Oracles and Time Hierarchy Theorem

In this recitation, we covered the proof of the time hierarchy theorem, as well as the definitions of oracle Turing machines and complexity classes P^A and NP^A. We discussed the relationships between these oracle classes and other complexity classes, and explained how to design algorithms for oracle Turing machines.

Oracle Turing Machine

Oracle Turing machines are a new computation model that extends the standard Turing machine model. To introduce this model, we will discuss its definition. We will also introduce complexity classes under this model and their relationships with existing complexity classes, and discuss methods for designing algorithms in the oracle setting.

Definitions

Definition 1. A *deterministic Turing Machine M with an oracle for language A*, denoted M^A, is a deterministic Turing machine equipped with an additional oracle tape. This tape allows the machine to read and write just as the work tape. It also has an extra transition: let the content of oracle type be x, then it can "ask" the oracle whether $x \in A$, replacing the content of the oracle tape with the answer in one timestep.

Similarly, a *nondeterministic Turing Machine with an oracle for language A*, denoted N^A, is a nondeterministic Turing machine, with the same oracle tape and extra transition as the deterministic version.

Intuition. Intuitively, a (nondeterministic) Turing machine with an oracle for A is a (nondeterministic) Turing machine, *that has the additional ability to determine whether a given string is in the language A* by invoking the oracle.

New Complexity Classes: P^A, NP^A

With the definition of oracle Turing Machines, we can define classes P^A and NP^A.

Definition 2. P^A consists of all the languages decided by some deterministic Turing machine with an oracle for language A *in polynomial time*. NP^A consists of all the languages decided by some *nondeterministic* Turing machine with an oracle for language A *in polynomial time*.

As in the definiton of P, here "polynomial time" means the run time $\leq Cn^C$ on any input with length n for some absolute constant C.
As in the definition of NP, we requires all branches to halt within polynomial time.

Example. P^{SAT} consists of all the languages that can be decided by some deterministic poly-time Turing Machine with a SAT oracle. This means that $NP \cup coNP \subseteq P^{SAT}$: any $L \in NP \cup coNP$ can be reduced to either SAT or $\overline{SAT}$ (as they are NP-complete and coNP-complete respectively), which the oracle can solve in one timestep.

Designing Algorithms for Oracle Turing Machines

Designing algorithms for P^A is similar to before, except the machine can now use the added power of the oracle. For NP^A, the old paradigm of designing algorithms still applies: we can come up with a certificate that can be quickly verified by a checker. The only difference is that *the checker now has the ability to query the A oracle*. To use this approach, we can simply guess a certificate and then use the checker to verify its correctness using the A oracle. Here is an example of this process in action.

Proposition 3. $\overline{MIN-FORMULA} = \{\varphi \mid \varphi$ *is a boolean formula, such that there exists a shorter boolean formula* φ' *equivalent to* $\varphi\} \in NP^{SAT}$.

Recall from recitation 8 that a boolean formula φ' is equivalent to φ if they are on the same set of variables $x_1, \cdots, x_n$, and they evaluate to the same value for any assignment of $x_1, \cdots, x_n$.

Proof strategy. One possible certificate for the language $\overline{MIN-FORMULA}$ is the shorter boolean formula, φ'. To check whether φ and φ' are equivalent, we can use the SAT oracle to determine whether φ and φ' differ on any assignments. If they do not, then φ and φ' are equivalent, and the certificate is valid.

Proof. Let $\overline{EQUIV-FORMULA} = \{\langle\varphi, \psi\rangle \mid \varphi$ and ψ are not equivalent boolean formulas$\}$. Note that $\overline{EQUIV-FORMULA} \in NP$; a nondeterministic Turing Machine can guess an inequivalent assignment and accept if φ and ψ differ when evaluated with that assignment. Using this language, we can build a nondeterministic Turing machine with a SAT oracle, denoted N^{SAT}, to solve the language $\overline{MIN-FORMULA}$ as follows:

1. Given a boolean formula φ with variables $x_1, \cdots, x_n$, the machine nondeterministically guesses a boolean formula φ' on the same variables such that $|\varphi'| < |\varphi|$.

2. The machine uses the SAT oracle to check whether φ' is equivalent to φ. Specifically, since $\overline{EQUIV - FORMULA} \in NP$, there is a reduction $\overline{EQUIV - FORMULA} \leq_p SAT$. The machine calculates $f(\langle \varphi, \varphi' \rangle)$, and checks if the resulting formula is satisfiable. If not, φ' and φ are equivalent, and the machine accepts.

The last step is to show that the proposed algorithm obeys the required space bounds. Note that the first step is the machine guessing a poly-size formula, so will only take polynomially long. The second step performs the reduction, which takes polynomial time, and writes the result to the oracle tape, again taking polynomial time. Thus, each thread will take polynomial time, showing that $\overline{MIN - FORMULA} \in NP^{SAT}$ as desired.

□

Oracle such that $P^A = NP^A$

The diagonalization method has been a powerful tool for us in the past, so a natural idea is to apply the diagonalization method to the P vs. NP question. However, there exists languages A and B where $P^A \neq NP^A$ and $P^B = NP^B$. This means that any proof strategy that works by simulating one machine with another will likely not be able to solve P vs. NP, as the proof will probably not change when oracles are added to the machines. Here, we show the existence of language B. Language A is shown in the book's Theorem 9.20.

Theorem 4. *There exists a language B such that $P^B = NP^B$.*

Proof. We take B to be $TQBF$. Clearly, $P^{TQBF} \subseteq NP^{TQBF}$ since any deterministic polynomial time Turing machine that has a $TQBF$ oracle can be simulated by a nondeterministic polynomial time Turing machine with a $TQBF$ oracle; the nondeterminstic machine can just not use its nondeterminism. All that remains to be shown is the other inclusion: that $NP^{TQBF} \subseteq P^{TQBF}$.

This can be shown with the following series of containments:

$$NP^{TQBF} \subseteq PSPACE^{TQBF} \subseteq PSPACE \subseteq P^{TQBF}.$$

The first inclusion in the result, $NP^{TQBF} \subseteq PSPACE^{TQBF}$, can be shown by noting that we can enumerate all the branches of a poly-time NTM using a poly-space TM with the same oracle, since each branch can only take poly-space.

To show the next inclusion, $PSPACE^{TQBF} \subseteq PSPACE$, note that $TQBF \in PSPACE$. Then, every time a poly-space TM calls the oracle, a poly-space TM without the oracle can simply solve the $TQBF$ instance itself!

Finally, we want to show that $PSPACE \subseteq P^{TQBF}$. For any $L \in PSPACE$, $L \leq_P TQBF$ since TQBF is $PSPACE$ complete. We can build poly-time deterministic Turing machine M^{TQBF} deciding L as follows:

- On instance x of L, compute $f(x)$ where f is the reduction function from L to $TQBF$.
- Use the $TQBF$ oracle to check if $f(x) \in TQBF$, accepts if true.

The correctness follows from the fact that $x \in L$ if and only if $f(x) \in TQBF$. Therefore, $L \in P^{TQBF}$, and we have that $PSPACE \subseteq P^{TQBF}$, proving our result.

□

Remark. Essentially, the part of the proof that shows $P^{TQBF} \subseteq NP^{TQBF}$ and that $NP^{TQBF} \subseteq PSPACE^{TQBF}$ relies on the fact that the inclusion relationships $P \subseteq NP \subseteq PSPACE$ is preserved when the same oracle is added to the original classes. This is known as the property of *relativization*. The diagonalization proof of time hierarchy in the next section also has this property of relativization. However, Theorem 4 implies that the relationship $P \neq NP$ (if true) does not have this property, and therefore cannot be proved using a diagonalization argument that can relativize.

Hierarchy Theorems

Intuitively, it would make sense that having more resources means that a Turing Machine would be able to solve more problems. In lecture, we show a result called the *space hierarchy theorem*, which states that for any time-constructible function t, there exists a language A that is decidable in $O(t(n))$ space but not $o(t(n))$ space. A similar result holds for time as well!

See the book's Definition 9.8 for the definition of time-constructibility.

Time Hierarchy Theorem

Theorem 5. *For any time constructible function $f : \mathbb{N} \to \mathbb{N}$, there exists a language A that is decidable in $O(f(n))$ time but not $o(f(n)/\log(n))$ time.*

Proof strategy. The idea of this proof is to construct a TM D that runs in $O(f(n))$ time deciding a language A. The key is that, for all TMs that run in time $o(f(n)/\log(n))$ time, D's language will differ from the TM with shorter runtime on at least one input value, via a diagonalization argument. This means that every TM that runs in less time will disagree with D, thus showing that A cannot be decided in less time.

The log factor is annoying, but necessary for this construction. It allows D to accurately count the amount of time that has passed.

Proof. We construct D as follows:

1. Given an input x where the length of x is n, the machine first calculates $f(n)$ by time constructibility and stores $\lceil f(n)/\log(n) \rceil$ in a binary counter. Decrement the counter by 1 before each step used to carry out stage 3. If the counter ever hits 0, reject.

2. Reject if x is not of the form $\langle M \rangle 10^*$.

3. Simulate M on x. Output the opposite as M.

We now show that this simulation runs in $f(n)$ time. Steps 1 and 2 run in time $O(f(n))$. Step 3, however, requires some ingenuity. If we store the counter, M's transition function, and M's tape all in separate parts of D's tape, it may take $O(f(n))$ time to simulate just one step of M, as D's head must move across the entire tape! The trick we play is to instead enlarge the tape alphabet to contain thrice the information as before, thus letting D's tape head read 3 simulated tapes at the same time. The first tape contains M's simulated tape, the second will contain M's transition functions, and the third will contain the binary counter.

If the original tape had an alphabet of $\{0, 1\}$, the new tape would have an alphabet of $\left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$.

At every step of simulating M, D will shift the information on the other two tapes to start at the cells that contain M's head. Because of this shifting, the cost of finding M's next action depends only on M (instead of the input), leading to only a constant overhead. By a similar argument, shifting M's description also incurs only a constant overhead. Thus, to simulate M for $g(n)$ steps, D only needs to run for $O(g(n))$ steps.

We also need to account for the binary counter. The largest value the binary counter will take is $\lceil f(n)/\log(n) \rceil$, which is $O(f(n))$. Thus, the space required to store the binary counter will be $O(\log(f(n)))$, meaning that shifting and updating the binary counter incurs a log factor. Since D simulates M for $\lceil f(n)/\log(n) \rceil$ steps and we incur a constant factor from moving M's description and finding the next action and a log factor from updating and shifting the binary counter, D decides A in $O(f(n))$ time, as desired.

We finally argue correctness. Assume that some TM N decides A in $g(n) \in o(f(n)/\log(n))$ time. Then, D can simulate N for $df(n)/\log(n)$ steps, for some positive constant d. By definition of g, there must exist some n_0 such that for all $n \geq n_0$, $dg(n) < t(n)/\log(n)$, so D's simulation will run to completion as long as the input has length greater than n_0. Consider the input $w = \langle M \rangle 10^{n_0}$, which has length longer than n_0. The simulation in step 3 will thus run to completion, but D will do the opposite of N on w, so N does not decide A!

□