CSCE-637 Complexity Theory

Lecture #4, November 5, 2020

Lecturer: Professor Jianer Chen

4 The Baker-Gill-Solovay Theorem

The central problem in complexity theory is the P versus NP problem. Recall that P is the class of languages that are accepted by deterministic polynomial-time Turing machines, while NP is the class of languages that are accepted by nondeterministic polynomial-time Turing machines. Therefore, the relationship between P and NP asks whether nondeterministic polynomial-time Turing machines are more powerful than deterministic polynomial-time Turing machines. The current status of research in complexity theory seems still pretty far from a solution to this question. We actually do not seem to have powerful enough techniques that enable us to derive precise relations between these two models.

We have seen Turing machines with oracles. Therefore, it is natural to ask when equipped with oracles, what can we say about polynomial-time computations that are deterministic and nondeterministic, respectively. This section gives interesting results on this question.

Let us first give formal definitions. Let A be an language (the oracle set). We say that a language L is in P^A if there is a deterministic polynomial-time oracle Turing machine that uses A as its oracle and accepts the language L. Similarly, a language L' is in NP^A if there is a nondeterministic polynomial-time oracle Turing machine that uses A as its oracle and accepts the language L'.

Clearly, $\mathrm{P}^A \subseteq \mathrm{NP}^A$ for all oracle sets A, since a deterministic polynomial-time oracle Turing machine is a special case of a nondeterministic polynomial-time oracle Turing machine.

Define by EXP the class of languages that are accepted by deterministic Turing machines (with no oracle) in time $O(2^{n^c})$, where c is any constant.

Theorem 4.1 *There is an oracle set A such that $\mathrm{P}^A \subseteq \mathrm{NP}^A$.*

PROOF. The observation given before the theorem shows that we only need to find an oracle set A such that $\mathrm{NP}^A \subseteq \mathrm{P}^A$.

The idea is quite simple. Although deterministic polynomial-time computation and nondeterministic polynomial-time computation *may* be different in power, if we add a "super-powerful" oracle to both models, the difference will be covered up. For this, let us consider the following oracle set:

$$A = \{(M, y, 1^m) \mid \text{the deterministic Turing machine } M \text{ accepts } y \text{ in } 2^m \text{ steps}\},$$

where 1^m devotes a sequence of m 1's. We remark that the oracle set A, as a language, is actually EXP-complete (i.e., every language in EXP can be reduced to A by a Karp-reduction, i.e., a polynomial-time many-one reduction). Therefore, the oracle set A catches the complexity of the class EXP, which is at least as high as that of nondeterministic polynomial-time Turing machines.

Now consider an arbitrary language L in NP^A. By definition, L is accepted by a nondeterministic oracle Turing machine M_1 that uses the oracle set A and runs in time $p(n)$, where $p(n)$ is a fixed polynomial of n. Consider the algorithm in Figure 1.

By the definition, M_1^A accepts x (i.e., $x \in L$) if and only if there is a computational path P of M_1 on the input x along which M_1 accepts x. For each query $(M, y, 1^m)$ made by M_1 along the path P, the machine M_2 runs the machine M on y up to 2^m steps to determine directly whether $(M, y, 1^m) \in A$ (note that M_2 can verify easily whether M is a deterministic Turing machine). Therefore, the simulation by M_2 on the path P correctly checks whether the oracle Turing machine M_1 using oracle A accepts the input x along the path P.

This verifies that the Turing machine M_2 accepts the language L. Now consider the running time of M_2. Since the nondeterministic Turing machine M_1 runs in time $p(n)$, there are at most $2^{p(n)}$ computational paths when M_1 runs on the input x, where $|x| = n$. Thus, the number of times the loop in step 1 is executed is bounded by $2^{p(n)}$. On each computational path P, M_1 runs in at most $p(n)$

Algorithm 1 $M_2(x)$
\\ M_2 accepts the language L
1. **for** each computational path P of M_1 on x **do**
1.1 simulate M_1 on x along the path P;
\\ during the simulation, for each query $(M, y, 1^m)$ made by M_1 on the
\\ path P, M_2 simulates M on y for 2^m steps to decide if $(M, y, 1^m) \in A$;
1.2 **if** M_1 accepts x on the path P **then** accept x and stop;
2. reject x.

Figure 1: A deterministic algorithm that simulates M_1^A on input x

steps. However, a step in which M_1 makes an oracle query $(M, y, 1^m)$ on A now becomes a simulation by M_2 on M for at most 2^m steps. Since the path P runs in at most $p(n)$ steps, we have $m \leq p(n)$ (i.e., M_1 cannot write down more than $p(n)$ 1's if it runs in $p(n)$ steps). Therefore, a query step of M_1 now is simulated by at most $O(2^{p(n)})$ steps of the Turing machine M_2. In conclusion, the computational path P of M_1 can be simulated by the Turing machine M_2 in no more than $O(2^{p(n)}p(n))$ steps. Combining this with the above discussion, we conclude that the running time of the Turing machine M_2 is bounded by $2^{p(n)} \cdot O(2^{p(n)}p(n)) = O(2^{2p(n)}p(n))$. Let $q(n)$ be a polynomial of n such that $2^{q(n)}$ is an upper bound on the running time of the Turing machine M_2, $q(n) = O(p(n) \log p(n))$.

Now consider a deterministic polynomial-time oracle Turing machine M_3 that uses the oracle set A. On an input x of length n, the Turing machine M_3 makes a query $(M_2, x, 1^{q(n)})$ on the oracle set A, and accepts x if and only if $(M_2, x, 1^{q(n)}) \in A$. Since x is in L if and only the deterministic Turing machine M_2 accepts x in no more than $2^{q(n)}$ steps, the oracle Turing machine M_3 using the oracle A accepts exactly the language L. Moreover, M_3 runs in polynomial time since $q(n)$ is a polynomial of $n = |x|$. This proves that $L \in \mathrm{P}^A$. Since L is an arbitrary language in NP^A, this proves $\mathrm{NP}^A \subseteq \mathrm{P}^A$, thus, completes the proof that for the oracle set A, $\mathrm{P}^A = \mathrm{NP}^A$. □

We now show that there is an oracle set B that separates P and NP, i.e., $\mathrm{P}^B \neq \mathrm{NP}^B$. To do this, we somehow need to make an oracle set B for which nondeterministic Turing machines have obvious and *provable* advantages. We start with the following lemma.

Lemma 4.2 *For any oracle B, the language $U_B = \{1^n \mid B$ contains a string of length $n\}$ is in NP^B.*

PROOF. A nondeterministic linear-time oracle Turing machine M that uses the oracle B can recognize U_B as follows. On the input 1^n of length n, M nondeterministically guesses a string x of length n, queries it on the oracle set B, and accepts if and only if $x \in B$. In particular, if $1^n \in U_B$, then one of the computational paths of M will guess a correct string x of length n in B so that computational path will accept 1^n. On the other hand, if $1^n \notin B$, then all strings x of length n guessed by M are not in B so all computational paths of M reject 1^n. This shows that the nondeterministic (and linear-time) Turing machine M accepts the language U_B, and $U_B \in \mathrm{NP}^B$. □

By Lemma 4.2, we only need to construct an oracle set B such that $U_B \notin \mathrm{P}^B$. This is done by the technique of diagonalization that has been standard and useful in the study of complexity theory and computability theory. We need some preparation before we formally describe the technique.

Recall that Turing machines, including oracle Turing machines, are given by their transition functions. Therefore, each Turing machine can be described by a finite sequence of symbols. Also, we assume that each Turing machine M can be described by *infinitely* many finite sequences. For this, we can fix a special "ending" symbol so that a valid representation of M followed by the ending symbol that is followed by any finite symbol sequence is also treated as a valid representation of the Turing machine M. Now assuming an universal encoding, each valid representation of a Turing machine can be given as a finite binary string. Therefore, all Turing machines can be listed as an infinite sequence of finite binary strings and for each Turing machine M, there are infinitely many finite binary strings in the sequence that are valid representations of M. In particular, all deterministic oracle Turing machines

can be listed as an infinite sequence

$$M_1, M_2, \ldots, M_i, \ldots, \tag{1}$$

in which for each deterministic oracle Turing machine M, there are infinite many indices i_j, $j \geq 0$, such that all M_{i_j} are valid representations of M.

Our construction of the oracle set B that makes $U_B \notin \mathrm{P}^B$ is based on the sequence (1), for which the set B is constructed stage by stage based on the Turing machines M_i, in the order of the sequence. The general idea is that for each i, we exclude the possibility that the oracle Turing machine M_i accepts the language U_B. To do this, for each i, we find an input length n_i and enforce M_i to make a "wrong" decision on the input 1^{n_i}, i.e., if M_i accepts 1^{n_i} then we exclude all strings of length n_i from B (thus, $1^{n_i} \notin U_B$), while if M_i rejects 1^{n_i}, then we make B to include some string of length n_i (thus, $1^{n_i} \in U_B$). To make sure that the running time of the Turing machine M_i will cover all possible polynomials of n_i, we run M_i on input 1^{n_i} for up to 2^{n_i-1} steps.

Note that on the input 1^{n_i} of length n_i, the machine M_i may query strings y of length larger than n_i for which the membership of y in B has not been decided yet. We must handle this carefully, so that when the Turing machine M_i uses the oracle B, it only uses the subset of B that has been constructed at the current stage.

The set B is constructed in stage. We devote by B_i the subset of B that is constructed at stage i. At each stage i, the set B_i is a finite set and contains strings of length bounded by n_i. We also enforce that $B_{i-1} \subseteq B_i$ and that $n_{i-1} < n_i$, for all i. Initially, we have $B = \emptyset$. Inductively, assume that we have constructed the set B_{i-1}, such that for each oracle Turing machine M_k with $k \leq i-1$, there is an input length n_k, $n_k \leq n_{i-1}$, such that using the oracle set B_{i-1}, M_k accepts 1^{n_k} if and only if B_{i-1} contains no string of length n_k, i.e., if and only if $1^{n_k} \notin U_{B_{i-1}}$. For each $k \leq i-1$, since the running time of the Turing machine M_k is limited to 2^{n_k-1} steps, the number of strings queried by M_k on the input 1^{n_k} is a finite number.

Now pick an integer $n_i > n_{i-1}$ that is also larger than the length of any string queried by an oracle Turing machine M_k on input 1^{n_k} with $k < i$, no matter whether the string is in B_{i-1} or not (recall that M_k on the input 1^{n_k} queries only finite number of strings on the oracle). We will consider if we want to add a string of length n_i to B_{i-1} to make B_i. Since n_i is larger than the length of any string queried by the oracle Turing machine M_k on input 1^{n_k} for any $k < i$, adding a string of length n_i to the oracle will not change the computational result of the Turing machine M_k on the input 1^{n_k}.

We simulate the oracle Turing machine M_i on the input 1^{n_i} for up to 2^{n_i-1} steps. If M_i queries a string y of length smaller than n_i on the oracle, then we let the answer to the query be consistent with $y \in B_{i-1}$. If M_i queries a string z of length at least n_i on the oracle, then we let the answer to the query be NO. Using these answers to the queries, if the Turing machine M_i accepts 1^{n_i} within 2^{n_i-1} steps, then we add no string of length n_i to the oracle set B so that the oracle B contains no string of length n_i (note that since $n_{i-1} < n_i$ for all i, neither a previous nor a later stage can add a string of length n_i to B). If the Turing machine M_i does not accept 1^{n_i} in 2^{n_i-1} steps (i.e., M_i either rejects 1^{n_i} or has not reached a decision yet), then we add a string of length n_i to the oracle set B. Note that since we only simulate M_i for at most 2^{n_i-1} steps, M_i can make at most 2^{n_i-1} queries. Therefore, on at most 2^{n_i-1} strings of length n_i, the Turing machine M_i on the input 1^{n_i} by our construction has answered NO so these strings should be excluded from the set B. Since there are 2^{n_i} strings of length n_i, there must be some strings of length n_i that are not queried by M_i on the input 1^{n_k}. Thus, including one b of these un-queried string in B will not change the computational result of M_i on the input 1^{n_i}. This completes the construction of the subset B_i. By the construction, M_i accepts 1^{n_i} within 2^{n_i-1} steps if and only if B_i contains no strings of length n_i. Moreover, since no Turing machine M_k on the input 1^{n_k} with $k < i$ queries a string of length n_i, adding the string b of length n_i to B will not change the computational result of M_k on 1^{n_k}. Therefore, we still maintain the condition that for all $k \leq i$, the oracle Turing machine M_k accepts 1^{n_k} if and only if B contains no string of length n_k.

Note that for each i, B_i is a finite set. However, we repeat the above construction for all deterministic oracle Turing machines M_i, the final set B, which is the limit of B_i, or the union of all B_i, is an infinite set. Moreover, recall that each deterministic oracle Turing machine has infinitely many valid representations in the sequence (1), for each deterministic oracle Turing machine M, there are actually

infinitely many inputs of the form 1^{n_i} such that M accepts the language U_B if and only if B contains no string of length n_i.

Theorem 4.3 *For the oracle set B constructed above,* $\mathrm{P}^B \neq \mathrm{NP}^B$.

PROOF. By Lemma 4.2, $U_B \in \mathrm{NP}^B$. Thus, it suffices to prove that $U_B \notin \mathrm{P}^B$.

Assume the contrary that $U_B \in \mathrm{P}^B$. Thus, U_B is accepted by a deterministic oracle Turing machine M that uses the oracle set B and runs in time $p(n)$, where $p(n)$ is a polynomial of n. Let i be an index such that the Turing machine M_i is a valid representation of M and $2^{n_i-1} > p(n_i)$ (recall that the deterministic oracle Turing machine M has infinitely many valid representations in the sequence (1) and that $n_{i-1} < n_i$ for all i. Thus, we can pick a sufficiently large index i such that n_i is large enough to ensure the exponential function 2^{n_i-1} being larger than the fixed polynomial function $p(n_i)$). Since the running time of M_i is bounded by $p(n_i)$, we can assume without loss of generality that on any input of length n_i, the Turing machine M_i runs in no more than $p(n_i)$ steps and stops with a decision (yes or no). Since $2^{n_i-1} > p(n_i)$, the Turing machine M_i accepts an input of length n_i in no more than $p(n_i)$ steps if and only if M_i accepts an input of length n_i in no more than 2^{n_i-1} steps.

By the construction of the oracle set B, the oracle Turing machine M_i accepts 1^{n_i} in $p(n_i)$ steps, which is equivalent to that the oracle Turing machine M_i accepts 1^{n_i} in 2^{n_i-1} steps, if and only if B contains no string of length n_i. Thus, M_i accepts 1^{n_i} if and only if 1^{n_i} is not in U_B, contradicting the assumption that M_i accepts U_B. □

Theorems 4.1 and 4.3 were first discovered by Baker, Gill and Solovey [3], which once were regarded as an evidence for the possibility of an independent mathematical system for the P versus NP problem (i.e., no current mathematical system would lead to a confirmed answer to the relationship between P and NP), based on the following reasonning. If there were a proof for P = NP, then one could probably be able to "relativize" the proof and allow the proof to also handle the query steps, which would give a proof for $\mathrm{P}^B = \mathrm{NP}^B$ for any oracle set B. But this would contradict Theorem 4.3. Similarly, relativizing a proof for P ≠ NP would show $\mathrm{P}^A \neq \mathrm{NP}^A$ for all oracle sets A, which would contradict Theorem 4.1. More formally, we say that a proof on relationships among complexity classes without oracles is *relativizable* if the proof can be migrated to lead to the relationships among complexity classes with oracles. Therefore, Theorems 4.1 and 4.3 exclude the possibilities of relativizable proofs for the relationship between P and NP. However, later research found out that there are many proofs for relationships among complexity classes are not relativizable. In particular, the possibility of non-relativizable proofs that give firm answers to the P versus NP problem cannot be eliminated.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] N. Alon, R. Yuster, and U. Zwick, Color-coding, *Journal of the ACM 42*, pp. 844-856, (1995).

[3] T. Baker, J. Gill, and R. Solvay, Relativizations of the P =? NP question, *SIAM Journal on Computing 4*, pp. 431-442, (1975).

[4] H.Bodlaender, R. Downey, M. Fellows, and D. Hermelin, On problems without polynomial kernels, *Journal of Computer and System Sciences 75*, pp. 423-434, (2009).

[5] J. Chen, I. A. Kanj, and G. Xia, Improved upper bounds for vertex cover, *Theoretical Computer Science 411*, pp. 3736-3756, (2010).

[6] J. Chen, J. Kneis, S. Lu, D. Molle, D. Richter, P. Rossmanith, S.-H. Sze, and F. Zhang, Randomized divide-and-conquer: improved path, matching, and packing algorithms, *SIAM Journal on Computingt 38*, pp. 2526-2547, (2009).

[7] L. Fortnow and R. Santhanam, Infeasibility of instance compression and succinct PCPs for NP, *Journal of Computer and System Sciences 77*, pp. 91-106, (2011).

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

[9] J. Gill, Computational complexity of probabilistic Turing machines, *SIAM Journal on Computing 6*, pp. 675-694, (1977).

[10] R. L. Graham, D. E. Knuth, and O. Pataschnik, *Concrete Mathematics - A Foundation for Computer Science*, Addison-Wesley, Reading, MA, 1992.

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

[12] R. Karp and R. Lipton, Some connections between nonuniform and uniform complexity classes, in *Proc. 12th ACM Symposium on Theory of Computing*, pp. 302-309, (1980).

[13] D. E. Knuth, *The Art of Computer Programming*, Vol. 2: Seminumerical Algorithms (3rd ed.), Addison-Wesley Professional, (1997).

[14] S. Mahaney, Sparse complete sets of NP: solution of a conjecture of Berman and Hartmanis, *Journal of Computer and System Sciences* 25-2, 130-143 (1982).

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

[16] N. Pippenger and M. J. Fischer, Relations among complexity measures, *Journal of the ACM* 26, 423-432 (1979).

[17] M. Sipser, A complexity theoretic approach to randomness, *Proc. 15th ACM Symposium on Theory of Computing*, 330-335 (1983).

[18] L. J. Stockmeyer, The polynomial-time hierarchy, *Theoreticl Computer Science* 3-1, 1-22 (1976).

[19] R. Williams, Finding paths of length k in $O^*(2^k)$ time, , *Information Processing Letters* 109-6, 315-318 (2009).

[20] C. K. Yap, Some consequences of non-uniform conditions on uniform classes, *Theoreticl Computer Science* 26, 287-300 (1983).