Lecture 6: The problem with Diagonalization

Anup Rao

April 13, 2022

THE ONLY WAY WE KNOW how to prove lower bounds on the running time of Turing Machines is via diagonalization. Can we hope to show that $\mathbf{P} \neq \mathbf{NP}$ by some kind of diagonalization argument? In this lecture, we discuss an issue that is an obstacle to finding such a proof.

Definition 1 (Oracle Machines)**.** *Given a function $O : \{0,1\}^* \to \{0,1\}$, an* oracle-machine *is a Turing Machine that is allowed to use a special oracle tape to make queries to O. Each query to O takes unit time.*

We can define $\mathbf{P}^O$, $\mathbf{NP}^O$ as functions computable in poly time (resp nondeterministic poly time) with oracle access to O.

Then we have the following theorem:

Theorem 2. *There exists an oracle A such that $\mathbf{P}^A = \mathbf{NP}^A$, and an oracle B such that $\mathbf{P}^B \neq \mathbf{NP}^B$.*

The theorem gives a hint about one of the ways in which it will be hard to determine whether or not $\mathbf{P} = \mathbf{NP}$. Any such proof must not work in the *relativized* worlds where access to A, B is permitted. On the other hand, the kinds of proofs that we have seen using diagonalization *do relativize*—the same argument would work even if the machines have oracle access to some oracle O.

Proof Let A be the function that on input α, x outputs 1 if and only if $M_\alpha(x)$ outputs 1 in $2^{|x|}$ steps. Then $\mathbf{P}^A = \mathbf{EXP}$, since every exponential time computation can be simulated with access to A, and every query to A can be simulated in exponential time. Also $\mathbf{NP}^A = \mathbf{EXP}$, since in exponential time we can simulate all queries to A and simulate all nondeterministic choices.

To simulate a machine M_α, that runs in time 2^{n^c}, we first create a new machine M'_α that runs M_α on the first $n^{1/c}$ bits of its input. Then we call the oracle on $M_{\alpha'}(y)$, where y is the input of length n^c with x as the first $n^{1/c}$ bits of y.

The second part is more interesting. We shall define an oracle $B : \{0,1\}^* \to \{0,1\}$ and a function $f \in \mathbf{NP}^B$ such that $f \notin \mathbf{P}^B$. f is defined in terms of B as follows:

$$f(x) = \begin{cases} 1 & \text{if there exists } y \text{ such that } |y| = |x| \text{ and } B(y) = 1, \\ 0 & \text{else.} \end{cases}$$

We first show that $f \in \mathbf{NP}^B$: a non-deterministic machine can guess y of the same length as x, and make a single query to verify that $B(y) = 1$.

To define B, we shall use diagonalization. Let $M_1, M_2, \ldots, M_i, \ldots$, be an enumeration of all machines that query B, with the feature

that every machine occurs infinitely often in the sequence. (Such an enumeration exists if we allow our programming language to have redundant lines). Our goal is to make sure that the $i'th$ machine fails to compute the correct value of $f(x)$ in time $2^{n/10}$, for some n where $n = |x|$. To do this we define the value of B gradually. We define the value of B in phases. After each phase, we shall have defined the value of B on a finite set of strings.

In Phase i, let t be so large that the value of B is not yet defined on each string of length t. Then run the i'th machine $M_i(1^t)$ for $2^{t/10}$ steps. Each time M_i queries a string of B whose value has not yet been defined, return 0 and define the value of B on that string to be 0. If M_i halts with value 1, then set B to be 0 on all strings of length t. If M_i halts with value 0, then pick a string y of length t that $M_i(1^t)$ did not query (note that such a string always exists since there are 2^t binary stings of length t and M_i did not take more than $2^{t/10}$ steps), and set $B(y) = 1$.

Set the value of B on strings that are not defined by the above process to be 0.

Suppose for the sake of contradiction that $f \in \mathbf{P}^B$. Then consider the machine M that computes f. Let i be the index such that the i'th machine in the enumeration is M and t be such that $M_i(1^t)$ was used to define B on strings of length t during the i'th phase. Since the machine occurs infinitely often, there is an i for which $2^{t/10}$ exceeds the running time of the machine. Clearly, $f(1^t) \neq M(1^t)$ and hence M does not compute f. ■

Polynomial time Reductions

One of the central questions in complexity theory is whether or not $\mathbf{P} = \mathbf{NP}$. Although we don't know the answer to this question, we can prove a lot about the class **NP**, via the concept of polynomial time reductions:

Definition 3. *A function f is polynomial time reducible to a function g if there is a polynomial time computable function h such that $f(x) = g(h(x))$. We write $f \leq_P g$.*

Note that the above definition is not the only one that makes sense. In general it makes sense to allow our reductions to make multiple calls to the problem being reduced to. However, we will be able to prove many of our results using the stronger notion above, so that is what we shall use.

Definition 4. *We say f is* **NP***-hard if $g \leq_P f$ for every $g \in \mathbf{NP}$. We say f is* **NP***-complete if f is* **NP***-hard and $f \in \mathbf{NP}$.*

Theorem 5. *Here are some easy facts that one can prove about reductions:*

- *If $f \leq_P g$ and $g \leq_P h$, then $f \leq_P h$.*
- *If f is* **NP***-hard and $f \in \mathbf{P}$, then $P = NP$.*
- *If f is* **NP***-complete, then* $\mathbf{P} = \mathbf{NP}$ *if and only if $f \in P$.*

NP-*complete problems*

The above definitions make sense because we do know of examples of **NP**-complete problems.

Circuit-Sat

Definition 6. *CircuitSat* $: \{0,1\}^* \to \{0,1\}$ *is the function that views its input as a circuit C and outputs 1 iff $\exists x$ such that $C(x) = 1$.*

I have claimed in class that circuits can simulate Turing Machines. Here is what you can actually prove in this regard:

Theorem 7. *If a function $f : \{0,1\}^* \to \{0,1\}$ can be computed in time $t(n)$ by a Turing machine, then for every n there is a circuit of size $O\left(t(n) \log t(n)\right)$ that computes f restricted to the inputs of size n.*

Although we did not prove this theorem in class, we sketched how you could find a circuit of size $O(t(n)^2)$ that computes f. The idea was to add a layer of gates that maintains the entire state of the Turing machine—contents of all tapes, pointers, and the line of code being executed. Then we add a new layer that computes this configuration after one execution step of the Turing machine, using the earlier configuration as input. A single configuration can be written down using $O(t(n))$ gates since we only need to write down the values of the tapes up to $O(t(n))$ coordinates. The new configuration can be computed from the old one with $O(t(n))$ gates as well. After repeating this $O(t(n))$ times, we obtain the final configuration of the Turing machine, which must include the value of $f(x)$.

Theorem 8. *CircuitSat is* **NP***-complete.*

Proof It is clear that *CircuitSat* is in **NP**. Next we show that for every $f \in \mathbf{NP}$, $f \leq_P$ *CircuitSat*. Let V be a verifier for f. Then to compute $f(x)$, the reduction will build the circuit $C_x(w)$ that computes $V(x,w)$, where here w are the input variables to the circuit and x is the input. Since $f(x) = 1$ if and only if there exists w such that $C_x(w) = 1$, we can determine the value of f by computing *CircuitSat*(C_x). ■