1 Introduction

In this week we study two ideas in complexity theory. The first idea is to understand how the complexity of deciding languages changes as machines are allowed to seek *advice* before they output their decision. This is called as non-uniform computation. The purpose of this is to understand upper bounds on what is normally thought of as computable. The second idea develops the first and aims to build a hierarchy of time complexity classes.

2 Non-uniform Polynomial Time Computation : P^{poly}

Informally speaking, we consider Turing machines that are enabled to receive additional advice during the course of their computation. However, for inputs of length n, the advice, a_n, is allowed to be a single string of length polynomial inn n. The idea behind defining such computations, and hence such Turing machines, is to get a better understanding of the likely separation between the classes P and NP. We now formally define the above class of languages.

Definition 2.1 *Let $L \subseteq \{0,1\}^*$ be a language. We say that $L \in P^{\text{poly}}$ if there is a polynomial time Turing machine M that takes two inputs, a string $x \in \{0,1\}^*$ and an advice $a_n \in \{0,1\}^*$ such that for all w with $|w| = n$, there exists a polynomail $p(n)$ with $|a_n| \leq p(|w|)$ and decides whether $w \in L$ or not.*

In the above definition, notice the following points. The length of the advice string is bound to be a polynomial of the length of the input string itself. This rules out advice strings which take the form of a look up table. A look up table for strings of length n requires an exponential length as there are an exponential number of strings of length n. This property ensures that the class of languages that are in P^{poly} has some interesting members that are otherwise not in P. Secondly, the above definition does not lead to practically constructible machies. So, the class P^{poly} is only of a theoretical interest.

Another important item to notice is that the advice string for inputs of length n is fixed independent of the input. One can think of the advice therefore hardwired into the machine. Another way to think of the machines in the definition is to for a language $L \in ppoly$, imagine a infinite sequence of machines $\{M_i\}_{\{}i \in \mathbb{N}\}$ such that on inputs w of length n, M_n can decide whether $w \in L$. This above view helps us see that NP is different from P^{poly}. For a machine M to decide a language $L \in$ NP, one can see that for an input w such that $w \in L$ there exists a polynomial sized witness $W(w)$ so that M accepts w. So the difference between NP and P^{poly} is that for languages in NP every string $w \in L$ of length n is allowed to have a different advice, whereas for languages in P^{poly}, the advice string is common for all inputs of length n. Another difference between NP and P^{poly} comes from the behaviour of the machine M deciding $L \in$ NP on inputs that are not in L. In the case of a language $L \in$ NP on inputs $w \notin L$, there exists *no* witness that leads to M accept w. The guarantee there is that there is not even a single witness that is useful. On the other hand, for $L \in P^{\text{poly}}$, the only guarantee is that there are *good* advice strings. In fact, it is not even necessary to establish that such an advice string can be found in polynomial time.

Theorem 2.2 *P^{poly} contains non-recursive languages.*

Proof. Let us define an unary language as a language which is a subset of 1^*. We show that there are nonrecursive unary languages and also that every unary language is in P^{poly}.

Let $L_{nr} \subseteq \{0,1\}^*$ be any nonrecursive language. We define a unary nonrecursive language based on L_{NR} as follows. $L_u = \{1^{0(w)} \mid w \in L_{NR}\}$ where $0(w)$ refers to the natural number obtained by treating w as a binary string. It is easy to observe that L_u is a unary language. To show that L_u is non-recursive, we notice that any machine that recognizes L_u can also recognize L_{NR}. Since L_{NR} is known to be nonrecursive, so is L_u.

To show that *every* unary language is in $ppoly$, we just have to exhibit the advice string. Notice that any unary language has at most one string, 1^n, of length n as a member. Therefore, define the advice string as:

$$a_n = \begin{cases} 1 & \text{if } 1^n \in L \\ 0 & \text{otherwise} \end{cases}$$

The advice string a_n is clearly a polynomial in the length of the input. Also, it is not required to show how to compute a_n. Therefore, $L_u \in P^{\text{poly}}$. $\square$

One of the underlying ideas of the above proof is the fact that the existence of short advice strings can be shown. This leads us to the following definition that helps us understand why the class P^{poly} can set apart P and NP.

Definition 2.3 *A language $L \subseteq \{0,1\}^*$ is said to be sparse if there exists a polynomial $p(n)$ such that L has at most $p(n)$ members of length n, i.e., $|L \cap \{0,1\}^n| \leq p(n)$.*

Notice that every unary language is a sparse language. Now, we present a theorem that helps us to study the relationship between NP and P^{poly}. To do so, we need to study the notion of a reduction called the Cook reduction.

Definition 2.4 *A language L_1 Cook-reduces to a language L_2 if there exists a Turing machine running in polynomial time that recoginzes L_1 given* oracle *access to L_2.*

Notice that the machine deciding L_1 can make several queries before deciding whether a given input w is in L_1. An immediate application the above definition is the following lemma.

Lemma 2.5 *If L_1 Cook reduces to L_2 and $L_2 \in$ P then also $L_1 \in$ P.*

Proof. Notice that the definition did not restrict the number of queries to the oracle machine that a machine deciding L_1 can make. However, in this lemma, we observe that if a machine runs in polynomial time then it cannot make more than polynomial many queries to the oracle. The formal proof follows.

As L_1 Cook reduces to L_2, let M be a polynomal time Turing machine that decides L_1 given oracle access to L_2. Since $L_2 \in$ P, we can take that there is a polynomial time machine M_2 that decides L_2. So, the actions of M_2 on a given input can be emulated in polynomial time. Using these, we now design a TM M_1 that can recognize L_1 in polynomial time without any oracle to L_2.

The machine M_1 emulates M until the point when M makes the first query to the oracle. To know the answer to the query, M_1 simply starts emulating M_2. This is repeated so that M_1 knows all the answers to all the queries that M would be making on input w. Since, M runs in polynomial time, the numer of queries and the length of each query would also be polynomial. To simulate M_2 on each of the polynomial queries in still a polynomial time event as M_2 runs in polynomial time. So, the overall runtime of the machine M_1 is a polynomial. $\square$

Our main theorem is now given as follows:

Theorem 2.6 *Every language $L \in NP$ is Cook reducible to a sparse language if and only if NP $\subseteq P^{poly}$.*

Proof. The significance of the above theorem comes from the beleif or conjecture that no language in NP-complete is sparse. Given that the satisfiability problem, or the language of satisfiable Boolean CNF formulas, SAT, is NP-complete, we just have to show that SAT is Cook-reducible to a sparse language if and only if NP $\subseteq P^{\text{poly}}$.

Let SAT be Cook-reducible to a sparse language L. We show that SAT $\in P^{\text{poly}}$ as follows. Since SAT is Cook-reducible to a sparse language, there is a polynomial time oracle machine, M^L that can solve SAT given oracle access to L. The proof is done by exhibiting a polynomial sized advice string for inputs of a given length.

Let the machine M^L run in time $p_{m^l}(n)$ on inputs of length n, where p_{m^l} is a polynomial. It therefore follows that M^L makes at most $p_{m^l}(|w|)$ queries to the oracle deciding L on an input w. We construct the advice string as follows. Let L_n be the set of all strings in L of length at most $p_{m^l}(n)$ where $n = |w|$. Notice that as L is sparse, also $|L_n|$ is a polynomial, say $p_l(n)$. Concatenate all the strings in L_n and set it to a_n. Notice that:

$$|a_n| = \sum_{i=1}^{p_l(i)} p_{m^l}(i) \cdot i = O(p_{m^l}(p_l(n)) \cdot p_l(n)^2)$$

From the above, a_n is of polynomial length. This can be used to show that there is a nonuniform polynomial time machine M with the advice string a_n as defined above. For an input w of length n, M can look up the entire a_n for getting the answer to each of the queries that M^L would have made. The runtime of M is in $O(|a_n| \cdot_l (n))$ which is still a polynomial.

For the other direction, we need to show that if SAT $\in P^{\text{poly}}$ then SAT Cook reduces to some sparse language. Since SAT $\in P^{\text{poly}}$, there is a nonuniform polynomial time machine M_u that given access to advice strings a_n for inputs of length n can decide SAT. The sparse language to which SAT Cook reduces to can be defined as follows. We know that there exists a polynomial $p(n)$ so that $|a_n| = p(n)$. The idea of this proof is to simulate □

3 Gaps in Space and Time

One of the important questions in complexity theory is to study what is not computable within a given resource. If the resource is time, then the question is to see whether there are problems that can be computed in time $t(n)$ but never in time “smaller” than time $t(n)$. An answer to this question indicates whether there is a refined hierarchy of languges that are decidable in a given time. Further, such a hierarchy shows that with more resources, indeed more computation can be achieved. In this section, we show that indeed such a hierarchy exists.

3.1 A Hiearchy Theorem for Space

We start with the following definition.

Definition 3.1 *A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is said to be space constructible if there exists a Turing machine that on input 1^n computes $f(n)$ using space in $O(f(n))$.*

It can be noted that most commonly occuring functions such as polynomials, logarithmic functions, and the like are space constructible. Some aspects of this will be a homework exercise.

The importance of the notion of space constructibility can be understood as follows.

Theorem 3.2 *Let f be any space constructible function. Then, there exists a language L such that L can be decided in $O(f(n))$ space but not in $o(f(n))$ space.*

Proof. The main idea of the proof is to exhibit a language that cannot be decided in $o(f(n))$ time but can be decided in $O(f(n))$ time. Unlike other examples of languages, such languages tend to be artificial in their description. This is due to the proof construction that creates such languages.

The proof builds an algorithm that essentially shows how to create such languages. The algorithm simulates every machine that can decide some language in $o(f(n))$ time. Using a technique similar to diagonalization, one can think of listing each such machine. In that list, each machine corresponds to some language that can be decided in $o(f(n))$ time. So the language we seek should differ from each of these languges in some way. To arrange for that, we make the new language that we build will be made to differ at the string denoting the machine itself. So, if the machine M accepts M as an input, we make the language to not contain M. So, the language will contain M if M rejects M and does not contain M if M accepts M. There are a few details to be worked out in this scheme as the following shows.

What do we do when a string M does not correspond to any machine that decides some language in $o(f(n))$ time? It turns out that it does not matter what we do with such strings. In general, the algorithm on an input w we first check if w corresponds to some Turing machine M. If so, the machine M is simulated on the string M itself. The output is reversed as explained earlier.

What do we do when the machine M does not halt? How to know whether M is entering an infinite loop, how to know if M is using more than $o(f(n))$ space? If M runs in $o(f(n))$ space, then certainly it cannot use more than $2^{o(f(n))}$ time. So one can count the number of steps that M is using on input M. If this count exceeds $2^{f(n)}$ then the string M is kept out of the language.

One final issue is with respect to asymptotics. For instance, for small input strings, even if M runs in $o(f(n))$ space, due to the constants involved, the actual space used may be more than $f(n)$. We have to guard against this problem. For this, instead of simulating M on input M we simulate M whenever the input is of the form $M10^*$. $\square$

The following corollaries follow the above theorem.

Corollary 3.3 *For two functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$, if $f_1(n) \in o(f_2(n))$, then also DSpace$(f_1(n)) \subset$ DSpace$(f_2(n))$.*

So all space is not the same. For instance, DSpace$(n^{c_1}) \subset$ DSpace(n^{c_2}) for any real numnbers $0 \leq c_1 < c_2$.

Corollary 3.4 *NLogSpace $\subset$ PSPACE.*

Let us denote by EXPSPACE the class of problems that can be solved in space 2^{n^k} for a constant k.

Corollary 3.5 *PSPACE $\subset$ EXPSPACE.*

Similar notions can be shown also for time hiearchies. One can define a notion of time constructibility similar to that of Definition **??**. With that definition, the following theorem can be shown.

Theorem 3.6 *Let f be any time constructible function. Then, there exists a language L such that L can be decided in $O(f(n))$ time but not in $o(f(n)/\log(f(n)))$ time.*

The proof of the above theorem is similar to the proof of the space hiearchy theorem. One can wonder then why the additional logarithmic term in the deonominator. Can we hope for a better theorem that is essentially similar to the space hierarchy theorem. Presently not for it is not known how to simulate a $f(n))$ time Turing machine, even assuming a single tape, cannot be simulated with only a constant factor overhead. In fact, nothing is known as to if such a simulation can be done any faster asympototically.

Some corollaries similar to those that follow from the space hierarchy theorem also apply here.

4 The Polynomial Hierarchy

Recall the definition of oracle computations. One interesting question is to study the power of oracles. Specifically, one can seek the nature of languages that are deciable by machines that can recognize a certain set of languages given oracle access to machines that can decide another set of languages. One such question could be: what languages are decidable by polynomial time deterministic Turing machines with oracle access to the class of NP? In this case, there is a Cook reduction from L to some language in NP. A general question is: for two complexity classes C_1 and C_2, what languages are decidable by machines for C_1 given oracle access to C_2. We make the following definition.

Definition 4.1 *Let M be a Turing machine and A be a language. The language $L(M^A)$ is the set of strings accepted by the machine M with oracle access to A.*

Notice the difference to the definition of Cook reduction. In the case of Cook reduction, M has to be a deterministic polynomial time machine. In the above definition, there is no such restriction on M. M can be a nondeterministic or a nonpolynomial or a randomized machine. All such choices make sense as we move towards the following definition.

We can generalize Definition 4.1 to replace the language A by a class of languages to get the following definition.

Definition 4.2 *Let C be a class of languages and M is a Turing machine. Then, M^C denotes the set of languages that are accepted by M given oracle access to some language in C. So,*

$$M^C = \{L(M^A) | A \in C\}$$

We slighlty abuse notation of the above definition by saying that $C_1^{C_2}$, when C_1 and C_2 are classes of languages, refers to the set of languages $\{L(M^A) | L(M^\Phi) \in C_1 \text{ and } A \in C_2)\}$. In the above, we use the notation M^Φ to denote the machine M given no oracle, or empty oracle. Some examples follow.

- NP $\subseteq$ P$^{\text{NP}}$.
- *co*NP $\subseteq$ P$^{\text{NP}}$.

For the first item, the machine M can simply mimic the actions of the oracle. For the second item, the machine simply reverses the actions of the oracle. The above two relations indicate that indeed NP$\cup$*co*NP $\subseteq$ P^{NP}.

To use the above framework, we make the following definition.

Definition 4.3 (Polynomial Hierarchy) *Define a sequence of sets of languages Σ_i as follows. Σ_1 = NP and Σ_{i+1} = NP$^{\Sigma_i}$. Similarly, define a sequence of sets of languages as follows. Π_1 = coNP, and in general $\Pi_i = co - \Sigma_i$. Finally, define Δ_{i+1} = P$^{\Sigma_i}$ for $i \geq 1$.*

We define the polynomial hierarchy, denoted PH as,

$$PH := \cup_i \Sigma_i.$$

Some of the following results that follow the above definition are:

Corollary 4.4 *The following results hold:*

- $\Sigma_i \cup \Pi_i \subseteq \Delta_{i+1} \subseteq \Sigma_{i+1} \cap \Pi_{i+1}$.

- *$\mathrm{P}^{\Sigma_i} = \mathrm{P}^{\Pi_i}$ and $\mathrm{NP}^{\Sigma_i} = \mathrm{NP}^{\Pi_i}$.*

Proof. The first part of the first item is clear from the definitions of Σ, Π, and Δ. For the second part of the first item, notice that $\Delta_{i+1} = \mathrm{P}^{\Sigma_i} \subseteq \mathrm{NP}^{\Sigma_i} = \Sigma^{i+1}$. Also, for any $L \in P^{\Sigma_i}$, it holds that also $\overline{L} \in \mathrm{P}^{\Sigma_i}$ as P^{Σ_i} is closed under complementation. Therefore, $\overline{L} \in P^{\Sigma_i} \subseteq \mathrm{NP}^{\Sigma_i} = \Sigma_{i+1}$ and hence $L \in \Pi_{i+1}$.

For the second item, notice that in each of the cases, the machine in P (respectively NP) has to simply flip the output of the oracle. It can be seen as either flipping the output of the oracle, or considering an oracle for the complement of a language. □

Other important observations that follow from the above definition are given as follows.

Theorem 4.5 *$PH \subseteq$ PSPACE.*

Proof. Using induction. Exercise. □

Finally, the motivation of defining the polynomial hierarchy is captured by the following lemmata.

Lemma 4.6 *If NP $= coNP$ then $PH \subseteq$ NP.*