Lecture 3: Algorithms and Comp. Complexity — Discovery of P $\neq$ NP Conjecture

An *algorithm* is a way to describe computation for achieving a given task, in a way more flexible and abstract than actual computer programs. For achieving the same computational task, we may use various different algorithms, and clearly, there should be some difference between their "efficiency." Good algorithms (most typically, efficient algorithms) are crucial for designing good computer software systems, and thus, investigating the design and analysis of algorithms and understanding their limits are important subjects of foundations of computing theory. Furthermore, many researchers (not only computer scientists) have started realizing that an approach from view points of algorithms (which is sometimes called "algorithmic view" or "computational view") provides us with new understanding of our world.

A field studying algorithms formally is recently called *Algorithm Theory*, which includes (under my personal interpretation) a field called *Computational Complexity Theory*. In our lecture, we discuss the famous P $\neq$ NP conjecture as a typical topic in Algorithm Theory.

> **An Important Message from O.W:** Today I would like to explain things so that you can appreciate the current framework of investigating algorithms, and for this purpose, I will explain a little bit some hisotry of the discovery of the P $\neq$ NP conjecture. Hope that I can also have you realize that algorithms can be used to understand computation.

1 Basic Definitions for Computational Complexity

Definition 1. *A* problem *(or more specifically,* computational problem*) is a task for computing a given function $f : D \to R$.*
Remark. *f may be a* multi-valued *function, a function with more than one values; for such f, the task is to compute one of those values.*

Sets D and R are called respectively *domain* and *range*. Each element of D is called a (*problem*) *instance*. We assume that all objects for computation are encoded in binary strings, i.e., elements of $\{0,1\}^*$. Thus, both D and R are subsets of $\{0,1\}^*$. Problems with $R = \{0,1\}$ are called *decision problems*, where 1 means "yes" (or "accept") and 0 means "no" (or "reject"). We usually assume that $D = \{0,1\}^*$ for decision problems. The length (i.e., the number of bits) of each instance is called its *size*.

> **An Important Message from O.W:** This is the way that we have developed to capture computational tasks for discussing "efficiency" of algorithms. The important point here is that each problem is not for computing something for one instance or some finite set of instances, although, our actual task

in practice is sometimes to get a result on one particular data. Founders of Algorithm Theory might have considered that it would be more useful to design algorithms for solving infinite number of instances, and it turned out that this is the right/wise choice, in particular, for our computerized society.

Example 1. A graph is a combinatorial object consisting of *vertices* and *edges*. A graph G is specfied a pair (V, E), where V is the set of its vertices and E is the set of its edges. We may assume that a graph is encoded in $\{0,1\}^*$.

For any graph $G = (V, E)$, its (*vertex*) *coloring* is a way to give a color to each vertex in V; such a coloring is called *proper* if no pair of adjacent vertices is given the same color. (We say that two vertices are *adjacent* if they are connected by an edge.)

Min. Coloring Problem (abbrev. Min.COLOR)
Instance: A graph $G = (V, E)$.
Question: How many colors are needed for proper coloring of G?

Coloring Problem (abbrev. COLOR)
Instance: A graph $G = (V, E)$ and an integer k.
Question: Is there any proper coloring of G using at most k colors?

3-Coloring Problem, Search Version (abbrev. Search.3-COLOR)
Instance: A graph $G = (V, E)$.
Question: Find a proper coloring of G that uses at most k colors.
Remark. The domain is the set of graphs with proper 3-coloring.

3-Coloring Problem (abbrev. 3-COLOR)
Instance: A graph $G = (V, E)$.
Question: Is there any proper coloring of G using at most 3 colors?

By the way, we may define 2-Coloring problem similarly. □

Example 2. Let $G = (V, E)$ be any graph. A *cycle* of G is a tour traversing vertices using edges that starts some vertex in V and ends at this vertex. A *Hamilton cycle* of G is a cycle that visits all vertices of V exactly once (except for the start vertex that is visited twice). An *Euler cycle* of G is a cycle that visits all edges of E exactly once.

Hamilton Cycle Problem (abbrev. HAM)
Instance: A graph $G = (V, E)$.
Question: Is there any Hamilton cycle in G?

Euler Cycle Problem (abbrev. EULER)
Instance: A graph $G = (V, E)$.
Question: Is there any Euler cycle in G? □

Example 3.

Primarity Test (abbrev. PRIME)
Instance: An integer n.
Question: Is it a prime number?

Factorization (abbrev. FACT)
Instance: An integer n.
Question: Compute its prime factorization.

An important point to note here is that the size of a given input instance n is measured by its bit length. That is, the size of n is approximately $\log_2 n$. □

We would usually state an algorithm by a "pseudo program", that is, say, a C program like description but in less precise way. For discussing the efficiency of algorithms, we usually[1] consider a pseudo program that is concrete/detail enough so that we may assume that each line can be executed in a real computer by some constant number of instructions. Then we roughly consider the execution of each line as "one step" and define the time complexity of a given algorithm A as follows.

Definition 2. *Let A be an algorithm for some problem stated by a pseudo code that is precise enough. For any instance x, define* $\mathrm{time}_A(x)$ *by*

$$\mathrm{time}_A(x) \;=\; \textit{the number of executed steps until } A \textit{ halts on } x.$$

Then the time complexity *of A is a function on nonnegative integers that is defined as follows for each $\ell \geq 0$.*

$$\mathrm{time}_A(\ell) \;=\; \max\left\{\; \mathrm{time}_A(x) \,|\, x \in \{0,1\}^\ell \;\right\}.$$

This complexity measures is called a *worst-case complexity measure*. Clearly, the worst-case criteria is not our only choice, and it is not always the best choice either. For example, there are cases that the average-case analysis is more appropriate. But the average-case analysis is usually complex, and it is even more difficult to provide a simple and appropriate framework for studying the average-case complexity; so, average-case complexity has not been studied until recently.

Since algorithms are for infinite instances, each time complexity is not a number but a function on nonnegative integers. Thus, we need some way to compare two functions f and g and determine which one is larger. For this, the following asymptotic comparison was introduced.

[1]Of course, for discussing formally/precisely/in detail, we need to consider some precise model of computation and define "one step" precisely w.r.t. this model. Most typically, Turing machine model is used for such a computation model; but you can use any favorite model so long as it is precise and somewhat consistent with real computers.

Definition 3. *For any functions f and g from $\mathbf{Z}^+$ to $\mathbf{Z}^+$, we write $f = O(g)$ if the following holds for some constants c_0 and n_0.*

$$\forall n \geq n_0 \ [\ f(n) \ \leq \ c_0 \cdot g(n)\ ].$$

Remark. *The big-O notation does not imply the asymptotic equivalence between two functions; it just means that one function is asymptotically bounded* from above *by another function. For example, we can write $f = O(g)$ for $f(n) = n$ and $g(n) = 2^n$.*

Now we define basic complexity classes[2].

Definition 4. *For any function $t(\ell)$ on nonnegative integers, a complexity class* $\mathrm{Time}(t(\ell))$ *is defined by*

$$\mathrm{Time}(t(\ell)) \ = \ \left\{ \ f \ : \ f \text{ is computable by some } A \text{ such that } \mathrm{time}_A(\ell) = O(t(\ell)) \ \right\}.$$

Then define classes P *and* EXP *as follows:*

$$\mathrm{P} \ = \ \bigcup_{p(\ell):\text{polynomial}} \mathrm{Time}(p(\ell)), \qquad \mathrm{EXP} \ = \ \bigcup_{p(\ell):\text{polynomial}} \mathrm{Time}(2^{p(\ell)}).$$

One of the fundamental theorems of Complexity Theory is to show the following rather trivial fact: the more time we can use, the more difficult problems we can solve; but its proof is not trivial at all!

Theorem 1. *Let $t_1(\ell)$ and $t_2(\ell)$ be any functions on nonnegative integers. If $t_2 >> t_1$ (for example, $t_1(\ell) = O(t_1(\ell)$ but $t_2(\ell) \neq O(t_1(\ell)^2)$), then we have* $\mathrm{Time}(t_2(\ell)) \stackrel{\subsetneq}{\neq} \mathrm{Time}(t_2(\ell))$.

2 Class NP and the P $\neq$ NP Conjecture

Class NP is the class of problems such that checking the correctness of answers is not so difficult. This notion is defined formally as follows.

Definition 5. *A problem $f : D \to R$ is* NP search problem *if we have some polynomial-time computable predicate*[3] *R such that for any instance $x \in D$ and any $y \in R$, we have*

$$y = f(x) \iff R(x, y) = 1.$$

Class NP *is the class of NP search problems.*

Remark. *Usually we define NP as the class of decision problems that determine whether give an instance $x \in \{0,1\}^*$ has a solution in the corresponding NP search problem.*

---

[2]Usually these complexity classes are defined for decision problems. But here we relax this condition and consider any computational problems

[3]A predicate is a function whose value is either 1 ("yes") or 0 ("no").

Example 4. First consider the problems HAM and EULER of Example 2. Let D_{HAM} and D_{EULER} be sets of instances having a Hamilton cycle and an Euler cycle respectively. Let $f_{\text{HAM}} : D_{\text{HAM}} \to R$ and $f_{\text{EULER}} : D_{\text{EULER}} \to R$ be multi-valued functions mapping an instance graph to its Hamilton and Euler cycle respectively. Then it is easy to show that these functions define NP search problems. Then HAM and EULER are the corresponding decision problems in NP.

Similarly, it is easy to see that Search.3-COLOR is NP search problem, and 3-COLOR is its decision version that is in NP. Also COLOR (and its search version) is in NP. Note, on the other hand, that Min.COLOR does not seem to satisfy the condition for NP search problem; in fact, it is believed that Min.COLOR is not in NP. □

It is easy to see the following relations. Note that the condition for $f \in$ NP is a generalization of the condition of $f \in$ P; then the first relation P $\subseteq$ NP follows. The second relation is from the fact that every problem $f \in$ NP is solvable by some exponential-time algorithm.

Theorem 2.

$$\text{P} \subseteq \text{NP} \subseteq \text{EXP}.$$

It has been believed that some problem in NP has no polynomial-time algorithm, that is, they are not in P and we have P $\subsetneq$ NP. This is the P $\neq$ NP cojecture.

> **An Important Message from O.W:** Obviously once both P and NP are defined, one can naturally ask P = NP or not; that is, everyone can ask the P $\neq$ NP conjecture easily. But I would like to claim that the P $\neq$ NP cojecture was discovered by Cook and Karp (also independently by Levin in U.S.S.R.) because the importance of the class NP and the P $\neq$ NP cojecture would not have been realized unless they showed the key results stated below.

Theorem 3.
(1) *SAT is in P* $\iff$ NP $\subseteq$ P (Cook, 1971; Levin, 1973).
(2) *3COLOR is in P (resp., HAM is in P, etc)* $\iff$ NP $\subseteq$ P (Karp, 1972; Levin, 1973).

These results are also important because some novel method for analyzing computional properties has been introduced for proving these results. They proved that, e.g., SAT and 3COLOR have similar computational difficulty by developing algorithms for relating their difficulty. Here we see two quite simple examples of such algorithmic analysis.

It seems that the problem Min.COLOR is not in NP (and maybe slightly more difficult). But yet we can show that if COLOR is polynomial-time solvable, then Min.COLOR is also. That is, the following theorem.

Theorem 4. *COLOR is in P* $\implies$ *Min.COLOR is in P.*

Proof. To show this, we design an algorithm B that solves Min.COLOR by using (any algorithm) C for the problem COLOR as a subroutine. We design B so that it runs in

polynomial-time if we ignore the computational cost of running C. (We skip here for the description of the algorithm B, which will be explained on the blackboard in the class.)

Now with this B we can show the theorem. This is because if C is indeed a polynomial-time algorithm, then the whole computation of B (including the execution of C) can be done in polynomial-time. □

By a similar algorithmic approach, we can show that the search version of the problem HAM is no harder than HAM itself.

Theorem 5. *HAM is in P* $\implies$ f_{HAM} *is in P.*

These results (and most notably the above results of Cook, Levin, and Karp) show that there are some other ways to use algorithms besides programming; algorithms can be used to relate problems/tasks computationally for understanding their similarity/difference from computational view points.

Homework assignment from this lecture

Solve one of the following problems.

Q2.1. Knowing how to compare algorithms by using complexity measures would be important even though you are not designing algorithms by yourselves. At least you had better understand related notions and notations sufficient enough to solve the following questions.

(1) Is it always appropriate to define a given computational task as a task of computing some function? Suppose you want to design a Chess-software, a software that plays a chess with human. For designing its core algorithm, would it be possible to specify the required task by some function (or a set of functions)?

(2) Prof. W uses his algorithm A to analyze his experimental data, which is crucial for his research project. But his assistant K analyzed this algorithm and showed that it needs $\mathrm{time}_A(\ell) = 4 \cdot 2^{0.3\ell} \cdot 10^{-2}$ seconds for analyzing ℓMbyte data. This is not efficient. So he spent some months to develop a new algorithm B whose running time on ℓMbyte data is $\mathrm{time}_B(\ell) = 4000\ell^2 \cdot 10^{-2}$ seconds. Prof. W is unhappy that his assistant spends so long time only for algorithm design; in particular, there is no so much difference for the size of data (approximately 74Mbytes) that his group needs to analyze currently. Give at least two technical explanations for supporting the effort of assistant K. We may assume that for both programs, their running time depend only on input data size. (*Hint.* The amount of data his group needs to analyze is growing. In fact, Prof. W is planning to buy a new computer (10 times faster than the current one) for preparing this data increase.)

Q2.2. Show that NP $\subseteq$ EXP holds. Give some example of NP problem that does not seem to belong $\mathrm{Time}\big(2^{\ell^2}\big)$.