CSCE-637 Complexity Theory

Lecture #2, October 22, 2020

Lecturer: Professor Jianer Chen

2 The classes BPP and P/poly

We will be mainly focused on *languages*, which are simply sets or decision problems (i.e., the yes-instances of a decision problem make the set of our interest). We assume that all set elements in our discussion are encoded in binary strings. Therefore, the *length* of an element x really refers to its *bitlength*, i.e., the length of its binary encoding, even the string x may be given in non-binary representation in our description. Note that this assumption losses no generality since the length of encoding in any fixed alphabet set is bounded by a constant times the length of binary encoding. As a result, sometimes we will also call an element a *binary string*.

A *Boolean circuit* C_n with n inputs $x_1, \cdots, x_n$ is a directed acyclic graph in which each node has fan-in (i.e., in-degree) either 0 or 2. The nodes of fan-in 0 are *input nodes* and are labeled from the set $\{0, 1, x_1, \overline{x}_1, \cdots, x_n, \overline{x}_n\}$. The nodes of fan-in 2 are called *gates* and are labeled either $\vee$ or $\wedge$. A set of the nodes is designated the *output* nodes. Note that we do not have "negation-gates". Any circuit can be easily converted into this form by De Morgan's Law. The *size* is the number of gates, and the *depth* is the maximum distance from an input to an output. Each node in a circuit of size s has a unique node number of length $O(\log s)$. We assume that circuits are topologically ordered in the sense that the node number of a gate is always larger than the node numbers of its inputs.

A *family* of circuits is a sequence $\mathcal{F} = \{C_n \mid n \geq 1\}$ of circuits, where circuit C_n has n inputs and one output. A family of circuits can be used to accept a language L in $\{0, 1\}^*$ such that for all n, a binary string x of length n is in L if and only if $C_n(x) = 1$, i.e., the output of the circuit C_n has value 1 when the input to C_n is x. The circuit family $\mathcal{F}$ is of *polynomial size* if there is a polynomial $p(n)$ such that the size of the circuit C_n is bounded by $p(n)$ for all n.

Definition 2.1 An *oracle Turing machine* M with an oracle set A is a standard Turing machine plus an *oracle tape* such that when a string y is written on the oracle tape (by M), the machine M can enter a *query state* that asks the membership of y in the oracle set A, and in a single step M gets the answer to the query (i.e., "yes, $y \in A$" or "no, $y \notin A''$).

Note that an oracle Turing machine can be in any of the modes we have encountered in our study of regular Turing machines. Thus, an oracle Turing machine can be deterministic, nondeterministic, or probabilistic. The time and space complexity of an oracle Turing machine are those spent by the oracle Turing machine, but not including those used in oracle queries. Thus, the space used in the oracle tape does not count as the space used by the oracle Turing machine, and the time spent on each query counts as a single step of the oracle Turing machine (however, the time spent by the oracle Turing machine on writing query strings in the oracle tape does count as the time for the oracle Turing machine). These conventions allow us to define the time and space complexity of oracle Turing machines.

Definition 2.2 A language L is *Turing reducible* to another language A, written as $L \leq_T^p A$, if L is accepted by a deterministic polynomial-time oracle Turing machine that uses A as its oracle set.

Turing reducibility was originally used by Steven Cook in his famous proof that the SATISFIABILITY problem is NP-complete (under Turing reducibility). Thus, Turing reducibility is also called *Cook-reducibility*. Under Turing reducibility, we can similarly define NP-hardness and NP-completeness: a language L is *NP-hard* under Turing reducibility if every language in NP is Turing reducible to L, and is *NP-complete* under Turing reducibility if in addition L is in NP. It is easy to see that if a language L is Karp-reducible to a language A (i.e., under the polynomial-time many-one reducibility), then L is also Turing reducible to A. Therefore, any NP-hard (resp. NP-complete) problem under Karp-reducibility, as we studied in *Analysis of Algorithms*, remains NP-hard (resp. NP-complete) under

Turing reducibility. On the other hand, whether there are languages L and A such that L is Turing reducible to A but not Karp-reducible to A has remained as a well-known open problem in complexity theory.

Definition 2.3 A language S in $\{0,1\}^*$ is *sparse* if there is a polynomial $p(n)$ such that for all n, the number of elements of length n in S is bounded by $p(n)$.

Note that in the above definition, this is equivalent to define a sparse set S by restricting the number of elements of length less than or equal to n in S to be bounded by a fixed polynomial of n.

Now we give our final definition in this section.

Definition 2.4 A language L is in the class P/`poly` if there is a function $h(n)$ whose length $|h(n)|$ is bounded by a polynomial of n, such that there is a deterministic polynomial-time Turing machine M_d that on input $\langle x, h(|x|)\rangle$ decides if $x \in L$.

We remark that the existence of the Turing machine M_d in the above definition does *not* mean that the language $\{\langle x, h(|x|)\rangle \mid x \in L\}$ is in P. In fact, the function $h(n)$ can be very complicated and checking whether a given binary string is a value of $h(n)$ for some n can be extremely difficult.

As an example, we show that the important complexity class BPP is a subclass of P/`poly`. Let L be a language in BPP. By the definition, there is a probabilistic polynomial-time Turing machine M that on any input x makes a correct decision (i.e., $x \in L$ or $x \notin L$) with a probability at least $\frac{1}{2} + \epsilon$, where $\epsilon > 0$ is a constant. Without loss of generality, we can assume $\epsilon < \frac{1}{2}$ (otherwise M is a deterministic Turing machine, and L is in P).

We first construct a new probabilistic Turing machine M_{2t} that accepts L: on an input x, the Turing machine M_{2t} simulates the machine M on the input x $2t$ times, then takes the majority outcomes of the $2t$ simulations as its decision. We analysis the probability that the machine M_{2t} makes mistakes.

Consider an arbitrary input x of length n. By the definition, the Turing machine M on the input x makes a correct decision with a probability equal to $1 + \epsilon_x$, where $\frac{1}{2} \geq \epsilon_x \geq \epsilon$ is a fixed constant for the input x.

When the decision of M_{2t} on x is incorrect, then only i of the $2t$ simulations of M on x are correct, where $i \leq t$. Fix the positions of these i incorrect simulations. Then the probability that exactly these i simulations of M on x are correct (and the rest $2t - i$ simulations are all incorrect) is $\left(\frac{1}{2} + \epsilon_x\right)^i \left(\frac{1}{2} - \epsilon_x\right)^{2t-i}$. Since there are $\binom{2t}{i}$ ways to pick i positions in the $2t$ simulations, the probability that exactly i simulations, on any positions, of M on x are correct and the rest $2t - i$ simulations are all incorrect is $\binom{2t}{i} \left(\frac{1}{2} + \epsilon_x\right)^i \left(\frac{1}{2} - \epsilon_x\right)^{2t-i}$. Since when $i \leq t$, the machine M_{2t} outputs an incorrect conclusion, the probability that the machine M_{2t} gives an incorrect conclusion is

$$\sum_{i=0}^{t} \binom{2t}{i} \left(\frac{1}{2} + \epsilon_x\right)^i \left(\frac{1}{2} - \epsilon_x\right)^{2t-i}.$$

Now play some simple mathematics.

$$
\begin{aligned}
& \sum_{i=0}^{t} \binom{2t}{i} \left(\frac{1}{2}+\epsilon_x\right)^i \left(\frac{1}{2}-\epsilon_x\right)^{2t-i} \\
= & \sum_{i=0}^{t} \binom{2t}{i} \left(\frac{1}{2}+\epsilon_x\right)^t \left(\frac{1}{2}-\epsilon_x\right)^t \left(\frac{1/2-\epsilon_x}{1/2+\epsilon_x}\right)^{t-i} \\
\le & \sum_{i=0}^{t} \binom{2t}{i} \left(\frac{1}{4}-\epsilon_x^2\right)^t \\
= & \left(\frac{1}{4}-\epsilon_x^2\right)^t \sum_{i=0}^{t} \binom{2t}{i} \\
\le & \left(\frac{1}{4}-\epsilon_x^2\right)^t \frac{2^{2t}}{2} \\
\le & \left(1-(2\epsilon_x)^2\right)^t
\end{aligned}
$$

Since $\frac{1}{2} \ge \epsilon_x \ge \epsilon > 0$, $1-(2\epsilon_x)^2$ is a constant such that satisfies $0 \le 1-(2\epsilon_x)^2 \le 1-(2\epsilon)^2 = c < 1$, where $c = 1-(2\epsilon)^2$ is a constant that is independent of x . Thus, on the input x, the probabilistic Turing machine M_{2t} makes mistake with a probability bounded by $(1-(2\epsilon_x)^2)^t \le c^t$. Now for input x of length n, let $t = \lceil (n+1)/\log(1//c) \rceil$, then the probability that M_{2t} makes mistake on x is bounded by $1/2^{n+1}$. Note that this probability error bound $1/2^{n+1}$ holds true for all inputs of length n. Since the Turing machine M is polynomial-time bounded, the Turing machine M_{2t} also has its running time bounded by a polynomial $q(n)$ of n on all inputs of length n. Thus, the computation of M_{2t} on an input x of length n can be depicted as a complete binary tree $\mathcal{T}$ of $2^{q(n)}$ leaves such that at most $2^{q(n)}/2^{n+1}$ of the leaves in $\mathcal{T}$ correspond to the computations that conclude incorrectly. Since there are totally 2^n inputs of length n (recall that all elements are encoded in binary), there are at most $2^n \cdot 2^{q(n)}/2^{n+1} = 2^{q(n)}/2$ leaves in $\mathcal{T}$ that would conclude incorrectly on some input of length n. In other words, at least one half of the leaves in $\mathcal{T}$ will always conclude correctly on all inputs of length n. Let l be any one of these always-correct leaves. Then by following the computational path P_n from the root to l in the tree $\mathcal{T}$, we will always reach a correct conclusion on any input of length n.

Note that the length of the computational path P_n is $q(n)$. Thus, P_n can be encoded into a binary string $h(n) = \text{bin}(P_n)$, whose length $|h(n)|$ is bounded by a polynomial of n. Now using the probabilistic Turing machine M_{2t} and the computation path P_n, we can construct a deterministic Turing machine M_d as follows. On an input $\langle x, h(|x|)\rangle$, the Turing machine M_d simulates the probabilistic Turing machine M_{2t} on x. However, when the machine M_{2t} tries to make a randomized branch, the machine M_d consults the path P_n to decide which branch to go. The Turing machine M_d concludes with $x \in L$ if and only if the computational path P_n of M on x leads to a leave that accepts x. By the above discussion, the computational path P_n of M on x always leads to a correct conclusion of M. Thus, the Turing machine M_d on input $\langle x, h(|x|)\rangle$ always concludes correctly on the membership of x in L. Moreover, the Turing machine M_d is obviously deterministic and runs in polynomial time.

Now the function $h(n) = \text{bin}(P_n)$ whose length is bounded by a polynomial of n and the deterministic polynomial-time Turing machine M_d show that the language L is in P/`poly`. Since L is an arbitrary language in BPP, this proves the following theorem.

Theorem 2.1 BPP ⊆ P/`poly`.

The complexity class BPP has drawn very significant attention in the research in computer science, from both theoretical and practical research in computation. We will study the relationship between BPP and the *advice model*. In the following, we first give several equivalent definitions of the class P/`poly`

Theorem 2.2 *Let L be a language. The following statements are equivalent:*

(1) *L is accepted by a polynomial-size circuit family;*
(2) *$L \in \mathrm{P}/\mathtt{poly}$;*
(3) *L is Turing reducible to a sparse set A.*

PROOF. We first prove that (1) and (2) are equivalent, then show that (2) and (3) are equivalent.

(1) $\iff$ (2): Suppose that L is accepted by a polynomial-size circuit family $\{C_n \mid n \geq 0\}$, where for all n, the size of the circuit C_n is bounded by a polynomial of n. Then, the length of the binary encoding $h(n) = \mathrm{bin}(C_n)$ of the circuit C_n is also bounded by a polynomial of n. Now it is easy to construct a deterministic polynomial-time Turing machine M_d such that on the input $\langle x, h(|x|)\rangle$, where $h(|x|) = \mathrm{bin}(C_{|x|})$, M_d determines whether the circuit $C_{|x|}$ accepts the input x, thus, determines whether $x \in L$. The function $h(n)$ and the Turing machine M_d show that $L \in \mathrm{P}/\mathtt{poly}$.

Conversely, suppose that $L \in \mathrm{P}/\mathtt{poly}$. Then there is a function $h(n)$ whose length $|h(n)|$ is bounded by a polynomial of n, such that there is a deterministic polynomial-time Turing machine M_d that on input $\langle x, h(|x|)\rangle$ determines whether $x \in L$. By a well-known result in complexity theory [?], the language accepted by M_d is accepted by a (polynomial-time uniform) circuit family $\mathcal{F} = \{C_m \mid m \geq 0\}$, where for all m, the size of the circuit C_m is bounded by a fixed polynomial of m. Since $|h(n)|$ is bounded by a polynomial of n, the length m of the pair $\langle x, h(|x|)\rangle$ is bounded by a polynomial of $|x|$. Therefore, the size of the circuit C_m that is for all the inputs $\langle x, h(|x|)\rangle$ of length $m = n + |h(n)|$, where $n = |x|$, is bounded by a polynomial of n. Note that for all strings x of length n, the corresponding pair $\langle x, h(|x|)\rangle$ have the same length thus are handled by the same circuit C_m in $\mathcal{F}$. Thus, the pair $\langle x, h(|x|)\rangle$ is accepted by the corresponding circuit C_m if and only if $x \in L$. Now if we assign the value $h(|x|)$ to the corresponding inputs in C_m, we get a circuit C'_n with n inputs such that C'_n accepts x of length n if and only if $x \in L$. Therefore, the circuit family $\mathcal{F}' = \{C'_n \mid n \geq 0\}$ accepts the language L. As we explained above, the size of the circuit C'_n, which is roughly the same as that of the corresponding circuit C_m in $\mathcal{F}$, is bounded by a polynomial of n. This proves that the language L is accepted by a polynomial-size circuit family.

This completes the proof for (1) $\iff$ (2).

(2) $\iff$ (3): Suppose that $L \in \mathrm{P}/\mathtt{poly}$. Thus, there is a function $h(n)$ whose length $|h(n)|$ is bounded by a polynomial $p(n)$ of n and there is a deterministic polynomial-time Turing machine M_d that on input $\langle x, h(|x|)\rangle$ determines whether $x \in L$. Now define a set B by

$$B = \{1^n\#1^{p^2(n)}\#b \mid b \text{ is a prefix of } h(n)\}.$$

Note that the set B is (very) sparse: for each length $m = 2+n+p^2(n)+l$, where $0 \leq l \leq p(n)$, there is at most one element $1^n\#1^{p^2(n)}\#b$ of length m in B, where b is of length l and is a prefix of $h(n)$.[1] Now we construct a deterministic oracle Turing machine M_2 that uses the oracle B and accepts the language L, as follows: on an input x of length n, M_2 starts with the string $s_0 = 1^n\#1^{p^2(n)}\#$. Inductively, assume that M_2 has constructed a string $s_i = 1^n\#1^{p^2(n)}\#b_i$ in B, where b_i is of length i and is a prefix of $h(n)$. Then M_2 queries the oracle B to find a symbol σ such that $s_i\sigma = 1^n\#1^{p^2(n)}\#b_i\sigma$ is in B (so $b_i\sigma$ is a prefix of $h(n)$), then let $s_{i+1} = s_i\sigma$ (if no such a symbol σ exists, then $b_i = h(n)$). Since $|h(n)|$ is bounded by the polynomial $p(n)$, the oracle machine M_2 can construct the function $h(n)$ in polynomial time, then call the Turing machine M_d on $\langle x, h(n)\rangle$ to decide if x is in L. The machine M_d runs in time polynomial in the length of $\langle x, h(n)\rangle$, which is bounded by a polynomial of $|x|$. As a result, the oracle Turing machine M_2 uses the sparse oracle B, runs in time polynomial in n, and accepts the language L. This proves that the language L is Turing reducible to the sparse set B.

Conversely, suppose that a language L is accepted by a deterministic oracle Turing machine M_1 with a sparse oracle set B such that the running time of M_1 is bounded by a polynomial $p_1(n)$ of n and for all m, the number of elements of length m in the oracle set B is bounded by a polynomial $p_2(m)$ of m. For each $m \geq 0$, let the elements of length m in B be $x_{m,1}$, $x_{m,2}$, $\ldots$, $x_{m,t}$, where $t \leq p_2(m)$. Define $s_m = x_{m,1}\#x_{m,2}\#\cdots\#x_{m,t}$, and let $h(n) = s_1\&s_2\&\cdots\&s_{p_1(n)}$. Note that for each

[1] Without loss of generality (otherwise we pick a larger polynomial), we can assume that the polynomial $p(n)$ satisfies the condition $p^2(n+1) + (n+1) + 2 > p^2(n) + n + 2 + p(n)$. Thus, for two different lengths n and h, two strings of the forms $1^n\#1^{p^2(n)}\#b$ and $1^h\#1^{p^2(h)}\#b'$ in B, where $|b| \leq p(n)$ and $|b'| \leq p(h)$, cannot have the same length.

m, $|s_m| = O(mp_2(m))$, so $|h(n)| = O(p_1(n) \cdot p_1(n)p_2(p_1(n))) = O(p_3(n))$, where $p_3(n)$ is a polynomial of n. Now we construct a another deterministic Turing machine M_2 that uses no oracle. On an input $\langle x, h(n)\rangle$, where $n = |x|$ and $|h(n)| = O(p_3(n))$, M_2 simulates the oracle Turing machine M_1 on the input x step by step except that when M_1 makes a query y to the oracle B, M_2 instead searches y in the string $h(n)$ to decide if $y \in B$. Since M_1 runs in time $p_1(n)$ on the input x of length n, the length of the query string y cannot be larger than $p_1(n)$ while $h(n)$ contains all s_i upto $i = p_1(n)$. Thus, we can always correctly decide if $y \in B$ by searching $h(n)$ in time $O(|h(n)|)$. Thus a query step of M_1 can be simulated by M_2 in time $O(|h(n)|)$. Since M_1 runs in time bounded by $p_1(n)$, the machine M_2 on input $\langle x, f(n)\rangle$ runs in time $O(p_1(n) \cdot |h(n)|)$ and decides if $x \in L$. Since $p_1(n) \leq |h(n)|$, and $|h(n)| \leq p_3(n)$, the running time $O(p_1(n) \cdot |h(n)|)$ of M_2 is bounded by a polynomial of n. Now the function $h(n)$ and the deterministic Turing machine M_d show that $L \in \mathrm{P}/\mathtt{poly}$.

This completes the proof for (1) $\Longleftrightarrow$ (2), thus completes the proof of the theorem. $\square$

References

[1] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H.Freeman and Company, New York, 1979.

[2] N. Immerman, Nondeterministic space is closed under complementation, *SIAM Journal on Computing* 17, 935-938 (1988).

[3] C. Papadimitriou, *Computational Complexity*, Addison Wesley, Reading, Mass., (1994).