University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: Siddarth Chalasani
Date: February 5, 2024

LECTURE

8

Oracles

This lecture, we will explore the power of Turing machines equipped with a magical "oracle" function. Specifically, we will analyse how deterministic and non-deterministic Turing machines perform when equipped with the same oracle. Furthermore, we will see (via Ladner's theorem) that there exists a notion of a "moderately hard" problem in NP.

8.1 Oracles

Informally, an oracle is a function or language that a given Turing machine can compute "for free". The TM can therefore leverage its solution to this oracle to compute other functions more efficiently.

More formally, let $O \subseteq \{0,1\}^*$. Then, a Turing machine equipped with oracle O can:

- write any string x on the oracle tape
- go into a special $QUERY$ state
- instantly receive answer on the oracle tape (1 if $x \in O$, 0 otherwise)

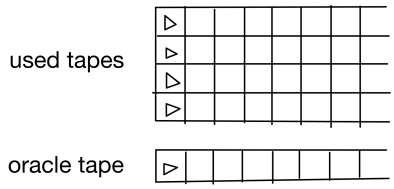

FIGURE 8.1: A Turing machine equipped with an oracle.

DEFINITION 8.1. P^O is the set of all languages decidable on a polytime TM with oracle O. Analogously, NP^O is the set of all languages decidable on a polytime NTM with oracle O.

DEFINITION 8.2. Let $\mathcal{C} \subseteq \mathcal{P}(\{0,1\}^*)$ be any family of languages. Then, $co\mathcal{C} = \{L : \overline{L} \in \mathcal{C}\}$.

REMARK 8.3. With the above definition in mind, we make the following two observations:

1. $NP \subseteq P^{SAT}$. This is true because the TM can run the polytime reduction from a given problem to SAT, and then just query SAT to find the answer.
2. $coNP \subseteq P^{SAT}$. This is true because of the symmetry of deterministic computing.

Given the above remark, we paint the following picture of complexity classes. Note that none of these subset relations are known to be proper. In particular, if $P = NP$, then all of these classes collapse to P.

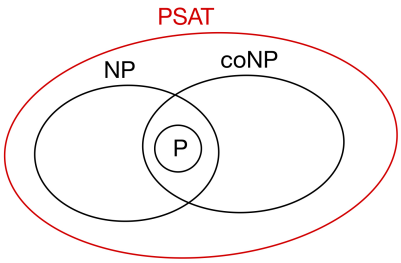

FIGURE 8.2: Complexity class view with P^{SAT}.

We now consider some challenge problems:

PROBLEM 8.4 (Challenge). If $SAT \in coNP$, then $coNP = NP$.

PROBLEM 8.5 (Challenge). Give an oracle O such that $P^O = NP^O$. Hint: consider an oracle so powerful that it makes all problems in P and NP trivial.

The following theorem shows that oracles are useful not only for bridging gaps between complexity classes, but also for separating them.

THEOREM 8.6. *There is an oracle $O \subseteq \{0,1\}^*$ such that $P^O \neq NP^O$.*

As with many results in this class, we will prove this theorem using diagonalization. The key idea is that a TM equipped with an oracle can only query the oracle polynomially many times. Therefore, there will always be a significant portion of unqueried strings. Taking advantage of this, we will construct our oracle iteratively so that the oracle maintains consistency on all the previously queried inputs while still having strings where the TM computes the incorrect result.

Proof. For a fixed oracle O, define $L_O = \{1^N : 0 \cap \{0,1\}^n \neq \phi\}$. This is the language that asks the question, "Does O contain a sting of length n?"

$L_O \in NP$, since on input 1^n, the NTM can guess $y \in \{0,1\}^n$ and check if $y \in O$.

Now, we will construct O such that $L_O \notin P_O$.

First, enumerate all polytime TMs: $M_1, M_2, \ldots, M_i, M_{i+1}, \ldots$, where M_i runs in time $n^i + i$ (pad sequence with repeated TMs as necessary).

We will construct O in stages $i = 1, 2, 3, \ldots$

For each stage i, we want to:

- declare the status of a finite set of inputs x (either $x \in O$ or $x \notin O$)
- ensure that $L_O \neq L(M_i^O)$ regardless of yet-to-be declared outputs

Here is the concrete algorithm for constructing O in stage i:

Stage i

1. Pick $n = n(i)$ so large that $2^n > n^i + i$ and every $x \in \{0,1\}^n$ is undeclared
2. Simulate M_i on 1^n. Every time M_i queries an undeclared string x, declare "$x \notin O$"
3. If $M_i(1^n) =$ `yes`, declare "$x \notin O$" for all $x \in \{0,1\}^n$.
   If $M_i(1^n) =$ `no`, declare "$x \in O$" for some previously undeclared $x \in \{0,1\}^n$
   Note that both of the above cases are consistent and possible because of step 1

□

8.2 Ladner's theorem

So far, we have seen "easy" problems (those in P), and "very hard" problems (those that are NP-hard). However, the question arises: are there problems that are "moderately hard"? That is, are there problems outside of P that are not NP-hard? Ladner's theorem proves the existence of such problems, given that $P \neq NP$.

THEOREM 8.7 (Ladner). *If $P \neq NP$, then there is $L \subseteq \{0,1\}^*$ such that*

- *$L \notin P$*
- *L is not NP-hard*

The basic idea behind this proof is to construct $L \subseteq SAT$ small enough that L is not NP-hard, but large enough that $L \notin NP$. It is clear that each of these conditions can be met independently: any finite subset of SAT cannot be NP-hard, and SAT itself is NP-hard and therefore not in P. In our proof, we will show that both of these conditions can be met concurrently.

Proof. Enumerate all polytime Turing machines: $M_1, M_2, \ldots, M_i, M_{i+1}, \ldots$ (algorithms)
Enumerate all polytime functions $\{0,1\}^* \to \{0,1\}^*$: $f_1, f_2, \ldots, f_i, f_{i+1}, \ldots$ (reductions)
Next, list all the strings $x \in SAT$.

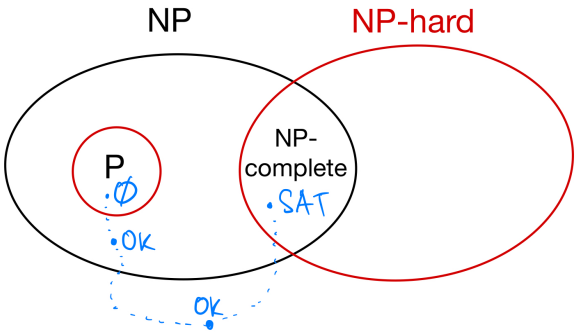

FIGURE 8.3: Idea behind Ladner's theorem.

Now, construct L in stages as follows:
Stage $i.a$: grow L by including strings in SAT until M_i can no longer decide L
Stage $i.b$: prune L until f_i can no longer be a reduction from SAT to L

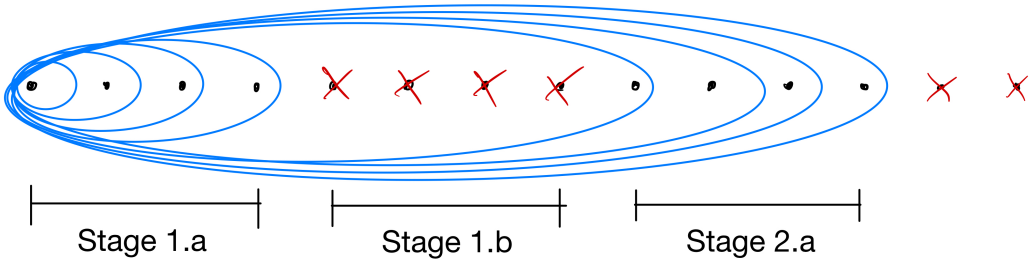

FIGURE 8.4: Stages of Ladner's theorem.

Why must Stage $i.a$ terminate?

- If M_i outputs `yes` on some string not in SAT, we are done
- Otherwise, M_i must misclassify infinitely many strings in SAT (because $SAT \notin P$). So, eventually, we will include a string in L that M_i misclassifies

Why must Stage $i.b$ terminate?
Assume otherwise: no matter how many strings we prune out, f_i is still a reduction from SAT to L. Then, f_i maps SAT to a finite subset of SAT. So, $SAT \in P$, a contradiction.

So, there exists neither a polytime TM that can decide L nor a polytime reduction from SAT to L. Therefore, we have constructed a language $L \subseteq SAT$ such that $L \notin P$ and L is not NP-hard. □

PROBLEM 8.8 (Challenge). Strengthen Ladner's theorem as follows:

THEOREM 8.9. *If $P \neq NP$, then there is $L \in NP$ such that $L \notin P$ and L is not NP-hard.*

That is, show that NP itself contains these "moderately hard" problems.

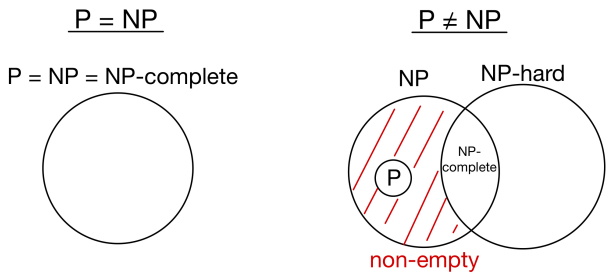

FIGURE 8.5: Improvement to Ladner's theorem.

8.3 Linear speedup theorem

Now, we will prove the linear speedup theorem, which was previously assigned as a challenge problem:

THEOREM 8.10. *If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$, then f is also computable in time $\epsilon \cdot T(n) + 2n$ for any $\epsilon > 0$.*

The main idea of the proof is to expand the alphabet so that each cell on the tape contains an arbitrarily large number of cells from the original tape. In this way, we can simulate many steps of the old TM in just a few steps of the new TM.

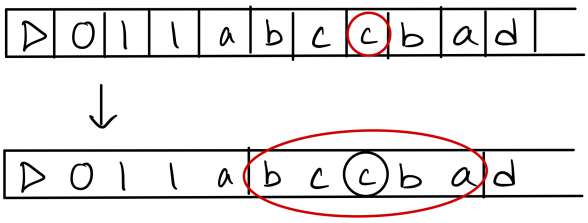

FIGURE 8.6: Alphabet expansion for linear speedup.

Proof. Suppose $f : \{0,1\}^* \to \{0,1\}^*$ is decided in time $T(n)$ by Turing machine M with alphabet Γ.

Fix $k \in \mathbb{N}$.
Construct a new alphabet $\Gamma' = (\Gamma \times \{0,1\})^k$ so that each cell on the new tape contains k cells of the old tape, with a special marker for the character within a cell where the computational head is currently positioned (see figure 8.6). Then, take the following steps on the new TM:

1. Step `LEFT` and store the contents in state
2. Step `RIGHT` and store the contents in state
3. Step `RIGHT` and store the contents in state
   With all of these cells memorized, there should be enough information to simulate k steps of the old TM. Do so, and memorize the changes to the tape. Apply any required updates to the right cell
4. Step `LEFT` and apply any required updates to the middle cell
5. If any updates need to be applied to the left cell, step `LEFT` and apply required updates
6. Step in the required direction so that head positions are consistent. Note that this takes at most 1 step because it is impossible to update the left cell and have head end up in right cell, as that would take more than k steps on the old TM

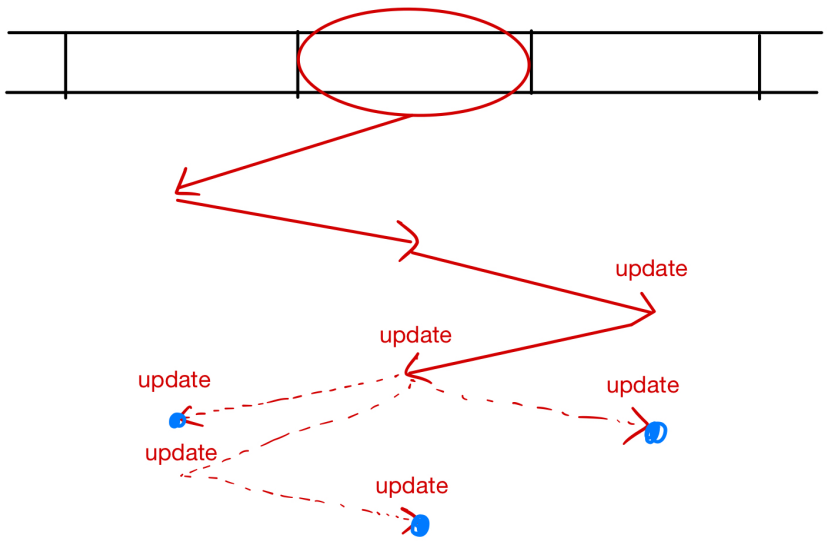

FIGURE 8.7: Update step for linear speedup.

Using the above algorithm, we simulate k steps on the old TM in at most 6 steps on the new TM. Combined with the "preprocessing" cost of compressing the input from n to $\frac{n}{k}$ terms, the total running time comes out to $2n + 6\left\lceil \frac{T(n)}{k} \right\rceil$. Take k as large as necessary. □