Recitation 12: Oracles and Probabilistic Computation

This week’s recitation covered a lot of interesting but challenging material from the last 3 lectures, which covered intractability, oracles and probabilistic computation. Regarding intractability, we showed that \( EQ_{REX} \) is not in \( P \) (or \( PSPACE \)) by showing that it is \( EXPSPACE \)-Complete. Then, we introduced the concept of oracles, which we will discuss in more depth in the following section.

Oracles and Relativization

Oracles are a powerful tool that allow us get a better understanding of the relationship between some complexity classes as well as the limitations of the diagonalization method we used to show the Hierarchy Theorems. More concretely, oracles will let us show that we cannot prove \( P \) is different from \( NP \) by using diagonalization. All that being said, let’s introduce what an oracle is.

**Definition 1.** An oracle for a language \( A \) is a device that can test membership of a string \( w \) in \( A \). Then, an oracle TM \( M^A \) is a TM with query access to an oracle for \( A \), which means it can use the oracle to test whether a string \( w \in A \) using a single computation step.

Intuitively, an oracle TM \( M^A \) allows us to measure the complexity of a language relative to language \( A \), since any work \( M^A \) does would be additional to the work that is needed to decide \( A \). Based on this we can define the following complexity classes:

**Definition 2.** We define \( P^A \) as the class of languages that are decidable in polynomial time by a deterministic Oracle Turing Machine that uses Oracle \( A \). We similarly define \( NP^A \) for nondeterministic machines.

Now, we can start to reason why diagonalization would not be able to show that \( P \) is different from \( NP \). First, note that any time we have carried out a diagonalization with Turing Machines, this generally involved simulating some Turing Machine \( S \) inside another TM \( R \) and then having \( R \) act in the opposite way as \( S \). If both of these machines had access to some oracle \( A \), then the diagonalization method would...
still work. This means that if we use diagonalization to show a separation between complexity classes $X$ and $Y$, then for any language $A$ we would have that $X^A \neq Y^A$. This turns out to be an issue to proving a separation between $P$ and $NP$ using the diagonalization method, since there exists a language $A$ such that $P^A = NP^A$.

**Theorem 1.** There exists a language $A$ such that $P^A = NP^A$.

**Proof.** We will show that $PTQBF = NP^{TQBF}$. This will be done via a chain of inclusions. First, note that:

$$NP^{TQBF} \subseteq NPSPACE$$

Given that all of the oracle queries could be computed in polyspace. Then, by Savitch’s Theorem we have that:

$$NP^{TQBF} \subseteq NPSPACE \subseteq PSPACE$$

Finally, since $TQBF$ is $PSPACE$-Complete we get that:

$$NP^{TQBF} \subseteq NPSPACE \subseteq PSPACE \subseteq PTQBF$$

Therefore $PTQBF = NP^{TQBF}$. $\square$

Now that we have seen how oracles show us the limitation of the diagonalization method to explore the relationship between $P$ and $NP$, we will explore the relationship between these classes when given a $SAT$-Oracle, namely $P^{SAT}$ and $NP^{SAT}$.

**A Language in $coNP^{SAT}$**

Now, we know that $NP \subseteq P^{SAT}$ and $P^{SAT} \subseteq NP^{SAT}$. We also know that $coNP \subseteq P^{SAT}$ since the oracle can answer non-membership queries for any NP problem. Since we believe that $coNP \neq NP$, this gives evidence that $P^{SAT}$ contains languages outside of $NP$. Something that is not known is whether $P^{SAT}$ and $NP^{SAT}$ are different. In this section, we will discuss a language that is in $coNP^{SAT}$ but is not known to be in any smaller class. This language will be $MIN – FORMULA$, which we will define as follows:

**Definition 3.** First, we say that two boolean formulas with the same set of variables are equivalent if they have the same truth value on all possible inputs. Then, a formula is considered minimal if there are no shorter equivalent formula.

Based on this, we define:

$$MIN – FORMULA = \{ \langle \phi \rangle \mid \phi \text{ is a minimal boolean formula} \}$$
We showed in previous recitation that this language is in \( PSPACE \). We will now show that it is in a smaller complexity class:

**Theorem 2.** \( MIN - FORMULA \) is in \( coNP^{SAT} \).

**Proof.** We will construct a nondeterministic \( SAT \)-Oracle TM \( N^{SAT} \) that decides \( MIN - FORMULA \):

\[
N: \text{On input } \langle \phi \rangle:
\]

1. Nondeterministically guess a formula \( \psi \) shorter than \( \phi \).
2. Query the SAT oracle on input \( (\phi \land \psi) \lor (\overline{\phi} \land \overline{\psi}) \).
3. If the SAT oracle returns ‘yes’, then accept. If the SAT oracle returns ‘no’, then accept.

This algorithm checks whether a formula is not minimal by nondeterministically finding a smaller formula that is equivalent. This equivalence testing uses the SAT oracle and the fact that if the formulae are equivalent, they will always evaluate to the same result.

This example allows us to see how combining nondeterminism and a SAT oracle allows us to give a fairly straightforward algorithm for a language that is not known to be either in \( P^{SAT} \) or \( NP \).

**Probabilistic Turing Machines**

We now move on to a new variation of Turing Machines, probabilistic Turing Machines. These are closely related to Nondeterministic Turing Machines, given that their computation branches, but they differ in that branching in probabilistic TMs happens through random coin flips. We then assign probabilities to each branch based on the number of coin flips on that branch, and based on this we can define the probability of accepting a string as the sum of the probabilities of branches that accept the string. We can use this to define the following complexity classes:

**Definition 4.** \( BPP \) is the class of languages \( A \) such that there exists a probabilistic TM \( M \) which:

- For \( w \in A \): \( \Pr[M \text{ accepts } w] \geq 2/3 \)
- For \( w \notin A \): \( \Pr[M \text{ accepts } w] < 1/3 \)

This means that strings in the language are accepted with high probability, and strings outside of the language are rejected with high probability. Similarly, we define RP as the class of languages \( A \) where there exists a probabilistic TM such that:
• For \( w \in A \): \( \Pr[M \text{ accepts } w] \geq 2/3 \)

• For \( w \notin A \): \( \Pr[M \text{ accepts } w] = 0 \)

Here, strings in the language are accepted with high probability and strings outside the language are never accepted. Thus, if the machine accepts a string, we can be certain that it is in \( A \).

Returning to the comparison with Nondeterministic TMs, it is important to highlight major differences between the classes NP and BPP. First, NP only requires one of the branches of the NTM to accept strings in the language, while BPP requires a majority of the branches to be accepting branches. The other difference is that for languages in NP, strings outside of the language will have no accepting branches in the corresponding TM. On the other hand, the probabilistic TM for a language in BPP may have some accepting branches for strings outside the language since the class only requires that the string is rejected with high probability.

Now, this notion of high and low probabilities is captured in the definition of BPP by using the thresholds 1/3 and 2/3, which is equivalent to saying that the error probability of the TM is 1/3. It is actually sufficient for the probability of accepting strings in the language to be greater than or equal to 1/2 and the probability of accepting strings outside the language to be strictly smaller than 1/2. This is due to the following result:

**Lemma 1** (Amplification Lemma). Let \( \epsilon \) be a constant such that \( 0 < \epsilon < \frac{1}{2} \). Then, if there exists a probabilistic poly-time TM \( M_1 \) with error probability \( \epsilon \), there is an equivalent probabilistic poly-time TM \( M_2 \) which can run repeated simulations of \( M_1 \) to achieve error probability \( 2^{-p(n)} \) for any polynomial \( p(n) \).

This lemma tells us that, as long as our error is less than 1/2, we can achieve the 1/3 error needed for the definition of BPP, and even achieve smaller error probabilities. This can be a very useful tool in problems where we simulate an existing BPP machine and we need to get a tighter bound on our error. One case where this often arises is when we need to run this BPP machine on many different inputs, which makes the overall error probability increase with each simulation. A good exercise to get practice with BPP and the amplification lemma is to show that BPP is closed under union, complement and concatenation.

Now, it is clear that a probabilistic TM can simulate another one by using coin-flip steps to simulate the other’s coin-flip steps. In the next section, we will analyze how probabilistic TMs can be simulated in DTMs and then relate BPP to deterministic complexity classes we are familiar with.
Relating BPP to Deterministic Classes

We will now analyze how a probabilistic TM can be simulated in a deterministic TM, which will be very similar to the way we showed that $NP \subseteq PSPACE$. Doing this will help us get a better intuition for how probabilistic TMs use their coin-flip steps and how this results in branching. After showing this, we will explore how to analyze the number of bits of randomness that a probabilistic TM uses, and how bounding the number of random bits needed to decide a language allows us to simulate a BPP machine in deterministic polynomial time.

We start by proving the following result:

**Theorem 3.** $BPP \subseteq PSPACE$

**Proof.** For any language $L \in BPP$ with probabilistic poly-time TM $M$, we give a poly-space DTM $D$ that decides $L$. Let $p$ be the maximum number of coin-flip steps used in any branch of $M$. Then, since $M$ runs in probabilistic polynomial time, we know that $p$ is at most polynomial in the length of the input. Based on that, the idea is to simulate all of the branches of $M$ by iterating through the possible results of $M$'s coin flips, reusing space, and counting the number of accepting branches.

We now give the DTM $D$:

**D:** On input $w$:

1. Initialize a counter $acc = 0$ and another $tot = 0$
2. For each $r \in \{0, 1\}^p$, reusing space:
   
   (a) Simulate $M$ on input $w$ by using the $i$-th bit of $r$ to pick a branch at the $i$-th coin-flip step.
   
   (b) If $M$ accepts on the current branch, increment $acc \leftarrow acc + 1$.
   
   (c) Increment $tot \leftarrow tot + 1$
3. If $acc/tot \geq 2/3$, then accept. Otherwise, reject.

Note that this machine will only accept if the majority of $M$’s branches are accepting branches, which would mean that $w \in L$ by the definition of $BPP$. Regarding poly-space, simulating $M$’s computation requires polynomial space since $M$ runs in probabilistic polynomial time. Then, since $p$ is polynomial in $n$, the total number of possible branches is $2^p$, but these can be counted using $p$ bits, so our counters are also poly-space.

This containment now allows us to compare $BPP$ to the complexity classes we have studied so far. Something interesting is that while $P$ is clearly in $BPP$, it is not known whether these two classes are different.
On top of this, it is not known whether \( NP \subseteq BPP \) or \( BPP \subset NP \), meaning the classes are incomparable. One main reason why we cannot determine a straightforward relationship between these classes was mentioned earlier. As a reminder, it is that for strings in a language, \( NP \) machines only require one accepting branch while \( BPP \) machines require a majority of branches to be accepting. For strings not in a language, \( BPP \) allows for some accepting branches while \( NP \) requires that there are none. This means that simply simulating a \( BPP \) machine in an \( NP \) machine (or vice versa) would not be sufficient to establish a relationship between the two classes.

To finish off our discussion of \( BPP \), we will look at a two-part problem which first helps us think about how to estimate the number of coin-flip steps needed in a \( BPP \) machine, and then how limiting the number of coin-flip steps a \( BPP \) machine uses allows us to simulate this machine efficiently.

**Example 1.** We define \( BPP(r) \) as the class of \( BPP \) machines that use at most \( O(r) \) coin-flip steps, or bits of randomness, on any given branch of their computation.

(a) Estimate the number of bits of randomness used in the \( BPP \) decider for \( EQ_{ROBP} \)

(b) Show that \( BPP(\log n) \in P \)

**Solution 1.** Here are the solution to the separate parts:

(a) First, recall the \( BPP \) machine for \( EQ_{ROBP} \), which tested whether two read-once branching programs \( B_1, B_2 \) with \( m \) input variables were equivalent by using arithmetization. This machine used its randomness to generate \( m \) random elements from some field \( \mathbb{F}_q \) for some prime \( q \) with \( q > 3m \).

For each of these elements, we need \( O(\log q) = O(\log m) \) random bits, and since we need to generate \( m \) of the field elements, the total number of random bits is approximately \( O(m \log m) \).

(b) We can use the same TM \( D \) that we used to show \( BPP \subseteq PSPACE \), but re-analyze it in the case where only \( O(\log n) \) coin-flip steps occur. First, note that our constant \( p = O(\log n) \), which means that the total number of branches is \( 2^{O(\log n)} = O(n^c) \).

This then means that our counters \( tot \) and \( acc \) are also \( O(n^c) \), so they can be represented using \( O(\log n) \) bits. As a result, we carry out the simulation of \( M \) for \( O(n^c) \) branches, and each branch must halt in polynomial time, so the total runtime of \( D \) is polynomial, which shows that \( BPP(\log n) \subseteq P \).

This analysis of the randomness used in \( BPP \) machines, and how they can be simulated in \( DTM \)s hopefully helps you get a better un-
derstanding of viewing probabilistic computation as branching based on coin flips. Thanks for an amazing semester!