CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 24

April 21, 2022

Instructor: Sepehr Assadi

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

1 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:

- 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)[1].
- 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[2].

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

[1]Perhaps, the strongest evidence that P $\neq$ NP is that despite tremendous effort, none of these problems have been solved in polynomial time (yet!). Although 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.

[2]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.

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.

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.