1

P versus NP, and More

Great Ideas in Theoretical Computer Science
Saarland University, Summer 2014

If you have tried to solve a crossword puzzle, you know that it is much harder to solve it than to verify a solution provided by someone else. Likewise, proving a mathematical statement by yourself is usually much harder than verifying a proof provided by your instructor. A usual explanation for this difference is that solving a puzzle, or proving a mathematical statement, requires some *creativity*, which makes it much harder than verifying a solution.

This lecture studies several complexity classes, which classify problems with respect to their hardness. We will define a complexity class P which consists of problems that can be solved efficiently, and a complexity class NP which consists of the problems that can be verified efficiently. One of the outstanding open questions in mathematics is whether $\mathsf{P} = \mathsf{NP}$, or verifying a statement is indeed much easier than proving a statement. We will discuss efforts theoreticians have made towards solving the question over the 40 years, and the consequences of this question.

1.1 P and NP

In early days of computer science, various algorithm design techniques are discovered which lead to efficient solutions of problems. We list a few here.

Problem 1.1 (Euler Cycle Problem)**.** *Give a graph G, is there a closed cycle which visits each edge of G exactly once? It is not difficult to prove that such cycle exists if and only if the degree of every vertex in G is even.*

Problem 1.2 (Shortest Path)**.** *Given a graph G and two vertices s and t, find the shortest path between s and t. The famous Dijkstra algorithm solving this problem is discovered by Edsgar Dijkstra in 1959 [4]. The algorithm runs in time linear in the number of edges in G.*

Problem 1.3 (Max Flow). *Given a graph G and two vertices s and t, compute the maximum flow between s and t. This problem was first formulated in 1954 by Ted Harris and Frank Ross. The first algorithm for this problem is presented in 1956, by Lester Ford, Jr., and Delbert Fulkerson [5].*

Algorithms solving these problems run in time polynomial in the size of the inputs, and we call such algorithms *efficient*. Efficient algorithms for solving problems are usually based on (i) mathematical properties of the problem, (ii) techniques of algorithm design, e.g. dynamic programing or greedy strategy, and (iii) use of data structures. However, there are problems for which the existence of efficient algorithms is unknown, and researchers still do not known the answer after 40 year's extensive studies. Let us look at some problems for example.

Problem 1.4 (Clique). *Given a graph G of n vertices, is there a clique of size k in G?*

Problem 1.5 (Hamiltonian Cycle). *Given a graph G of n vertices, is there a cycle of length n which visits every vertex exactly once?*

Problem 1.6 (Satisfiability). *Given a boolean formula ϕ with n variables and m clauses, where every clause has at least three literals, is φ satisfiable? That is, we ask if there is an assignment of $x_1, \ldots, x_n$ such that $\varphi(x_1, \ldots, x_n) = 1$?*

While efficient algorithms for these problems are unknown, these problems have some properties in common: (i) all existing algorithms for solving any of these problems are essentially based on brute-force search, i.e. listing all possible candidate solutions and checking if one candidate is indeed a solution. (ii) For every candidate solution, verifying the candidate is efficient and takes polynomial-time. Thousands of problems in different disciplines have been shown to have these two properties. A question of efficient algorithms for these problems essentially asks whether brute-force search can be avoided.

In 1956, Kurt Gödel wrote a letter to John von Neumann [7], and mentioned this problem in the letter. Despite that Kurt Gödel described the question in a remarkably modern way, the question can be more or less summarized by *how much we can improve upon the brute-force search*. Unfortunately, the question was stated as a mathematical logic problem and the Gödel's letter was "lost". It takes theoretical computer scientists almost 15 years until Stephen Cook re-formulated the P versus NP conjecture in [3]!

Search versus Decision. We have informally defined P as a set of problems which can be efficiently solved, and NP as a set of problems for which a solution can be efficiently verified. To give mathematical definitions of P and NP, we reduce search problems to decision problems, i.e. we study the decision problems ("Is this cycle a Hamiltonian cycle?") instead of search problems ("Find a Hamiltonian cycle."). Clearly, search problems are harder than decision problems. Solutions of a search problem gives a solution of decision problems. Moreover, the hardness of decision problems implies the hardness of the corresponding search problems.

Problems versus Sets. We build up correspondence between problems and sets. For any problem $\mathcal{P}$, we define a set $S_{\mathcal{P}}$, and $S_{\mathcal{P}}$ consists of the instances for which the answer to the decision problem is Yes. For instance, for a parameter k the Clique problem is defined as the set

$$\text{Clique} \triangleq \{G \mid \exists v_1, \ldots, v_k : G(v_1, \ldots, v_k) \text{ is a complete graph}\}, \tag{1.1}$$

where $G(v_1, \ldots, v_k)$ is the induced subgraph of G by $v_1, \ldots, v_k$. The satisfiability problem is defined as

$$\text{SAT} \triangleq \{\varphi \mid \exists x_1, \ldots, x_n : \varphi(x_1, \ldots, x_n) = 1\}. \tag{1.2}$$

That is, SAT consists of boolean formulae for which there is a satisfiable assignment.

Certificate. From (1.1) and (1.2), we know that every instance in the set has a short *certificate*, a string supporting the membership of every instance in the set. For instance, for the set Clique, every graph $G \in$ Clique can use vertices $v_1, \cdots, v_k$ as a certificate, and for the set SAT, every infeasible assignment is a certificate of $\varphi \in$ SAT. Moreover, given the certificate, membership of instances in a set can be effectively verified. For instance, let us look at the Hamiltonian problem. We know that an n-vertex graph $G \in$ Hamiltonian if and only if there is a permutation of G, denoted by $v_1, \ldots, v_n$, such that (i) every vertex u of G appears exactly once in the sequence, and (ii) for any $1 \leq i \leq n-1$, $\{v_i, v_{i+1}\}$, as well as $\{v_n, v_1\}$, are edges in G. For the SAT problem, it is easy to verify if $x_1, \ldots, x_n, x_i \in \{0, 1\}$, is a feasible assignment of φ.

P, NP and NP-Completeness. Now we define P and NP. The complexity class P formalizes the intuition of *efficiently solvable problems*, and consists of the problems that can be solved in polynomial-time. The complexity class NP formalizes the intuition of *efficiently verifiable problems*, and consists of all problems that admit a short "certificat" for membership. Given this certificate, also called a *witness*, membership can be verified efficiently in polynomial time.

Definition 1.7 (P)**.** *The class* P *consists of those problems that are solvable in polynomial time, i.e. the problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.*

Definition 1.8 (NP)**.** *The class* NP *consists of problems for which the solutions can be verified in polynomial-time. More formally,* NP *consists of sets L such that for all $x \in \{0, 1\}^\star$*

$$x \in L \Leftrightarrow \exists w \in \{0, 1\}^{p(|x|)} : M(x, w) = 1,$$

where $p : \mathbb{N} \mapsto \mathbb{N}$ is a polynomial, and M is a polynomial-time algorithm that is used to verify the membership of x.

Intuitively, an algorithm M verifies a language L if for any string $x \in L$, there is a certificate y that M can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$.

Example 1.9 (Composite Numbers)**.** *Given a number N, decide N is a composite. This problem*

is in NP*, since any number N is composite iff there is two numbers $p, q \geq 2$ such that $N = p \cdot q$. Hence any numbers p, q with $p \cdot q = N$ is the witness of N being composite.*

Example 1.10 (Subset Sum)**.** *Given a set S of numbers $x_1, \cdots, x_n$, and a number T, decide if there is a subset of numbers in S that sums up to T. The certificate is the list of members in S.*

Since if a problem in P then we can solve it in polynomial-time without knowing a certificate, we have that P $\subseteq$ NP. However, because of various practical problems for which only brute-force based algorithms are known, people are interested in finding polynomial-time algorithms for such problems.

To study this, Stephen Cook [3] and Leonid Levin [10] independently proposed the notion of NP-completeness around 1971, and gave examples of combinatorial NP-complete problems. They show that, in order to prove P $=$ NP, it suffices to study a few problems in NP. These problems are supposed to be the "hardest" problems in NP, and is called NP-complete problems (NPC problems). They further show that a polynomial-time algorithm for any problem in NPC implies P $=$ NP.

Definition 1.11 (Reduction)**.** *A language $L \subseteq \{0,1\}^\star$ is a polynomial-time reducible to a language $L' \subseteq \{0,1\}^\star$, denoted by $L \leq_p L'$, if there is a polynomial-time computable function $f : \{0,1\}^\star \mapsto \{0,1\}^\star$ such that for every $x \in \{0,1\}^\star$, $x \in L$ if and only if $f(x) \in L'$.*

We say that L' is NP*-hard if $L \leq_p L'$ for every $L \in$* NP*. We say that L' is* NP*-complete if L' is* NP*-hard and $L' \in$* NP*.*

Theorem 1.12. *If any* NP*-complete problem can be solved in polynomial-time, then every* NP*-complete problem has a polynomial-time algorithm.*

Theorem 1.13 (Cook-Levin Theorem)**.** SAT *is* NP*-complete.*

In 1972, Richard Karp further showed 21 problems where are NP-complete [8]. In particular, Karp's paper includes NP-completeness proofs for the clique, vertex cover, and the Hamiltonian cycle problem. After that, thousands of problems are shown to be NP-complete.

What are the consequences of the P vs. NP question? First, P $=$ NP implies that every efficiently verifiable problem can be solved efficiently, and brute-force search can be essentially avoided. Various *seemingly hard* problems have efficient algorithms for the exact solutions. Moreover, all randomized polynomial-time algorithms can be derandomized, and we do not need randomness in algorithm design. On a negative side, P $=$ NP implies that any encryption scheme has a trivial decoding scheme, and modern cryptography does not exist anymore.

On the other hand, P $\neq$ NP implies that brute-force search is essential and cannot be avoided for most problems. This also shows existence of hard problems which are the basis for cryptography.

Ladner Theorem. We know that (i) P $\subseteq$ NP, and (2) NPC $\subseteq$ NP $\setminus$ P, if P $\neq$ NP. A tricky question arising is that, assuming P $\neq$ NP, if there are problems that is in an "intermediate" state between P and NPC? I.e. is NP $\setminus$ P $\setminus$ NPC $= \emptyset$? Richard Ladner gave an affirmative answer to this question [9] in 1975.

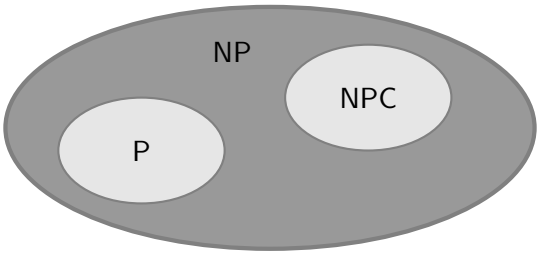

Figure 1.1: How most theoretical computer scientists view the relationship among P, NP, and NPC. Here $\mathsf{P} \neq \mathsf{NP}$ and $\mathsf{NP} \setminus \mathsf{P} \setminus \mathsf{NPC} \neq \emptyset$.

Theorem 1.14 (Ladner Theorem, 1975)**.** *If* $\mathsf{P} \neq \mathsf{NP}$*, then* $\mathsf{NP} \setminus \mathsf{P} \setminus \mathsf{NPC} \neq \emptyset$.

Ladner showed that if $\mathsf{P} \neq \mathsf{NP}$, then there are infinitely many levels of difficulties inside NP, see Figure 1.1. However, assuming that $\mathsf{P} \neq \mathsf{NP}$ we know few natural candidates which "should" be non NP-complete.

Problem 1.15 (graph isomorphism)**.** *Given two simple graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$*, decide if there is an isomorphism mapping between* G_1 *and* G_2*, i.e. a permutation* $\phi : V_1 \mapsto V_2$ *such that* $\{u, v\} \in E_1$ *if and only if* $\{\phi(u), \phi(v)\} \in E_2$.

Problem 1.16 (Factoring)**.** *Given a number* $N \in \mathbb{N}, L, U$*, decide if* N *has a prime factor* p *in the interval* $[L, U]$.

These two problems are in NP, and are conjectured to be not NP-complete. Graph Isomorphism is used to build zero-knowledge proofs, and Factoring is widely used nowadays in building crypto systems. Indeed, we do not know yet how to base cryptography on NP-complete problems.

1.2 Polynomial-Time Hierarchy

Oracle. Alam Turing introduced the notion of *oracle* in his PhD thesis in 1939. Informally, an oracle is a hypothetical device that would solve a computational program, free of charge. For instance, say we have a subroutine to multiply two matrices. After we create the subroutine, we do not have to think about how to multiply two matrices again, we simply think about it as a "black block" which always returns the correct answer.

Now assume that we have a program M, and program M uses another program A as a subroutine. Program M can invoke A as many times as it likes. We use M^A to represent such a program. Moreover, we denote P^A by the set of polynomial-time algorithms where each algorithm can use A as a subroutine and the runtime of A is assumed to be $O(1)$. Similarly, for complexity classes A, B, we define A^B as the set of problems which can be solved by using an algorithm in A that invokes another procedure in B as a subroutine.

Example 1.17. $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$.

Proof. Since use of oracles gives algorithms extra power, we have $\mathsf{P} \subseteq \mathsf{P}^{\mathsf{P}}$. Now we prove $\mathsf{P}^{\mathsf{P}} \subseteq \mathsf{P}$. Let L be any problem in P^{P}, then there is a polynomial-time algorithm $\mathcal{A}$ to solve L, where $\mathcal{A}$ uses another polynomial-time algorithm $\mathcal{O}$ as an oracle. Suppose that on input x the runtime of $\mathcal{A}$ is $p(|x|)$, and the runtime of $\mathcal{O}$ is $q(|x|)$, where p, q are polynomials. Now, instead of spending $O(1)$ time to get an answer from the oracle $\mathcal{O}$, algorithm $\mathcal{A}$ simulates the task of $\mathcal{O}$, and the runtime of this new algorithm is at most $p(|x|) \cdot q(|x|)$, which is a polynomial of $|x|$. Hence $L \in \mathsf{P}$, which implies that $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$. □

So far we discussed two complexity classes: P and NP. While most practical problems can be categorized into these two classes and most people in early 70s were studying the relationship between P and NP, a young graduate student, Larry Stockmeyer, and his advisor Albert Meyer started to think what the next step is. In the 1972's paper [11], they looked at the following problem.

Problem 1.18. *The set* MINIMAL *consists of all boolean formulas for which there is no shorter and equivalent boolean formula.*

We look at the complexity of $\overline{\mathsf{MINIMAL}}$, and show the following result.

Theorem 1.19. $\overline{\mathsf{MINIMAL}} \in \mathsf{NP}^{\mathsf{NP}}$.

Proof. Note that for any formula φ and φ' of n variables, if $\varphi \not\equiv \varphi'$, then there is an assignment $x'_1, \ldots, x'_n$ such that $\varphi(x'_1, \ldots, x'_n) \neq \varphi'(x'_1, \ldots, x'_n)$. This assignment $x'_1, \ldots, x'_n$ is the certificate of $\varphi \not\equiv \varphi'$. Hence the question of testing $\varphi \not\equiv \varphi'$ is in NP. Let $\mathcal{A} \in \mathsf{NP}$ be the algorithm testing if $\varphi \not\equiv \varphi'$.

Now we use $\mathcal{A}$ as an oracle. By definition, a formula $\varphi \in \overline{\mathsf{MINIMAL}}$ if and only if there is a formula φ' such that (1) the size of φ' is smaller than the size of φ, and (2) $\varphi \equiv \varphi'$.

By Item (1), φ' can be used as a certificate, and the property of Item (2) can be checked by oracle $\mathcal{A}$. Therefore we have that $\overline{\mathsf{MINIMAL}} \in \mathsf{NP}^{\mathsf{NP}}$. □

Polynomial-Time Hierarchy. In 1975, Larry Stockmeyer generalized the notion of P, NP, and oracles, and defined the Polynomial-Time Hierarchy [14]. Polynomial-Time Hierarchy contains an infinite number of subclasses, and these subclasses are conjectured to be distinct.

Definition 1.20 (Polynomial-Time Hierarchy)**.** *Σ_i is a sequence of sets and is defined inductively as follows:*

1. $\Sigma_0 \triangleq \mathsf{P}$, $\Sigma_1 \triangleq \mathsf{NP}$;
2. $\Sigma_{i+1} \triangleq \mathsf{NP}^{\Sigma_i}$.

Moreover, let $\Pi_i \triangleq \mathrm{co}\Sigma_i$, and $\Delta_{i+1} \triangleq \mathsf{P}^{\Sigma_i}$.

We say the Polynomial-Time Hierarchy is infinite if $\Sigma_i \neq \Sigma_{i+1}$ for any $i \geq 0$ and otherwise we say it collapses. We call the Polynomial-Time Hierarchy collapses to the ith level if $\Sigma_i = \Sigma_{i+1}$. In such case, we have that $\mathsf{PH} = \Sigma_i$.

Theorem 1.21. *The following statements hold:*

1. *If* $\Sigma_i = \Pi_i$*, then* $\mathsf{PH} = \Sigma_i$*;*
2. *If* $\Sigma_i = \Sigma_{i+1}$*, then* $\mathsf{PH} = \Sigma_i$*.*

Many complexity theoreticians conjecture that the Polynomial-Time Hierarchy is infinite and such a conjecture implies that many other complexity results. In computational complexity theory we will see many "pigs can fly" theorems, that show that if some conjecture does not hold then the Polynomial-Time Hierarchy collapses. If someone eventually does prove that the Polynomial-Time Hierarchy is infinite, then we will immediately get that all these conjectures are true. For instance, one such "pigs can fly" result relates the complexity of graph isomorphism to Σ_2.

Theorem 1.22 ([2])**.** *If Graph Isomorphism is* NPC*, then* $\mathsf{PH} = \Sigma_2$*.*

Problem 1.23. *Language* Σ_i-SAT *is the set of boolean formulae* φ *such that there are vectors* $\vec{x_i}$ *of boolean variables satisfying*

$$\exists \vec{x_1} \forall \vec{x_2} \cdots Q_i \vec{x_i} \varphi(\vec{x_1}, \ldots, \vec{x_i}) = 1,$$

where $Q_i = \forall$ *if* i *is even, and* $Q_i = \exists$ *otherwise. Language* Π_i-SAT *is defined similarly.*

Theorem 1.24. Σ_i-SAT *is* Σ_i*-complete, and* Π_i-SAT *is* Π_i*-complete.*

Theorem 1.24 shows that there are complete problems in every level of PH. In contrast to thousands of NP-complete problems we known, our knowledge of complete problems in various levels of PH is quite limited. A list of complete problems in higher levels of PH can be found in [12].

1.3 Remarks

- The definition of NP shows that for every language L in NP, and every $x \in L$, there is a short witness y, such that membership of x can be efficiently verified given the witness y. This definition implies that a non-deterministic algorithm $\mathcal{A}$ for language L. For any input x, algorithm $\mathcal{A}$ picks a random string y' as the witness and verifies the membership of x. If $x \notin L$, there is no witness at all, and algorithm $\mathcal{A}$ always outputs correct answers. Otherwise, $x \in L$, and algorithm $\mathcal{A}$ successes with nonzero probability.
- The complexity class P consists of problems that can be solved in polynomial-time. We can use a similar way to define the complexity class BPP, which consists of problems that can be solved by a randomized polynomial-time algorithm, i.e., for every problem $\mathcal{P}$ in BPP, there is a randomized algorithm that runs in polynomial-time and outputs correct solutions of $\mathcal{P}$ with probability at least 99%. Clearly, $\mathsf{P} \subseteq \mathsf{BPP}$. A lot of research in complexity theory indicates that randomization does not provide more power over

the deterministic algorithms, and most theoreticians believe that P = BPP.

- We already showed that computing exact solutions of certain problems are NP-hard. The study of *hardness of approximation* shows that, for many NP-hard problems, even computing an approximate solution with certain approximate guarantees are NP-hard.

1.4 Further Reading

"The history and status of the P versus NP question" is an excellent survey [13], in which you can find the Gödel's original letter and the translation in English. An essay by Scott Aaaronson titled "NP-Complete Problems and Physical Reality" [1] addresses the question if NP-complete problems can be solved efficiently in the physical reality. [6] summaries the work of Larry Stockmeyer.

References

[1] Scott Aaronson. Guest column: NP-complete problems and physical reality. *SIGACT News*, 36(1):30–52, 2005.

[2] Ravi B. Boppana, Johan Håstad, and Stathis Zachos. Does co-NP have short interactive proofs? *Inf. Process. Lett.*, 25(2):127–132, 1987.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing (STOC'71)*, pages 151–158, 1971.

[4] Edsgar Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian J. of Mathematics*, 8:399–404, 1956.

[6] Lance Fortnow. Beyond NP: the work and legacy of larry stockmeyer. In *37th Annual ACM Symposium on Theory of Computing (STOC'05)*, pages 120–127, 2005.

[7] Kurt Gödel. A letter to John von Neumann. http://rjlipton.wordpress.com/the-gdel-letter/.

[8] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[9] Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, 1975.

[10] Leonid A. Levin. Universal sequential search problems. *PINFTRANS: Problems of Information Transmission (translated from Problemy Peredachi Informatsii (Russian))*, 9, 1973.

[11] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT*, pages 125–129, 1972.

[12] M. Schaefer and C. Umans. Completeness in the polynomial-timer hierarchy: A compendium. http://ovid.cs.depaul.edu/documents/phcom.ps.

[13] Michael Sipser. The history and status of the P versus NP question. In *24th Annual ACM Symposium on Theory of Computing (STOC'92)*, pages 603–618, 1992.

[14] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.