CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 11: The Theory of NP

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

A Story

- Suppose you work in industry, and your boss gives you the task of finding an efficient algorithm for some problem very important to the company.
- After laboring long hours on the problem for over a month, you have no idea at all toward an efficient algorithm.
- Giving up, you return to your boss and ashamedly announce that you simply can't find an efficient algorithm.
  - Your boss: "I'm going to fire you and hire some smarter guys to solve this problem!"
  - You: "It is not because I'm stupid, but it is not possible to find an efficient algorithm for this problem!"
- So, your boss gives you another month to prove that it is impossible.

A Story

- After a second month of hardworking, you fail again.
  - At this point you've failed to obtain an efficient algorithm and you've failed to prove that such an algorithm is not possible.
  - You are on the edge of being fired.
- Suddenly, you discover that the company's problem is similar to the Traveling Salesperson Problem (TSP), and if the company's problem is solved, it will lead to a more efficient algorithm for TSP.
- However, no one has ever found a more efficient algorithm for TSP or proven that such an algorithm is not possible.
- You see one last hope. If you could prove that an efficient algorithm for the company's problem would automatically yield a more efficient algorithm for TSP, it would mean that your boss is asking you to accomplish something that even the greatest computer scientists can't solve.
- You ask for a chance to prove this, and your boss agrees.

A Story

- After only a week of effort, you do indeed prove that an efficient algorithm for the company's problem would automatically yield a more efficient algorithm for TSP.
- Rather than being fired, you're given a promotion because you have saved the company a lot of money.
- Your boss now realizes that it would not be worthy to continue to expend great effort looking for an exact algorithm for the company's problem and that other directions, such as looking for an approximate solution, should be explored.
- Happy ending~

Computational Complexity

- What we have just described is exactly what computer scientists have successfully done for the last 30 years.
  - We can't solve it, and we can't prove it is impossible, so we show that it is in the same class of other similar problems that are equally hard.
- If we had an efficient algorithm for any one of them, we would have efficient algorithms for all of them.
- Such an algorithm has never been found, but it's never been proven that one is not possible.
- These interesting problems are called *NP-complete* and are the focus of this lecture.

Intractability

- The dictionary defines *intractable* as “difficult to treat or work.”
- This definition is too vague to be of much use to us. To make the notion more concrete, we now introduce the concept of a “polynomial-time algorithm.”

Definition

A *polynomial-time algorithm* is one whose worst-case time complexity is bounded above by a polynomial function of its input size. That is, if n is the input size, there exists a polynomial $p(n)$ such that

$$W(n) \in O(p(n)).$$

Intractability

Example 1

Algorithms with the following worst-case time complexities are all polynomial-time.

$$2n \quad 3n^3 + 4n \quad 5n + n^{10} \quad n \lg n$$

Algorithms with the following worst-case time complexities are not polynomial-time.

$$2^n \quad 2^{0.01n} \quad 2^{\sqrt{n}} \quad n!$$

Intractability

- In computer science, a problem is called *intractable* if it is impossible to solve it with a polynomial-time algorithm.
- We stress that intractability is **a property of a problem**; it is not a property of any one algorithm for that problem.
- For a problem to be intractable, there must be no polynomial-time algorithm that solves it.
  - However, obtaining a nonpolynomial-time algorithm for a problem does not mean that the problem is intractable.

Intractability

- There are three general categories of problems as far as intractability is concerned:
  1. Problems for which polynomial-time algorithms have been found.
  2. Problems that have been proven to be intractable.
  3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found.
- It is a surprising phenomenon that most problems in computer science seem to fall into either the **first** or **third** category.
  - Its hard to prove the intractability of a problem (can't find a polynomial-time algorithm for it).

The Three General Problem Categories

Problems for which polynomial-time algorithms gave been found.

- Any problem for which we have found a polynomial-time algorithm falls in this first category.
- For example, we have found:
  - $\Theta(n \lg n)$ algorithms for sorting.
  - $\Theta(\lg n)$ algorithm for searching a sorted array.
  - $\Theta(n^{2.38})$ algorithm for matrix multiplication.
  - $\Theta(n^3)$ algorithm for chained matrix multiplication
- There are some other problems for which we have developed polynomial-time algorithms, but for which the obvious brute-force algorithms are nonpolynomial (usually exponential), include the Shortest Paths problem, the Optimal Binary Search Tree problem, and the Minimum Spanning Tree problem.

The Three General Problem Categories

Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found.

- This category includes any problem for which a polynomial-time algorithm has never been found, but yet no one has ever proven that such an algorithm is not possible.
- For example, the 0-1 Knapsack problem, the Traveling Salesperson problem, the Sum-of-Subsets problem, the m-Coloring problem for $m \geq 3$, and the Hamiltonian Circuits problem all fall into this category.
- We have found branch-and-bound algorithms, backtracking algorithms, and other algorithms for these problems that are efficient for many large instances. However, they are still not polynomial-time.

Decision Problem

- It is more convenient to develop the theory if we restrict ourselves to *decision problems*.
- The output of a decision problem is a simple “yes” or “no” answer.
- Yet when we introduced some of the problems mentioned previously, we presented them as optimization problems.
  - The output is an optimal solution.
- Each optimization problem, however, has a corresponding decision problem, as the examples that follow illustrate.

Decision Problem

Example 2: Traveling Salesperson Problem

- The *Traveling Salesperson Optimization problem* is to determine a tour with minimal total weight on its edges.
- The corresponding *Traveling Salesperson Decision problem* is to determine for a given positive number d whether there is a tour having total weight no greater than d.
  - This problem has the same parameters as the Traveling Salesperson Optimization problem plus the additional parameter d.

Decision Problem

Example 3: 0-1 Knapsack Problem

- The *0-1 Knapsack Optimization problem* is to determine the maximum total profit of the items that can be placed in a knapsack given that each item has a weight and a profit, and that there is a maximum total weight W that can be carried in the sack.
- The corresponding *0-1 Knapsack Decision problem* is to determine, for a given profit P, whether it is possible to load the knapsack so as to keep the total weight no greater than W, while making the total profit at least equal to P.
  - This problem has the same parameters as the 0-1 Knapsack Optimization problem plus the additional parameter P.

Decision Problem

- A polynomial-time algorithm for an optimization problem automatically solves the corresponding decision problem.
  - E.g., find the minimal total weight of TSP and compare it with d.
- For many decision problems, it's been shown that a polynomial-time algorithm for the decision problem would also yield a polynomial-time algorithm for the corresponding optimization problem.
  - Therefore, we can initially develop our theory considering only decision problems.

The Sets P and NP

Definition

P is the set of all decision problems that can be solved by polynomial-time algorithms.

- First we consider the set of decision problems that can be solved by polynomial-time algorithms.
- All decision problems for which we have found polynomial-time algorithms are certainly in P.
  - Searching, sorting, matrix multiplication…
- However, could some decision problem like TSP for which we have not found a polynomial-time algorithm also be in P?
  - We don't know! It could possibly be in P.
  - No one finds a polynomial-time algorithm and no one proves that it is not in P.

Polynomial-Time Verifiability

- It seems that the research on this kind of hard problem gets stuck.
- Instead, for this kind of decision problem like TSP, we'd like to if we can verify an answer in polynimial-time.
  - For TSP, to verify an answer is to return "yes" or "no" by given a tour.
- Now, we care about how to verify the answer rather than how the answer is found.
  - For TSP, verifying a tour is simply to sum up all its weights and compare it with d. It is obviously in polynomial-time.

Nondeterministic Algorithm

- To state the notion of polynomial-time verifiability more concretely, we introduce the concept of a *nondeterministic algorithm*.
- We can think of such an algorithm as being composed of the following two separate stages:
  1. **Guessing (Nondeterministic) Stage:** Given an instance of a problem, this stage simply produces some string S, a guess at a solution to the instance. It is called nondeterministic because any string in an arbitrary matter can be produced in this stage.
  2. **Verification (Deterministic) Stage:** The instance and the string S are the input to this stage. This stage then proceeds in an ordinary deterministic manner either (1) the answer for this instance is “yes”, (2) the answer for this instance is “no”, or (3) can’t verify at all (that is, going into an infinite loop). It is called deterministic because only “yes” or “no” can be produced in this stage.

Polynomial-Time Nondeterministic Algorithm

Definition

A *polynomial-time nondeterministic algorithm* is a nondeterministic algorithm whose verification stage is a polynomial-time algorithm.

- We are not sure if one can be solved in polynomial-time, but we know that we can verify an answer of it in polynomial-time.

Definition

NP is the set of all decision problems that can be solved by polynomial-time nondeterministic algorithms.

- Notice that NP stands for “nondeterministic polynomial”, rather than “non-polynomial”.

P and NP

- There are thousands of problems that no one has been able to solve with polynomial-time algorithms but that have been proven to be in NP because polynomial-time nondeterministic algorithms have been developed for them.
  - The answers to them can be verified in polynomial-time.
- Every problem in P is also in NP.
  - This is trivially true because any problem in P can be solved by a polynomial-time algorithm.
  - When it is solved, it is also verified.
- There are only few problems that have been proved not in NP.
  - The intractable problems, e.g. the Halting problem and Presburger Arithmetic.

P and NP

- Because P is in NP, it is easy to think that NP contains P as a proper subset.
- However, this may not be the case. That is, no one has ever proven that there is a problem in NP that is not in P.
- No one knows if P=NP or not yet.
  - If P=NP, it means that once we can verify answer of a problem in polynomial-time, we can solve it in polynomial-time!

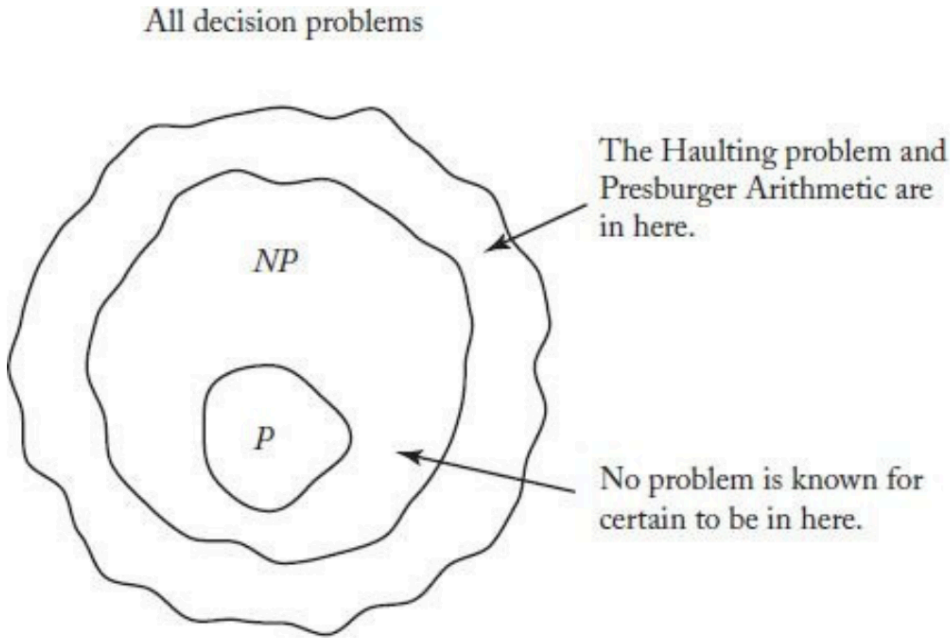

Image source: Figure 9.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

P=NP?

- To prove P=NP or P≠NP is one of the Millennium Prize Problems, which are seven problems in mathematics that were stated by the Clay Mathematics Institute on May 24, 2000.
  - Others are Birch and Swinnerton-Dyer conjecture, Hodge conjecture, Navier–Stokes existence and smoothness, Poincaré conjecture, Riemann hypothesis, and Yang-Mills existence and mass gap.
- A correct solution to any of the problems results in a US$1 million prize being awarded by the institute to the discoverer(s).
  - To date, the only problem that have been solved is the Poincaré conjecture, which was solved in 2003 by the Russian mathematician Grigori Perelman, who declined the prize money (amazing???).
  - Actually, if you are able to solve any of them, US$1 million is indeed not very important, because your name will be remembered by history forever.
- In 2002, a poll over 100 computer scientists showed that 61 of them believed P≠NP and 9 of them believed P=NP.

Statistic source: http://www.cs.umd.edu/~gasarch/papers/poll.pdf

Consequences of Solution

- Either P=NP or P≠NP is proved would advance theory enormously, and perhaps have huge practical consequences as well.
- If P=NP is proved:
  - Some cryptography encryption methods in NP will not be treated safe any more.
  - Researchers will be more confident to propose polynomial-time algorithms for problems in NP.
- If P≠NP is proved:
  - Some cryptography encryption methods in NP will be treated safe.
  - It would allow one to show in a formal way that many common problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems.

NP-Complete Problems

- Now, one wants to ask: are all problems in NP have the same difficulty?
  - For example, our dynamic programming algorithm for TSP is worst-case $\Theta(n^2 2^n)$ and for the 0-1 Knapsack problem is worst-case $\Theta(2^n)$.
  - The state space tree in the branch-and-bound algorithm for TSP has $(n-1)!$ leaves, whereas for the 0-1 Knapsack problem has only about 2^{n+1} nodes.
  - It seems that perhaps the 0-1 Knapsack problem is easier than TSP.
- How can we compare there difficulty?
  - We can build a relationship between problems.
  - If every solution of problem A is a solution of problem B, we can consider B is at least as hard as A.
  - These problem Bs are called NP-complete.

Transformation Algorithm

- Suppose we want to solve decision problem A and we have an algorithm that solves decision problem B.
- Suppose further there is an algorithm that creates an instance y of problem B from **every** instance x of problem A, such that an algorithm for problem B answers "yes" for y **if and only** if the answer to problem A is "yes" for x.
- Such an algorithm is called a *transformation algorithm* and is actually a function that maps every instance of problem A to an instance of problem B.

Reducibility

Definition

If there exists a polynomial-time transformation algorithm from decision problem A to decision problem B, problem A is polynomial-time many-one *reducible* to problem B. (Usually we just say that problem A *reduces* to problem B.) In symbols, we write

$$A \propto B$$

Theorem 1

If decision problem B is in P and $A \propto B$, then decision problem A is in P.

NP-Complete Problems

Definition

A problem B is called *NP-complete* if both of the following are true:

1. B is in NP.
2. For every other problem A in NP, $A \propto B$.

Theorem 2

A problem C is NP-complete if both of the following are true:

1. C is in NP.
2. For some other NP-complete problem B, $B \propto C$.

- By definition, NP-complete problems are the hardest NP decision problems, because every other problems in NP can reduce to NP-complete problems.

CNF-Satisfiability Problem

- A logical (Boolean) variable is a variable that can have one of two values: true or false.
- If x is a logical variable, $\bar{x}$ is the negation of x.
- A clause is a sequence of literals separated by the logical **OR** operator ($\vee$).
- A logical expression in Conjunctive Normal Form (CNF) is a sequence of clauses separated by the logical **AND** operator ($\wedge$).
- The following is an example of a logical expression in CNF:

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_3 \vee x_4)$$

- The *CNF-Satisfiability Decision problem* is to determine, for a given logical expression in CNF, whether there is some assignment that makes the expression true.

CNF-Satisfiability Problem

Theorem 3

(Cook's Theorem) CNF-Satisfiability is NP-complete.

- In 1971, Stephen Cook proves the first NP-complete problem.
- This problem itself is not interesting at all.
- The important thing is that we can use Theorem 2 and Theorem 3 to show a great number of problems being NP-complete, simply by showing that CNF-Satisfiability reduces to that problem.
- Then, amazing things happen.

Cook in 2008

Image source: https://en.wikipedia.org/wiki/Stephen_Cook

NP-Complete Problems

- It can be shown that:
  - CNF-Satisfiability ∝ Hamiltonian Circuits Decision problem.
  - Hamiltonian Circuits Decision problem ∝ Undirected Traveling Salesperson Decision problem.
  - Undirected Traveling Salesperson Decision problem ∝ Traveling Salesperson Decision problem.
  - …
- Now, we don't need to use the definition to prove the NP-completeness of a problem. Instead, simply find another NP-complete problem and show the reducibility.

Proof of an NP-Complete Problem

- Generally, proving NP-completeness of a problem A consists of the following steps:
  - Show that A is in NP.
  - Choose an NP-complete problem B, prove that $B \propto A$.
    - Describe the transformation algorithm f.
    - Argue that if every instance x of B maps to an instance $f(x)$ of A.
    - Argue that the answer to problem B is "yes" for x **if and only** if the answer to problem A is "yes" for $f(x)$.
    - Briefly explain why f is computable in polynomial-time.

Proof of an NP-Complete Problem

Example 4: Proof the NP-completeness of Partition problem.

- The Partition problem is defined as follows:
- Given a sequence of integers $A = \{a_1, \dots, a_n\}$, determine whether there is a partition of the integers into two subsets such that the sum of the elements in one subset is equal to that of the other.
- Formally, determine whether there exists an answer $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = \sum_{i=1}^{n} a_i \,/2$.
  - For example, given $\{a_1, a_2, a_3, a_4, a_5\} = \{1, 6, 4, 3, 2\}$, we have $a_1 + a_3 + a_4 = a_2 + a_5 = 8$.
- It is a decision problem.

Proof of an NP-Complete Problem

Example 4 (cont'd)

- **Step 1**: Show that Partition decision problem is in NP by checking its verification stage in polynomial-time or not.
  - Observe that if someone gives us the set S, we can determine the sums of the elements in the two subsets and check that whether they are equal to each other or not.
  - Input: A set of index S.
  - Output: Yes or No.
    - $sum_1 = \sum_{i \in S} a_i, sum_2 = \sum_{i=1}^{n} a_i - sum_1.$
    - If $sum_1 == sum_2$, return Yes; else return No.
- It is in polynomial-time.

Proof of an NP-Complete Problem

Example 4 (cont'd)

- **Step 2**: Prove that Subset-Sum $\propto$ Partition.
- Compare the two problems:
  - Subset-Sum (can be found in NP-complete problem list)
    - Instance: $(a_1, \dots, a_n, t)$, where t and all a_i are integers.
    - Question: whether there exists an answer $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = t$.
  - Partition
    - Instance: $(a_1, \dots, a_n)$, where all a_i are integers.
    - Question: whether there exists an answer $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = \sum_{j \notin S} a_j$.

Proof of an NP-Complete Problem

Example 4 (cont'd)

- Transform every instance of Subset-Sum to an instance of Partition.
- Let $x = (a_1, \ldots, a_n, t)$ be an instance of Subset-Sum and $a = \sum_{i=1}^{n} a_i$. We define the transformation algorithm as

$$f(x) = (a_1, a_2, \ldots, a_n, a_{n+1})$$

  where $a_{n+1} = 2t - a$.
- It is clear that this transform can be done in polynomial time.
- Then, we want to show that:

The answer S to Subset-Sum problem is "yes" for x

$\Leftrightarrow$ The answer S to Partition problem is "yes" for $f(x)$.

Proof of an NP-Complete Problem

Example 4 (cont'd)

- Prove $\Longrightarrow$:
  - Let the answer $S \subseteq \{1,2,\ldots,n\}$ to Subset-Sum problem is "yes" for x such that $\sum_{i\in S} a_i = t$ and $\sum_{i=1}^{n} a_i = a$.
  - Let $T = \{1,2,\ldots,n+1\} - \mathrm{S}$, we have

$$\sum_{j\in T} a_j = a + a_{n+1} - t = a + 2t - a - t = t = \sum_{i\in S} a_i .$$

  - Because $\sum_{i\in S} a_i = \sum_{j\in T} a_j = \sum_{j\notin S} a_j$, the answer S to Partition problem is also "yes" for $f(x)$.

Proof of an NP-Complete Problem

Example 4 (cont'd)

- Prove $\Longleftarrow$:
  - If there exists the solution $S \subseteq \{1,2,\ldots,n\}$ to Partition problem is also "yes" for $f(x)$, such that letting $T = \{1,2,\ldots,n+1\} - S$ we have

$$\sum_{i \in S} a_i = \sum_{j \in T} a_j = \frac{a + (2t - a)}{2} = t.$$

  - Without loss of generality, assume that $n+1 \in T$, then we have $S \subseteq \{1,2,\ldots,n\}$ and $\sum_{i \in S} a_i = t$.
  - Because $\sum_{i \in S} a_i = t$, the answer S to Subset-Sum problem is also "yes" for $f(x)$.

Proof of an NP-Complete Problem

Example 4 (cont'd)

- Because
  - Step 1: Partition is in NP,
  - Step 2: Subset-Sum is NP-complete and Subset-Sum $\propto$ Partition,

we can conclude that Partition problem is NP-complete.

NP-Hard Problems

- Notice that all NP-complete problems are decision problems. Next we extend our results to problems in general.

Definition

If problem A can be solved in polynomial time using a hypothetical polynomial time algorithm for problem B, then problem A is *polynomial-time Turing reducible* to problem B. (Usually we just say A Turing reduces to B.) In symbols, we write

$$A \propto_T B$$

NP-Hard Problems

Definition

A problem B is called NP-hard if, for some NP-complete problem A,

$$A \propto_T B$$

- Therefore, all problems in NP reduce to any NP-hard problem. This means that if a polynomial-time algorithm exists for any NP-hard problem, then P = NP.
- Every NP-complete problem is NP-hard, because NP-complete problems reduces to each other .
- It can be shown that Traveling Salesperson Decision problem $\propto_T$ Traveling Salesperson Optimization problem.
  - Therefore, TSP is NP-hard.

Relationship

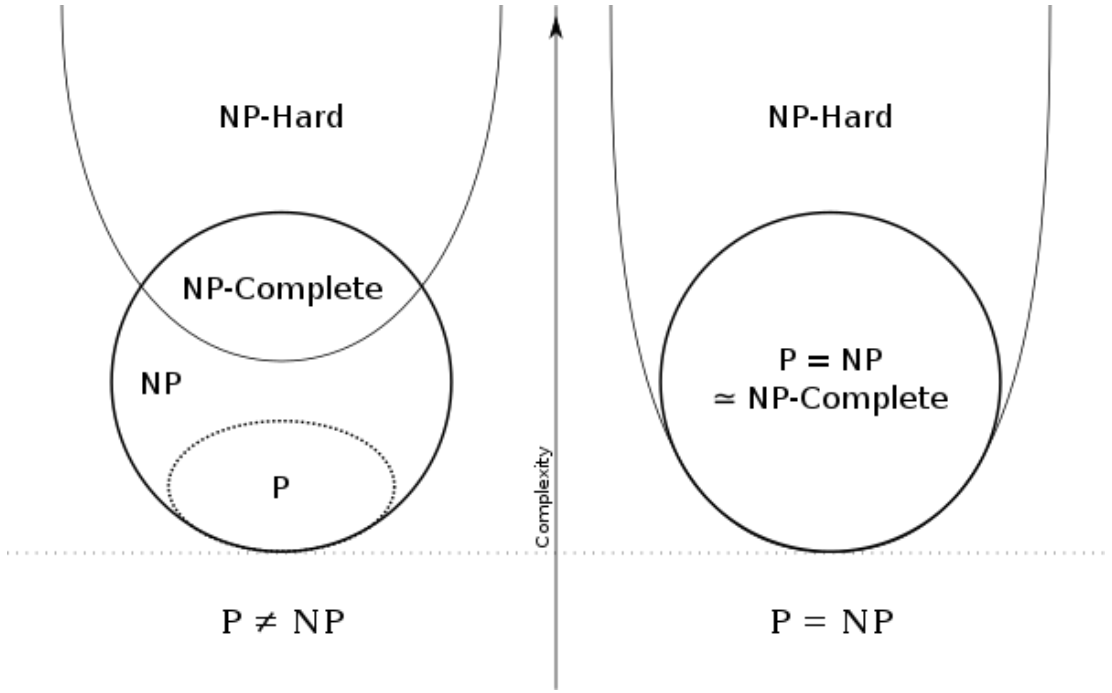

Image source: https://en.wikipedia.org/wiki/NP-completeness

Handling NP-Hard Problems

In the absence of polynomial-time algorithms for problems known to be NP-hard, what can we do about solving such problems?

- The dynamic programming, backtracking and branch-and-bound algorithms for these problems are all worst-case nonpolynomial-time.
  - However, they are often efficient for many large instances.
- Another approach is to develop approximation algorithms.
  - An approximation algorithm is an algorithm that is not guaranteed to give optimal solutions, but rather yields solutions that are reasonably close to optimal.
  - Often we can obtain a bound that gives a guarantee as to how close a solution is to being optimal.
    - E.g. for TSP, we show that $minapprox < 2 \times mindist$.

Conclusion

After this lecture, you should know:

- What is a polynomial-time algorithm.
- What are P and NP.
- How a problem can be reduced to another problem.
- What is an NP-complete problem.
- How to prove a problem being NP-complete.
- What is an NP-hard problem.

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺