Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 12

Lecture 56

Welcome to the 56th lecture of the second-level algorithm course. In the last class, we introduced the concept of the complexity classes P, NP, NP-hard, and NP-completeness. Let us continue that study in this lecture. In the last class, we assumed, without loss of much generality, that all our problems are decision problems. Let us begin this lecture by justifying that claim. That concept is called self-reduction.

Most problems that we deal with in the real world, or most well-studied problems, are self-reducible. That means what? That means the following. Let me write it as a theorem for a particular problem, say satisfiability or SAT. So, in the standard version of satisfiability, we are given an input instance, and we either have to say whether it is satisfiable or not. If it is satisfiable, we also have to output a satisfying assignment. That is the optimization version of satisfiability. So, let us write down that problem statement. Optimization version of satisfiability. So, in this case, the input is a Boolean formula, let us call it f, with variables $x_1, \ldots, x_n$ and we need to output or we need to check if the formula if $x_1, \ldots, x_n$ is satisfiable, if the formula is satisfiable then we also need to output a satisfying assignment. So, this is the optimization version of course, in the decision version we do not need to output a satisfying assignment even if the formula is satisfiable. So, clearly the optimization version is more general, but here is the interesting theorem of self reduction.

There is a polynomial time algorithm for the optimization version of satisfiability if and only if there is polynomial time algorithm for the decision version of satisfiability. proof . One direction is obvious if there exists a polynomial time algorithm for the optimization version of satisfiability then that algorithm also solves the decision version of satisfiability. It is the other direction that is more interesting and for the other direction we use the following algorithm. First for the other direction We use the following algorithm. We suppose in this direction we will assume that the decision version is

polynomial-time solvable. That means we assume that there is a polynomial-time algorithm. Let us call that algorithm B. for the decision version of satisfiability. So, what we do is we suppose we are given an arbitrary instance of the optimization version of satisfiability, which is just a formula f. So, let $f(x_1,\ldots,x_n)$ be any instance of the optimization version satisfiability. We run the algorithm B on $f(x_1,\ldots,x_n)$. If B outputs no, then we output no because in this case we know that the input formula is not satisfiable. Otherwise, we try setting or we run B on we set x_1 to true and on f set x_1 to false, ok, and see which one is satisfiable. At least one of them is satisfiable. We set x_1 to that.

So, let us look at the pseudocode. So, let us call it optimization version of satisfiability. It takes an input formula, Boolean formula $f(x_1,\ldots,x_n)$, as input. What it first does is if n is 0. then it is satisfiable, then return true. That means if the formula is empty, else what we do is we try both setting x_1 to true or x_2 to false; one of them must be true if $f(x_1,\ldots,x_n)$ is true. So, Loop in this case. So, for i equal to 1 to n, if f of x_1, x So, in the i-th iteration, we have already set the variables $x_1,\ldots,x_{i-1}$.

So, that assignment—let's denote it by bar—and set x_1 to true in this case. If this formula is satisfiable, then we set x_i to true. in If this formula is satisfiable, then we set x_i to false. If both of them are not satisfiable, then—else if—the formula is not satisfiable, then we return no.

So, in the for loop, we set $x_1,\ldots,x_n$, and then we return the satisfying assignment. So, it is easy to see that the algorithm is correct, and the proof of correctness we leave as homework. The algorithm written above. What is the runtime of the algorithm? We run the algorithm B twice in every iteration—at most twice in every iteration.

So, the runtime becomes $O(n)$ times the runtime of B, ok. So, in particular, if B runs in polynomial time, then the above algorithm also runs in polynomial time. So, for this reason, what we have mentioned in the last class is that the assumption that the problems are decision problems is almost without loss of generality. I strongly suggest that for all the problems studied in this entire course, you try to design a self-reduction from the optimization version to the decision version of the problem. So, now let us introduce another important complexity class, which is called co-NP.

co-NP complement NP. Formally, a decision problem belongs to co-NP if there exists a polynomial-time algorithm that can verify every no-instance of the problem given an input instance and a polynomial-size

So, for example, unsatisfiability belongs to the complexity class co-NP. Indeed, to prove that some unsatisfiability instance is a no-instance—an instance of unsatisfiability is a no-instance—it is enough to show a satisfying assignment of the input formula. So, any satisfying assignment of the input formula can be verified in polynomial time to prove that the formula is not unsatisfiable.

That is, it is a no instance, ok. So, these are the complement problems of NP, and these also we can make similar observations. Like NP, the complexity class P is a subset of co-NP, ok. Because if you can solve the problem in polynomial time, you can verify the no instance in polynomial time with an empty certificate. So, this shows that P is a subset of

NP intersection co-NP. Pictorially, if this is NP, this is co-NP, the complexity class P is a subset of the intersection of NP and co-NP, ok. So, let us stop here. Thank you.