15-451/651: Design & Analysis of Algorithms October 27, 2014
Lecture #17 last changed: October 27, 2014

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

1 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 $A(I) = \text{YES}$ 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 $V(I, X) = \text{YES}$,
- If I is a NO-instance, then for all X, $V(I, X) = \text{NO}$,

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

In all our **NP**-hardness reductions (say from problem Q to problem Q'), we will ensure we only "make a single call to the subroutine for problem Q'". This is called a *many-one* or *Karp* reduction:

Many-one reduction (a.k.a. Karp reduction) from problem Q to problem Q': To reduce problem Q to problem Q' we want a function f that takes arbitrary instances of Q to instances of Q' such that:

1. if I is a YES-instance of Q then $f(I)$ is a YES-instance of Q'.
2. if I is a NO-instance of Q then $f(I)$ is a NO-instance of Q'.
3. f can be computed in polynomial time.

So, if we had an algorithm for Q', and a function f with the above properties, we could using it to solve Q on any instance I by running it on $f(I)$.

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.

2 Circuit-SAT and 3-SAT

Remember, a problem Q is **NP**-complete if:

1. $Q \in \mathbf{NP}$, 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 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?

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 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 2 *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.[1]

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:

[1] Using the terminology of the previous lecture, this is a *Karp reduction* (or many-one reduction) from Circuit-SAT to 3-SAT. In other words, an instance C of Circuit-SAT is being mapped to a single instance $f(C)$ of 3-SAT so that C is a YES-instance if and only if $f(C)$ is a YES-instance. You should use Karp reductions for all problems whenever possible.

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

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, and reduce the unknown problem to the ***NP****-complete problem.* Doing the reduction the wrong way is just as much work but does not prove the result you want to prove!

3 CLIQUE

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

Definition 3 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 Clique is clearly in **NP**.

Theorem 4 Clique *is* **NP***-Complete.*

Proof: We will reduce 3-SAT to 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:

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

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. We claim that the maximum size is m if and only if the original formula has a satisfying assignment.

Suppose 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).

For the other direction, we can either show that if there *isn't* a satisfying assignment to F then the maximum clique in the graph has size $< m$, or argue the contrapositive and show that if there is a m-clique in the graph, then there is a satisfying assignment for the formula. 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.

Finally, to complete the proof, we note that 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 Clique is **NP**-complete. ■

3.1 Proving NP-completeness in 2 Easy Steps

If you want to prove that problem Q is NP-complete, you need to do two things:

1. Show that Q is in NP.

2. Choose some NP-hard problem P to reduce from. This problem could be 3-SAT or Clique or Independent Set or Vertex Cover or any of the zillions of NP-hard problems known. Most of the time in this course we will suggest a problem to reduce from (but you can choose another one if you like). In the real world you will have to figure out this problem P yourself.

   Now you want to reduce **from** P **to** Q. In other words, given any instance I of P, show how to transform it into an instance $f(I)$ of Q, such that

   $$I \text{ is a YES-instance of } P \iff f(I) \text{ is a YES-instance of } Q.$$

   Note the " $\iff$ " in the middle—*you need to show both directions.* You also need to show that the mapping $f(\cdot)$ can be done in polynomial time (and hence $f(I)$ has size polynomial in the size of the original instance I).

A common mistake is reducing from the problem Q to the hard problem P. Think about what this means. It means you can model your problem as a hard problem. Just because you can model the problem of adding two numbers as a linear program or as 3-SAT does not make addition complicated. You want to reduce the hard problem P to your problem Q, this shows Q is "at least as hard" as P.

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 5 Independent Set *is* **NP**-*complete.*

Proof: We reduce from Clique. Given an instance (G, k) of the 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 6 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. ■

5 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 Clique, Independent Set, and Vertex Cover.

6 Bonus: A Non-Trivial Proof of Membership in NP

Most of the time the proofs of a problem belonging to **NP** are trivial: you can use the solution as the witness X. Here's a non-trivial example that we alluded to in lecture, a proof that Primes is in **NP**. Recall that Primes is the decision problem: given a number N, is it prime? To show this is in **NP**, we need to show a poly-time verifier algorithm $V(N,X)$ that N is a prime if and only if there exists a short witness X which will make the verifier say YES.

Today we know that Primes is in **P**, so we could just use that algorithm as a verifier. In fact we could use the fact that testing Primality has a randomized algorithm with one-sided error to also show that Primes is in **NP**. But those result are somewhat advanced, can we use something simpler?

Here is a proof due to Vaughan Pratt[2] from 1975 that uses basic number theory. He uses the following theorem of Édouard Lucas[3]:

[2] Computer Science Professor at Stanford. He was one of the inventors of the deterministic linear-time median-finding algorithm, and also the Knuth-Morris-Pratt string matching algorithm. Also designed the logo for Sun Microsystems.

[3] French mathematician (1842-1891), worked on number theory, and on Fibonacci numbers and the Lucas numbers named after him. Apparently invented the Tower of Hanoi puzzle (or at least popularized it). Died when cut by a piece of glass from a broken plate at a banquet.

Theorem 7 *A number p is a prime if and only if there exists some $g \in \{0, 1, \ldots, p-1\}$ such that*

$$g^{p-1} \equiv 1 \textit{ mod } p \qquad \textit{and} \qquad g^{(p-1)/q} \not\equiv 1 \textit{ mod } p \quad \textit{for all primes } q|(p-1).$$

Great. So if N was indeed a prime, the witness could be this number g corresponding to N that Lucas promises. We could check for that g to the appropriate powers was either equivalent to 1 or not. (By repeating squaring we can compute those powers in time $O(\log N)$.)

Hmm, we need to check the condition for all primes q that divide $N-1$. No problem: we can write down the prime factorization of $N-1$ as part of the witness. It can say: $N-1 = q_1 \cdot q_2 \cdot \ldots \cdot q_k$. Note that there are at most $\log_2(N-1)$ many distinct primes in this list (since each of them is at least 2), and each of them takes $O(\log N)$ bits to write down. And what about their primality? This is the clever part: we recursively write down witnesses for their primality. (The base case is 3 or smaller, then we stop.)

How long is this witness? Let's just look at the number of numbers we write down, each number will be $O(\log N)$ bits.

Note we wrote down g, and then k numbers q_i. That's a total of $k+1$ numbers. And then we recurse. Say each q_i required $c(\log_2 q_i) - 2$ numbers to write down a witness of primality. Then we need to ensure that

$$(k+1) + \sum_{i=1}^{k} (c \log_2 q_i - 3) \leq c \log_2 N - 3$$

But $\sum_i^k \log_2 q_i = \log_2(N-1)$. So we get that the LHS is $(k+1) + c\log_2(N-1) - 3k = c\log_2(N-1) - 2k + 1$. And finally, we use the fact that $N-1$ cannot be prime if N is (except for $N=3$), so $k \geq 2$ and thus $c\log_2(N-1) - 2k + 1 \leq c\log_2 N - 3$. Finally, looking at the base case shows that $c \geq 4$ suffices.

To summarize, the witness used at most $O(\log N)$ numbers, each of $O(\log N)$ bits. This completes the proof that PRIMES is in **NP**.

7 Bonus: Co-NP

Just like we defined NP as the class of problems for which there were short proofs for YES-instances, we can define a class of problems for which 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 $C_1(x) \neq C_2(x)$).