Lecture 19

NP-Completeness I

19.1 Overview

In the past few lectures we have looked at increasingly more expressive problems that we were able to solve using efficient algorithms. In this lecture we introduce a class of problems that are so expressive — they are able to model *any* problem in an extremely large class called **NP** — that we believe them to be *intrinsically unsolvable by polynomial-time algorithms.* These are the **NP-complete** problems. What is particularly surprising about this class is that they include many problems that at first glance appear to be quite benign. Specific topics in this lecture include:

- Reductions and expressiveness
- Informal definitions and the ESP problem
- Formal definitions: decision problems, P and NP.
- Circuit-SAT and 3-SAT

19.2 Introduction: Reduction and Expressiveness

In the last few lectures have seen a series of increasingly more expressive problems: network flow, min cost max flow, and finally linear programming. These problems have the property that you can code up a lot of different problems in their "language". So, by solving these well, we end up with important tools we can use to solve other problems.

To talk about this a little more precisely, it is helpful to make the following definitions:

Definition 19.1 *We say that an algorithm runs in* **Polynomial Time** *if, for some constant c, its running time is $O(n^c)$, where n is the size of the input.*

In the above definition, "size of input" means "number of bits it takes to write the input down". So, to be precise, when defining a problem and asking whether or not a certain algorithm runs in polynomial time, it is important to say how the input is given. For instance, the basic Ford-Fulkerson algorithm is *not* a polynomial-time algorithm for network flow when edge capacities are written in binary, but both of the Edmonds-Karp algorithms *are* polynomial-time.

Definition 19.2 *A Problem A is* **poly-time reducible** *to problem B (written as $A \leq_p B$) if, given a polynomial-time algorithm for B, we can use it to produce a polynomial-time algorithm for A.*[1] *Problem A is* **poly-time equivalent** *to problem B ($A =_p B$) if $A \leq_p B$ and $B \leq_p A$.*

For instance, we gave an efficient algorithm for Bipartite Matching by showing it was poly-time reducible to Max Flow. Notice that it could be that $A \leq_p B$ and yet our fastest algorithm for solving problem A might be slower than our fastest algorithm for solving problem B (because our reduction might involve several calls to the algorithm for problem B, or might involve blowing up the input size by a polynomial but still nontrivial amount).

19.3 Our first NP-Complete Problem: ESP

Many of the problems we would like to solve have the property that if someone handed us a solution, we could at least check if the solution was correct. For instance the Traveling Salesman Problem asks: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" We may not know how to find such a tour quickly, but if someone gave such a tour to us, we could easily check if it satisfied the desired conditions (visited all the vertices and had total length at most k). Similarly, for the 3-Coloring problem: "Given a graph G, can vertices be assigned colors red, blue, and green so that no two neighbors have the same color?" we could easily check a proposed solution if we had one. Problems of this kind are called **NP** problems (we define the class **NP** formally in Section 19.4).

Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem of this kind. Here is a natural candidate:

Definition 19.3 Existence of a verifiable Solution Problem (ESP)*: The input to this problem is in three parts. The first part is a program $V(I, X)$, written in some standard programming language, that has two arguments and outputs a binary YES or NO. The second part is a string I intended as a first argument, and the third part is a bound b written in unary (b bits). Question: does there exist a string X, $|X| \leq b$, such that $V(I, X)$ halts in at most b steps and outputs YES?*[2]

Why are we calling this the "existence of a verifiable solution problem" (besides the ESP acronym)?[3] Consider some problem we might want to solve like 3-COLORING: Given a graph G we want to know if there is an assignment of colors to the vertices such that at most three different colors are used and no two neighbors have the same color. We don't know any fast ways of solving that problem, but we can easily write a program V that given inputs $I = G$ and $X =$ an assignment of colors to the vertices, verifies whether X indeed satisfies our requirements (uses at most three colors and gives no two adjacent vertices the same color). Furthermore, this solution-verifier is linear time. So, if we had an algorithm to solve the ESP, we could feed in this V, feed in the graph G, feed in a bound b that is linear in the size of G, and solve the 3-COLORING problem.

---

[1] Formally, $A \leq_p B$ if problem A is solvable in polynomial time given access to an $O(n^c)$-time black-box oracle for solving arbitrary instances of problem B.

[2] The reason for b is that we will want the problem to have the property that if we "had ESP" and could guess the string X (if it exists), we could at least check our guess in polynomial time (by running V for b steps).

[3] Thanks to Manuel Blum for suggesting the acronym.

Technically, the above procedure just tells us if a coloring exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question to find a coloring? One way is we can pick some vertex v and start adding edges from v to the rest of G one at a time, at each step re-running the above procedure. If the answer is "NO" then we take the edge back out. At the end of this procedure, we can give v and all nodes v is not connected to one color (say, red) and then use BFS to 2-color the remainder.

Let's try another problem. Say we wanted to factor a large integer N written in binary. We don't know any polynomial-time algorithms for solving that problem. But, we can easily write a verifier that given N and a proposed factor X, tells us if X is a solution to the problem. In fact, let's modify this slightly (you'll see why in a second) so the verifier takes in an additional integer k (so $I = (N, k)$) and outputs YES iff X divides N *and* $1 < X < k$.

So, if we can solve the ESP, we can tell if N has a factor between 2 and $k - 1$ by feeding V and $I = (N, k)$ into our solver. Then, if we want to actually *find* a factor, we can do binary search on k. (That's why we needed the extra input k.)

In fact, by design, we could use a polynomial-time algorithm for the ESP to solve *any* problem for which the size of a solution is polynomial in the size of the input, and we have a polynomial-time method for *verifying* if a proposed solution is correct: we just write down the verifier V and then make the length bound b big enough to handle the running time of V.

Interestingly, the ESP also belongs to this same category (as pointed out in Footnote 2). Namely if someone hands us a proposed solution X, we can check it by just running V.

19.3.1 ESP, NP, and NP-completeness

This class of problems — problems for which we can efficiently verify a proposed solution — is called **NP**. A problem Q is said to be NP-complete if (a) Q is in **NP** and (2) you could use a polynomial-time algorithm for Q to solve *any* problem in **NP** in polynomial time. That is, for any Q' in **NP**, we have $Q' \leq_p Q$. And we have just proven our first **NP**-complete problem, namely the ESP.

This ESP problem seems pretty stylized. But we can now show that other simpler-looking problems have the property that if you could solve them in polynomial-time, then you could solve the ESP in polynomial time as well, so they too are **NP**-complete.

So, the way to think of **NP**-completeness is that an **NP**-complete problem is highly expressive. It is so expressive, that we believe there are no polynomial-time algorithms for solving them. In particular, if we *could* solve an **NP**-complete problem in polynomial-time, then it would mean that for any problem where we could *check* a proposed solution efficiently, we could also *find* such a solution efficiently. Now, onto formal definitions.

19.4 Formal definitions: P, NP, and NP-Completeness

We will formally be considering decision problems: problems whose answer is YES or NO. E.g., "Does the given network have a flow of value at least k?" or "Does the given graph have a 3-coloring?" For such problems, we can split all possible instances into two categories: YES-instances (whose correct answer is YES) and NO-instances (whose correct answer is NO). We can also put any ill-formed instances into the NO category. We now define the complexity classes **P** and **NP**.

Definition 19.4 P *is the set of decision problems solvable in polynomial time.*

E.g., the decision version of the network flow problem: "Given a network G and a flow value k, does there exist a flow $\geq k$?" belongs to **P**.

Definition 19.5 NP *is the set of decision problems that have polynomial-time* verifiers. *Specifically, problem Q is in* **NP** *if there is a polynomial-time algorithm $V(I, X)$ 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, $V(I, X) = NO$.*

Furthermore, X should have length polynomial in size of I (since we are really only giving V time polynomial in the size of the instance, not the combined size of the instance and solution).

The second input X to the verifier V is often called a *witness*. E.g., for 3-coloring, the witness that an answer is YES is the coloring. For factoring, the witness that N has a factor between 2 and k is a factor. For the Traveling Salesman Problem: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" the witness is the tour. All these problems belong to **NP**. Of course, any problem in **P** is also in **NP**, since V could just ignore X and directly solve I. So, $\mathbf{P} \subseteq \mathbf{NP}$.

A huge open question in complexity theory is whether $\mathbf{P} = \mathbf{NP}$. It would be quite strange if they were equal since that would mean that any problem for which a solution can be easily *verified* also has the property that a solution can be easily *found*. So most people believe $\mathbf{P} \neq \mathbf{NP}$. But, it's very hard to prove that a fast algorithm for something does *not* exist. So, it's still an open problem.

Definition 19.6 *Problem Q is* **NP***-complete if:*

1. *Q is in* **NP***, and*
2. *For any other problem Q' in* **NP***, $Q' \leq_p Q$.*

So if Q is **NP**-complete and you could solve Q in polynomial time, you could solve *any* problem in **NP** in polynomial time. If Q just satisfies part (2) of the definition, then it's called **NP**-hard.

As we showed above, the ESP is **NP**-complete: it belongs to **NP** (that was the reason for including the bound b in unary, so that running the verifier for b steps counts as being polynomial-time), and we saw that we could use a polynomial-time algorithm for the ESP to solve any other problem in **NP** in polynomial-time.

19.5 Circuit-SAT and 3-SAT

Though the ESP is **NP**-complete, is a bit unweildy. We will now develop two more natural problems that also are **NP**-complete: CIRCUIT-SAT and 3-SAT. Both of them will be obviously in **NP**. To show they are **NP**-complete, we will show that ESP $\leq_p$ CIRCUIT-SAT, and then that CIRCUIT-SAT $\leq_p$ 3-SAT. Notice that this is enough: it means that if you had a polynomial-time algorithm

for 3-SAT then you would also have a polynomial-time algorithm for CIRCUIT-SAT; and if you had a polynomial-time algorithm for CIRCUIT-SAT, then you would also have a polynomial-time algorithm for ESP; and we already know that if you have a polynomial-time algorithm for ESP, you can solve any problem in **NP** in polynomial-time. In other words, to show that a problem Q is **NP**-complete, we just need to show that $Q' \leq_p Q$ for *some* **NP**-complete problem Q' (plus show that $Q \in$ **NP**).

Definition 19.7 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?*

Theorem 19.1 *CIRCUIT-SAT is* **NP***-complete.*

Proof Sketch: First of all, CIRCUIT-SAT is clearly in **NP**, since you can just guess the input and try it. To show it is **NP**-complete, we need to show that if we could solve this, then we could solve the ESP. Say we are given V, I, and b, and want to tell if there exists X such that $V(I, X)$ halts and outputs YES within at most than b steps. Since we only care about running V for b steps we can assume it uses at most b bits of memory, including the space for its arguments. We will now use the fact that one can construct a RAM with b bits of memory (including its stored program) and a standard instruction set using only $O(b \log b)$ NAND gates and a clock. By unrolling this design for b levels, we can remove loops and create a circuit that simulates what V computes within b time steps. We then hardwire the inputs corresponding to I and feed this into our CIRCUIT-SAT solver. ■

So, we now have one more **NP**-complete problem. Still, CIRCUIT-SAT looks complicated: we weren't expecting to be able to solve it in polynomial-time. However, now we will show that a much simpler-looking problem, 3-SAT has the property that CIRCUIT-SAT $\leq_p$ 3-SAT.

Definition 19.8 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. E.g., $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge \ldots$. Goal: find an assignment to the variables that satisfies the formula if one exists.*

We'll save the proof that 3-SAT is **NP**-complete for the next lecture, but before we end, here is formally how we are going to do our reductions. Say we have some problem A that we know is **NP**-complete. We want to show problem B is **NP**-complete too. Well, first we show B is in **NP** but that is usually the easy part. The main thing we need to do is show that $A \leq_p B$; that is, any polynomial-time algorithm for B would give a polynomial-time algorithm for A. We will do this through the following method called a *many-one* or *Karp* reduction:

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

1. if x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. if x is a NO-instance of A then $f(x)$ is a NO-instance of B.
3. f can be computed in polynomial time.

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