CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 25

December 5, 2019

Instructor: Sepehr Assadi

1 Complexity Classes P and NP

We finished the last lecture by defining the complexity classes P and NP as follows:

- **Class P:** The set of all decision problems that admit a poly-time algorithms. In other words, these are problems that can be *solved* efficiently.
- **Class NP:** The set of all decision problems that admit a poly-time verifier. In other words, these are problems that can be *verified* efficiently. Intuitively, NP is the set of decision problems where we can verify a Yes answer (or True or 1) "efficiently" *assuming* we have the solution (or some other form of proofs) in front of us.

We also showed that it is easy to see P $\subseteq$ NP, i.e., any problem that can be solved in polynomial time can definitely be verified in polynomial time also. A highly fundamental question is that whether the inverse of this statement is also true or not meaning that is it the case that any problem that can be verified in polynomial time can also be solved in polynomial time? This problem can be stated concisely as:

Is P = NP?

Many researcher believe that P $\neq$ NP although at this point we are nowhere near answering this question conclusively. One intuitive (and extremely vague) reason for the conjecture that P $\neq$ NP is simply because we all consider solving a problem "harder" than reading a solution to the problem and checking whether it is correct or not[1], namely, verifying a problem. However, a way stronger argument can be made by considering the notion of *NP-hard* and *NP-complete* problems which we now define.

2 NP-Hard and NP-Complete Problems

We have the following two definitions:

- **NP-Hard Problems:** We say that a problem Π is NP-hard if having a polynomial-time algorithm for Π implies that P = NP, namely, by giving a poly-time algorithm for Π, we obtain a poly-time algorithm for *all* problems in NP. Note that NP-hard problems are *not* necessarily decision problems.
- **NP-Complete Problems:** We say that a *decision* problem Π is NP-complete if it is NP-hard and it also belongs to the class NP. Intuitively, NP-complete problems are the "hardest" problems in NP: if we can solve any one of them in polynomial time, we can solve *every* NP problem in polynomial time. Alternatively, we can also think of NP-complete problems as "easiest" problems that are NP-hard.

So why do we care about NP-hard (or NP-complete) problems? The answer is two-fold:

[1]If nothing else, hopefully the homeworks in this course have convinced you of this!

- If you believe (or suspect, or think, or guess, or whatever) P = NP, then "all" you need to do is to pick your favorite NP-hard problem and design a polynomial time algorithm for that. There are tons and tons of very famous NP-hard and NP-complete problems (and literally thousands of no-so-famous ones) and numerous people have studied them in depth in the hope of obtaining a polynomial time algorithm for any of them (so far with no success)[2]
- On the other hand, if you believe (or suspect, yada yada) P $\neq$ NP, then by proving that a problem is NP-hard (or NP-complete), you have effectively convinced yourself (and tons of other people) that this problem does not have a polynomial time (or "efficient") algorithm[3].

So how do we prove a problem is NP-hard? By **reduction** to any other NP-hard problem!

Reductions for Proving NP-Hardness

Suppose you want to prove that problem A is NP-hard. Suppose you also already know that problem B is NP-hard. "All" you have to do now is to prove that *any* polynomial-time algorithm for problem A can be used to solve problem B in polynomial-time also. But this is basically reducing problem B to A that you have done multiple times in this course (and surely elsewhere). The only thing we need to ensure that the reduction takes polynomial time itself.

The plan for testing whether a problem A is NP-hard then is simply to see if we can do reduce *any* other NP-hard problem B to this problem (in polynomial time). Note that in this plan, it is completely irrelevant that we do *not* know a polynomial-time algorithm for A. All we need to do is to show that *if* there is ever a polynomial-time algorithm for A, then the same algorithm would also imply a polynomial-time algorithm for B. But then we are done, since, *by definition*, having a polynomial-time algorithm for B implies P = NP.

Wait! For the above plan to work, we also need to have at least one NP-hard problem to begin with. That original problem is *Circuit-SAT* problem (proven to be NP-hard separately by Cook and Levin).

Remark: Before moving on, let us briefly also state what it takes to prove that a problem A is NP-complete (instead of NP-hard). In that case, we need to (1) do a reduction to prove that A is NP-hard *and* (2) give a poly-time verifier to prove that A belongs to NP (recall the definition of NP-complete problems).

Circuit-SAT Problem: The "First" NP-Complete Problem

The Circuit-SAT (short form for (boolean) circuit satisfiability) problem is defined as follows.

Problem 1 (Circuit-SAT Problem). We are given a boolean circuit C with (binary) AND, OR, (unary) NOT gates, and n input bits. For any $x \in \{0,1\}^n$, we use $C(x)$ to denote the value of circuit on x. The Circuit-SAT problem then asks does there exists at least one boolean input x such that $C(x) = 1$ or not? In other words, is this circuit *ever* satisfiable (outputs one) or not?

Circuit-SAT is in NP. A simple verifier is as follows:

- Proof: If C is satisfiable then there should exists some x where $C(x) = 1$. We can use any such x as a proof.
- The verifiers then gets the input circuit C and the proof x that $C(x) = 1$. We can evaluate $C(x)$ in the circuit in polynomial time by computing the value of each gate from bottom-up (or in a top-down fashion using a DFS-type algorithm). Either way, given a value x, computing $C(x)$ can be done in polynomial time and once we computed $C(x)$, we can check whether $C(x) = 1$ or not.

[2]Perhaps, the strongest evidence that P $\neq$ NP is that despite tremendous effort, none of these problems have been solved in polynomial time (yet!). Admittedly one could also argue that despite tremendous effort, no one has yet prove P $\neq$ NP either so maybe this is not such a strong evidence after all.

[3]Proving that a problem is NP-hard has this extra benefit that no one in their right mind would still expect you to give a polynomial-time algorithm for that problem even if that person sincerely believes P = NP.

As we designed a poly-time verifier for this problem, this means that Circuit-SAT is in NP.

Circuit-SAT is NP-Hard (NP-Complete). The is the so-called Cook-Levin theorem. Proving this result is beyond the scope of our course at this point and we shall assume its correctness for this course (you are strongly encouraged to take a look at the proof of this result in Chapter 34.3 of the CLRS book).

3 NP-Hardness Reductions

Equipped with the NP-hardness of Circuit-SAT (namely, Cook-Levin theorem) and the reduction plan outlined above, we are going to prove that a couple of different problems are NP-hard.

3-SAT Problem

To define the 3-SAT problem, we first need some definition. We say that a boolean formula is a CNF if it is AND of multiple *clauses*, each of which is OR of multiple *literals* (i.e., a variable or its negation). For instance,

- A CNF formula:
$$\underbrace{(a \vee b \vee c)}_{\text{clause}} \wedge (\bar{a} \vee \overbrace{b}^{\text{literal}}) \wedge \underbrace{(\bar{a} \vee \bar{b} \vee c)}_{\text{clause}} \wedge (\overbrace{\bar{b}}^{\text{literal}} \vee \bar{c}).$$

- Not a CNF formula:
$$(a \wedge b \wedge c) \vee (\bar{a} \wedge b).$$

A 3-CNF formula is then any CNF formula where each clause has *at most* 3 literals. We are now ready to define the 3-SAT problem (sometimes called 3-CNF-SAT problem).

Problem 2 (**3-SAT Problem**)**.** We are given a boolean 3-CNF formula Φ with n variables and m clauses. For any $y \in \{0,1\}^n$, we use $\Phi(y)$ to denote the value of the formula when assigning y to the variables of the formula. The 3-SAT problem asks does there exists at least one assignment y such that $\Phi(y) = 1$ or not? In other words, is this 3-CNF formula *ever* satisfiable or not?

3-SAT is in NP. Similar to the Circuit-SAT problem a simple verifier is as follows:

- Proof: If Φ is satisfiable then there should exists some assignment y to the variables where $\Phi(y) = 1$. We can use any such y as a proof.
- The verifiers then gets the input formula Φ and the proof y that $\Phi(y) = 1$. We can evaluate $\Phi(y)$ by computing the value of each clause and checking whether every clause is satisfied or not and hence check for ourself whether $\Phi(y) = 1$ indeed or not.

As we designed a poly-time verifier for this problem, this means that 3-SAT is in NP.

3-SAT is NP-hard. This is the first time we can use our plan above for proving NP-hardness of 3-SAT by doing a reduction (from literally the only remaining NP-hard problem at this point, namely, Circuit-SAT).

The goal is to show that *any* polynomial time algorithm for 3-SAT also implies a polynomial time algorithm for Circuit-SAT (and since Circuit-SAT is NP-hard, this implies P = NP, which in turn implies 3-SAT is also NP-hard). The reduction is as follows.

Reduction: Given an input C to the Circuit-SAT problem, we turn it into a 3-CNF formula Φ as follows:

1. Define a new variable for every *wire* in the circuit C (including the input wires called variables and the output wire). These are all the variables in the formula Φ.
2. For every gate G in the circuit C we add the following clauses to the formula:
   (a) AND: Let a be the variable for the output wire of this gate and b and c be the variables for input wires. We want to have $a = b \wedge c$ in our formula. To do so, we add the following clauses to Φ:
   $$(a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c).$$
   (b) OR: Let a be the variable for the output wire of this gate and b and c be the variables for input wires. We want to have $a = b \vee c$ in our formula. To do so, we add the following clauses to Φ:
   $$(\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}).$$
   (c) NOT: Let a be the variable for the output wire of this gate and b be the variable for the input wire. We want to have $a = \bar{b}$ in our formula. To do so, we add the following clauses to Φ:
   $$(a \vee b) \wedge (\bar{a} \vee \bar{b}).$$
3. This concludes the description of the formula Φ.

We then simply run our (supposed) algorithm for 3-SAT on the formula Φ and if the algorithm outputs Φ is satisfiable, we output that C is satisfiable and otherwise output C is not satisfiable.

Similar to any other algorithm, this reduction is also simply an algorithm (although technically speaking we do not know the algorithm for 3-SAT inside it and instead assume there is a poly-time algorithm for 3-SAT but this does not change anything). The runtime of this algorithm is clearly polynomial if we have a poly-time 3-SAT algorithm since size of Φ is polynomial in the size of circuit C. It thus only remains to prove the correctness of this reduction (similar to any other algorithm) which we will do in the next lecture.