**CS760 Topics in Computational Complexity** **2024-25 Sem I**

# Lecture 7: 23-08-2024

*Scribe: Aditya Singh, Tec Narayan* *Lecturer: Rohit Gurjar*

## 1 Time Hierarchy Theorem

In the previous class, we defined two time-constructible functions $f(n)$ and $g(n)$, where $f(n)$ is significantly smaller than $g(n)$, i.e., $f(n)\log(f(n)) = o(g(n))$.

We define $D(x)$ as the output of running a Universal Turing Machine on the encoding $\langle M_x, x\rangle$ for $g(|x|)$ steps:

- If there is no output: Return 0.
- If the output is $b$: Return $1 - b$.

The function $D(x)$ can be computed in $\mathcal{O}(g(n))$ time (by definition), and we claim that it cannot in $\mathcal{O}(f(n))$ time.

Let $N$ be a Turing Machine that always halts in $cf(n)$ time.

**Claim 7.1.** *The machine $N$ will differ from $D(x)$ on some input.*

The encoding for which this happens is the encoding of $N$ itself. Since $N$ always halts in $cf(n)$ time, we can simulate its computation on input $x$ by a universal Turing machine in time $c'cf(|x|)\log(|x|)$.

Assume:

$$c' \cdot cf(|x|)\log(|x|) < g(|x|) \tag{1}$$

This holds only when $|x|$ is sufficiently large.

From (1), we can conclude that $D$ will flip the output of $N$ on its own encoding in $g(|x|)$ steps. This would mean that $N$ and $D$ will disagree on some input. To satisfy the condition that $|x|$ must be large enough, we use the fact that every Turing Machine $N$ has infinitely many encodings. Therefore, we can choose a sufficiently large encoding for $N$.

## 2 Non-Deterministic Time Hierarchy

For the Time Hierarchy Theorem in the non-deterministic case, we only need the condition $f(n+1) = o(g(n))$ to show that $NTIME(f(n)) \subsetneq NTIME(g(n))$. We skip the proof.

**Theorem 7.2** (Ladner's Theorem)**.** *If $P \neq NP$, then there exists a decision problem in NP that is neither in P nor NP-complete.*

**Idea of the Proof:** We first define a language using a recursive approach and assume that it belongs to the class P. This assumption would imply that P = NP. Next, we consider the language to be NP-complete and use a diagonalization argument to show that this also leads to P = NP. By exploring both scenarios, we demonstrate that both assumptions lead to the conclusion that P = NP, which contradicts our initial assumption that P $\neq$ NP. Hence, there must exist a language in NP that is neither in P nor NP-complete. See Arora-Barak Chapter 3 for the construction of such a language.

Most problems fall into one of two categories: those in P and those that are NP-complete. However, there are some problems in NP for which it is not yet clear whether they are in P or NP-complete. Examples of such problems include Integer Factoring, Nash Equilibrium (a search problem), Graph isomorphism etc.

# 3 Oracle Turing Machines

Oracle Turing Machines are a theoretical model of computation that provides a special oracle or "black box" which can answer certain questions instantly.
**Example:** The class $P^{SAT}$ represents polynomial time Turing machines with access to a SAT oracle. This means that the machine can query the SAT oracle to solve satisfiability problems in constant time.
***Remark:*** Just like we did a diagonalization proof for time hierarchy, one may wonder if there is a diagonalization proof for showing P $\neq$ NP. One should note that the diagonalization technique does not use much about the computation model. It only assume few properties like: every Turing machine has an encoding, every string represents a Turing machine, and there is a universal Turing machine to simulate any given Turing machine. That means if some diagonalization argument works for showing P $\neq$ NP, then it will also show $P^O \neq NP^O$ for any oracle $O$. However, it turns out that there exist oracles such that $P^O = NP^O$. This means we cannot use a diagonalization argument for showing P $\neq$ NP.
Let's check if SAT is such an oracle.

## 3.1 Checking whether $P^{SAT} = NP^{SAT}$

For $NP^{SAT}$, the verifier operates within $P^{SAT}$. The verifier is used to check the validity of a certificate with respect to a SAT oracle. Consider the problem $Q$ which needs to decide whether $\exists x \forall y\ \rho(x, y)$ is true for a given formula $\rho$. Here $x$ is a set of $n$ variables and $y$ is a set of $m$ variables.

$$Q \in NP^{SAT} = NP^{UNSAT}$$

To see this, we can provide $x$ as a certificate and then $\forall y\ \rho(x, y)$ can be verified using a SAT oracle. This is because $\forall y\ \rho(x, y)$ is the negative of $\exists y\ \neg\rho(x, y)$.
On the other hand, it is unlikely that problem $Q$ is in $P^{SAT}$.

## 3.2 More Powerful Oracles

A more powerful oracle is one that can handle exponential-time problems. We will show that

$$P^{EXP} = NP^{EXP}$$

Here, EXP oracle is one that can solve a particular EXP-complete problem. But we can also solve other exponential-time problems, because by definition, they can be reduced to an EXP-complete problem in polynomial time.
We can easily show that EXP $\subseteq P^{EXP} \subseteq$ EXP, i.e., $P^{EXP} =$ EXP.
Now, consider $NP^{EXP}$. Clearly $P^{EXP} \subseteq NP^{EXP}$. We only need to argue that $NP^{EXP} \subseteq$ EXP. We can go over all possible certificates in exponential time. For each certificate while verifying we may need EXP oracle, which we will just solve in exponential time. So, overall time is exponential times exponential, which is exponential.
***Note:*** NP $\subseteq P^{NP}$. And $P^{NP} \subseteq$ NP is unlikely, because UNSAT is in $P^{NP}$.

# 4 Infinite Hierarchy Theorem

The Time Hierarchy Theorem says that there is an infinite hierarchy of time complexity classes, with each class containing strictly more problems than the one below it.
It turns out that if P $\neq$ NP then there is also an infinite hierarchy between P and NP-complete This is a generalization of Ladner's theorem. It says that there are infinitely many complexity classes between P and NP-Complete, assuming P $\neq$ NP. What this means is that there is an infinite chain of problems, such that any problem is not polytime-reducible to the one below it. As a consequence, the problems in P and those in NP-Complete are "infinitely far apart" in terms of computational complexity.
Interesting to note that we do not currently know how far certain problems, such as Integer Factoring, are from being in P.

# 5 Padding Argument

The Padding Argument is a technique used in complexity theory to show relationships between complexity classes under certain assumptions. Here we show that if P = NP, then it would follow that EXP = NEXP.

## Proof

Assume $P = \text{NP}$. We aim to prove that this implies EXP = NEXP.
Let $L \in \text{NEXP}$ be a language that can be decided by a non-deterministic Turing machine in exponential time. Define a new language $L'$ by padding the inputs of $L$ with an exponentially long string of ones:

$$L' = \{x \cdot 1^{2^{|x|^c}} : x \in L\}$$

Here $c$ is such that $L \in \text{NTIME}(2^{n^c})$. The padded string $1^{2^{|x|}}$ ensures that $L'$ has a structure that allows it to be decided in non-deterministic polynomial time. This is because the time complexity $2^{|x|^c}$ is polynomial in the new input size $2^{|x|^c}$. The same happens with the certificate size.

$$\implies L' \in \text{NP}$$

Since we assumed P = NP, it follows that:

$$\implies L' \in \text{P}$$

However, $L'$ is essentially $L$ with some padding, and the padding does not change the inherent difficulty of the problem. For any input $x$ of $L$, we pad it with $1^{2^{|x|^c}}$ and run the algorithm for $L'$. The algorithm which is polynomial time for the padded input, is exponential time in terms of $|x|$. Hence,

$$\implies L \in \text{EXP}$$

This shows that under the assumption P = NP, the original language $L$ (which was in NEXP) would also be in EXP. Therefore:

$$\implies \text{EXP} = \text{NEXP}$$

This completes the proof that P = NP implies EXP = NEXP.

# 6 NP-Hard Problem: MAX-CUT

In the last class, we proved that $3SAT \leq_P$ INDSET. Now, we will prove that INDSET $\leq_P$ MAX-CUT, which implies that MAX-CUT is NP-Hard.
We are considering the decision versions of the problems:
- **INDSET**: Given a graph $G$, does $G$ have an independent set of size $k$?
- **MAX-CUT**: Given a graph $H$, does $H$ have a cut with at least $k'$ cut-edges?

## 6.1 Reduction from INDSET to MAX-CUT

Given a graph $G = (V, E)$, we construct a new graph $H$ by adding a new vertex $w$ to $G$, so $H = G + \{w\}$. We connect each vertex $v \in V$ to $w$ with exactly $(\deg_G(v) - 1)$ edges.
We want to show that $G$ has an independent set of size $k$ if and only if $H$ has a cut with $k'$ cut-edges, for some $k'$ that is related to $k$. We will in fact show a stronger statement that if $G$ has maximum independent set size equal to $k$, then $k'$ is the size of the maximum cut in $H$.

## 6.2 Cut Construction

Let $S$ be a maximum independent set in $G$ with $|S| = k$. Consider the cut in $H$ given by $(S \cup \{w\}, V \setminus S)$.
For every vertex $v \in S$, all edges incident to $v$ in the original graph $G$ must be part of the cut in $H$, since $S$ is an independent set.
For every vertex $u \in V \setminus S$, since you have connected $u$ to $w$ with $(\deg(u) - 1)$ edges, all these edges will be part of the cut in $H$.
Thus, the size of the cut $S \cup \{w\}$ in $H$ can be expressed as:

$$\text{CUT}(S \cup \{w\}) = \sum_{v \in S} \deg(v) + \sum_{u \in V \setminus S} (\deg(u) - 1)$$

This simplifies to:

$$\text{CUT}(S \cup \{w\}) = 2|E| - (n - k)$$

where $n = |V|$ is the number of vertices in $G$. We define $k' := 2|E| - (n - k)$.

## 6.3 Proof of Maximality

Now, we show that $k'$ is indeed the maximum cut value.
Consider any cut in graph $H$. One of the sides must contain vertex $\{w\}$. So, we can assume the cut to be $S \cup \{w\}$, for some subset $S \subseteq V$ (not necessarily independent). The size of the cut is

$$\text{CUT}(S \cup \{w\}) = \sum_{v \in S} \deg(v) - 2|E(S)| + \sum_{u \in V \setminus S} (\deg(u) - 1)$$

where $E(S)$ is the number of edges within the set $S$.
This can be rewritten as:

$$\text{CUT}(S \cup \{w\}) = 2|E| - 2|E(S)| - n + |S|$$

We need to prove that:

$$\text{CUT}(S \cup \{w\}) \leq k'$$

That is:

$$2|E| - 2|E(S)| - n + |S| \leq 2|E| - (n - k)$$

This is equivalent to showing that for any subset $S \subseteq V$ in graph $G$,

$$|S| - 2|E(S)| \leq k(\text{maximum independent set size})$$

In other words $|S| - 2|E(S)|$ is maximized when $S$ is an independent set. The remaining part is left as homework.