness of Independent Set and Vertex Cover

20.2 Introduction

Let us begin with a quick recap of our discussion in the last lecture. First of all, to be clear in our terminology, a *problem* means something like 3-coloring or network flow, and an *instance* means a specific instance of that problem: the graph to color, or the network and distinguished nodes s and t we want to find the flow between. A *decision problem* is just a problem where each instance is either a YES-instance or a NO-instance, and the goal is to decide which type your given instance is. E.g., for 3-coloring, G is a YES-instance if it has a 3-coloring and is a NO-instance if not. For the Traveling Salesman Problem, an instance consists of a graph G together with an integer k, and the pair (G, k) is a YES-instance iff G has a TSP tour of total length at most k.

We now define our key problem classes of interest.

P: The class of decision problems Q that have polynomial-time algorithms. $Q \in \mathbf{P}$ if there exists a polynomial-time algorithm A such that $A(I) =$ YES iff I is a YES-instance of Q.

NP: The class of decision problems where at least the YES-instances have short proofs (that can be checked in polynomial-time). $Q \in \mathbf{NP}$ if there exists a verifier $V(I, X)$ such that:

- If I is a YES-instance, then there exists X such that $V(I, X) =$ YES,
- If I is a NO-instance, then for all X, $V(I, X) =$ NO,

and furthermore the length of X and the running time of V are polynomial in $|I|$.

co-NP: vice-versa — there are short proofs for NO-instances. Specifically, $Q \in$ **co-NP** if there exists a verifier $V(I, X)$ such that:

- If I is a YES-instance, for all X, $V(I, X) =$ YES,
- If I is a NO-instance, then there exists X such that $V(I, X) =$ NO,

and furthermore the length of X and the running time of V are polynomial in $|I|$.

For example, the problem Circuit-Equivalence: "Given two circuits C_1, C_2, do they compute the same function?" is in **co-NP**, because if the answer is NO, then there is a short, easily verified proof (an input x such that $C_1(x) \neq C_2(x)$).

A problem Q is **NP**-complete if:

1. $Q \in$ **NP**, and
2. Any other Q' in **NP** is polynomial-time reducible to Q; that is, $Q' \leq_p Q$.

If Q just satisfies (2) then it's called **NP**-hard. Last time we showed that the following problem is **NP**-complete:

> **Circuit-SAT**: Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?

Aside: we could define the *search*-version of a problem in **NP** as: "...and furthermore, if I is a YES-instance, then *produce* X such that $V(I, X) =$ YES." If we can solve any **NP**-complete decision problem in polynomial time then we can actually solve search-version of any problem in **NP** in polynomial-time too. The reason is that if we can solve an **NP**-complete problem in polynomial time, then we can solve the ESP in polynomial time, and we already saw how that allows us to produce the X for any given verifier V.

Unfortunately, Circuit-SAT is a little unweildy. What's *especially interesting* about NP-completeness is not just that such problems *exist*, but that a lot of very innocuous-looking problems are NP-complete. To show results of this form, we will first reduce Circuit-SAT to the much simpler-looking 3-SAT problem (i.e., show Circuit-SAT $\leq_p$ 3-SAT). Recall the definition of 3-SAT from last time:

Definition 20.1 3-SAT: *Given: a CNF formula (AND of ORs) over n variables $x_1, \ldots, x_n$, where each clause has at most 3 variables in it. Goal: find an assignment to the variables that satisfies the formula if one exists.*

Theorem 20.1 *CIRCUIT-SAT $\leq_p$ 3-SAT. I.e., if we can solve 3-SAT in polynomial time, then we can solve CIRCUIT-SAT in polynomial time (and thus all of* **NP***).*

Proof: We need to define a function f that converts instances C of Circuit-SAT to instances of 3-SAT such that the formula $f(C)$ produced is satisfiable iff the circuit C had an input x such that $C(x) = 1$. Moreover, $f(C)$ should be computable in polynomial time, which among other things means we cannot blow up the size of C by more than a polynomial factor.

First of all, let's assume our input is given as a list of gates, where for each gate g_i we are told what its inputs are connected to. For example, such a list might look like: $g_1 = \mathsf{NAND}(x_1, x_3)$; $g_2 == \mathsf{NAND}(g_1, x_4)$; $g_3 = \mathsf{NAND}(x_1, 1)$; $g_4 = \mathsf{NAND}(g_1, g_2)$; .... In addition we are told which gate g_m is the output of the circuit.

We will now compile this into an instance of 3-SAT as follows. We will make one variable for each input x_i of the circuit, and one for every gate g_i. We now write each NAND as a conjunction of 4 clauses. In particular, we just replace each statement of the form "$y_3 = \mathsf{NAND}(y_1, y_2)$" with:

$$\begin{array}{lll} & (y_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\ \text{AND} & (y_1 \text{ OR } \overline{y}_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 1 \text{ then we must have } y_3 = 1 \\ \text{AND} & (\overline{y}_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\ \text{AND} & (\overline{y}_1 \text{ OR } \overline{y}_2 \text{ OR } \overline{y}_3). & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 1 \text{ we must have } y_3 = 0 \end{array}$$

Finally, we add the clause (g_m), requiring the circuit to output 1. In other words, we are asking: is there an input to the circuit *and* a setting of all the gates such that the output of the circuit is equal to 1, *and* each gate is doing what it's supposed to? So, the 3-CNF formula produced is satisfiable if and only if the circuit has a setting of inputs that causes it to output 1. The size of the formula is linear in the size of the circuit. Moreover, the construction can be done in polynomial (actually, linear) time. So, if we had a polynomial-time algorithm to solve 3-SAT, then we could solve circuit-SAT in polynomial time too. ■

Important note: Now that we know 3-SAT is **NP**-complete, in order to prove some other **NP** problem Q is **NP**-complete, we just need to reduce 3-SAT to Q; i.e., to show that 3-SAT $\leq_p Q$. In particular, we want to construct a (polynomial-time computable) function f that converts instances of 3-SAT to instances of Q that preserves the YES/NO answer. This means that if we could solve Q efficiently then we could solve 3-SAT efficiently. *Make sure you understand this reasoning — a lot of people make the mistake of doing the reduction the other way around.* Doing the reduction the wrong way is just as much work but does not prove the result you want to prove!

20.3 CLIQUE

We will now use the fact that 3-SAT is **NP**-complete to prove that a natural graph problem called the Max-Clique problem is **NP**-complete.

Definition 20.2 Max-Clique*: Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbors). Decision problem: "Given G and integer k, does G contain a clique of size $\geq k$?"*

Note that Max-Clique is clearly in **NP**.

Theorem 20.2 Max-Clique *is* **NP**-*Complete.*

Proof: We will reduce 3-SAT to Max-Clique. Specifically, given a 3-CNF formula F of m clauses over n variables, we construct a graph as follows. First, for each clause c of F we create one node for every assignment to variables in c that satisfies c. E.g., say we have:

$$F = (x_1 \vee x_2 \vee \overline{x}_4) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge \ldots$$

Then in this case we would create nodes like this:

$$\begin{array}{llll}
(x_1 = 0, x_2 = 0, x_4 = 0) & (x_3 = 0, x_4 = 0) & (x_2 = 0, x_3 = 0) & \dots \\
(x_1 = 0, x_2 = 1, x_4 = 0) & (x_3 = 0, x_4 = 1) & (x_2 = 0, x_3 = 1) & \\
(x_1 = 0, x_2 = 1, x_4 = 1) & (x_3 = 1, x_4 = 1) & (x_2 = 1, x_3 = 0) & \\
(x_1 = 1, x_2 = 0, x_4 = 0) & & & \\
(x_1 = 1, x_2 = 0, x_4 = 1) & & & \\
(x_1 = 1, x_2 = 1, x_4 = 0) & & & \\
(x_1 = 1, x_2 = 1, x_4 = 1) & & &
\end{array}$$

We then put an edge between two nodes if the partial assignments are consistent. Notice that the maximum possible clique size is m because there are no edges between any two nodes that correspond to the same clause c. Moreover, if the 3-SAT problem *does* have a satisfying assignment, then in fact there *is* an m-clique (just pick some satisfying assignment and take the m nodes consistent with that assignment). So, to prove that this reduction (with $k = m$) is correct we need to show that if there *isn't* a satisfying assignment to F then the maximum clique in the graph has size $< m$. We can argue this by looking at the contrapositive. Specifically, if the graph has an m-clique, then this clique must contain one node per clause c. So, just read off the assignment given in the nodes of the clique: this by construction will satisfy all the clauses. So, we have shown this graph has a clique of size m iff F was satisfiable. Also, our reduction is polynomial time since the graph produced has total size at most quadratic in the size of the formula F ($O(m)$ nodes, $O(m^2)$ edges). Therefore Max-Clique is **NP**-complete. ■

20.4 Independent Set and Vertex Cover

An Independent Set in a graph is a set of nodes no two of which have an edge. E.g., in a 7-cycle, the largest independent set has size 3, and in the graph coloring problem, the set of nodes colored red is an independent set. The Independent Set problem is: given a graph G and an integer k, does G have an independent set of size $\geq k$?

Theorem 20.3 Independent Set *is* **NP***-complete.*

Proof: We reduce from Max-Clique. Given an instance (G, k) of the Max-Clique problem, we output the instance (H, k) of the Independent Set problem where H is the complement of G. That is, H has edge (u, v) iff G does *not* have edge (u, v). Then H has an independent set of size k iff G has a k-clique. ■

A *vertex cover* in a graph is a set of nodes such that every edge is incident to at least one of them. For instance, if the graph represents rooms and corridors in a museum, then a vertex cover is a set of rooms we can put security guards in such that every corridor is observed by at least one guard. In this case we want the smallest cover possible. The Vertex Cover problem is: given a graph G and an integer k, does G have a vertex cover of size $\leq k$?

Theorem 20.4 Vertex Cover *is* **NP***-complete.*

Proof: If C is a vertex cover in a graph G with vertex set V, then $V - C$ is an independent set. Also if S is an independent set, then $V - S$ is a vertex cover. So, the reduction from Independent Set to Vertex Cover is very simple: given an instance (G, k) for Independent Set, produce the instance $(G, n - k)$ for Vertex Cover, where $n = |V|$. In other words, to solve the question "is there an independent set of size at least k" just solve the question "is there a vertex cover of size $\leq n - k$?" So, Vertex Cover is **NP**-Complete too. ■

20.5 Beyond NP

As mentioned earlier, it is an open problem whether $\mathbf{P} \neq \mathbf{NP}$ (though everyone believes they are different). It is also an open problem whether $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ (though again, everyone believes they are different). One can also define even more expressive classes. For instance, **PSPACE** is the class of all problems solvable by an algorithm that uses a polynomial amount of memory. Any problem in **NP** is also in **PSPACE**, because one way to solve the problem is to take the given instance I and then simply run the verifier $V(I, X)$ on all possible proof strings X, halting with YES if any of the runs outputs YES, and halting with NO otherwise. (Remember, we have a polynomial upper bound on $|X|$, so this uses only a polynomial amount of space.) Similarly, any problem in **co-NP** is also in **PSPACE**. Unfortunately, it is not even known for certain that $\mathbf{P} \neq \mathbf{PSPACE}$ (though all the classes mentioned above are believed to be different). One can also define classes that are *provably* larger than **P** and **NP**. For instance, **EXPtime** is the class of all problems solvable in time $O(2^{n^c})$ for some constant c. This class is known to *strictly* contain **P**. The class **NEXPtime** is the class of all problems that have a *verifier* which runs in time $O(2^{n^c})$ for some constant c. This class is known to strictly contain **NP**. The class of all Turing-computable problems is known to strictly contain all of the above, and some problems such as the Halting problem (given a program $\mathcal{A}$ and an input x, determine whether or not $\mathcal{A}(x)$ halts) are not even contained in that!

20.6 NP-Completeness summary

NP-complete problems have the dual property that they belong to **NP** and they capture the essence of the entire class in that a polynomial-time algorithm to solve one of them would let you solve anything in **NP**.

We proved that 3-SAT is **NP**-complete by reduction from Circuit-SAT. Given a circuit C, we showed how to compile it into a 3-CNF formula by using extra variables for each gate, such that the formula produced is satisfiable if and only if there exists x such that $C(x) = 1$. This means that a polynomial-time algorithm for 3-SAT could solve any problem in **NP** in polynomial-time, even factoring. Moreover, 3-SAT is a simple-looking enough problem that we can use it to show that many other problems are **NP**-complete as well, including Max-Clique, Independent Set, and Vertex Cover.