Optimization Theory and Methods

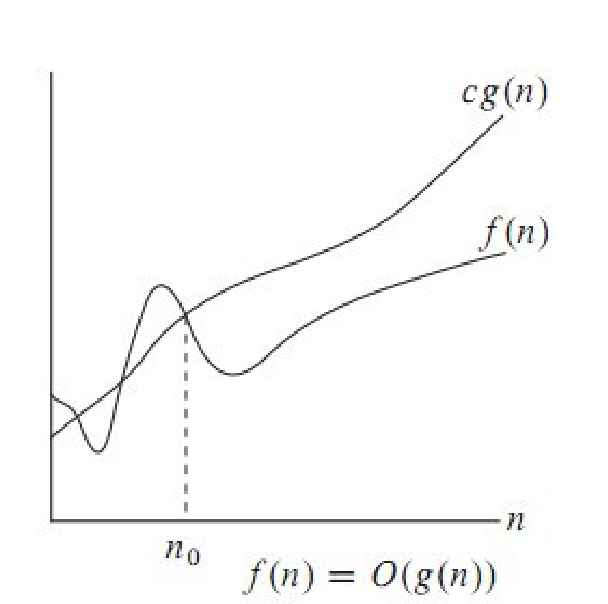
2025 Autumn

魏可佶 kejiwei@tongji.edu.cn
https://kejiwei.github.io/

- Actual number of steps taken by an algorithm and the actual run times will depend on specific problem instance.
- So let us aim to find an upper bound on the number of steps taken by an algorithm.
- A step is an arithmetic operation like addition, subtraction, multiplication, division, comparison, assignment, etc.
- As the problem size (that is, number of nodes and arcs for the case of network problems) increases, the run time and the number of steps will obviously increase.
- So our upper bounds on the run times will be functions of number of nodes (n) and number of arcs (m).
- We are quite satisfied if we are within a constant factor. Otherwise the task becomes too complex.

- We say that an algorithm runs in **polynomial time** if the number of steps taken by the algorithm is bounded above by a polynomial in n and m.
- We use big '$\boldsymbol{O}$' to indicate upper bounds.
- For example, we may say that an algorithm is $\boldsymbol{O(n^2)}$. That means the algorithm takes at most cn^2 steps for some constant c. E.g., at most $14n^2$ steps.
- We say that an algorithm runs in **exponential time** whenever it does not run in polynomial time.

- $f(n) = O(g(n))$: *iff* there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$
- O-notation to give an upper bound on a function.



- **Sorting a list of n numbers:** [42, 3, 17, 26, … , 100]

$$n \, log_2 n$$

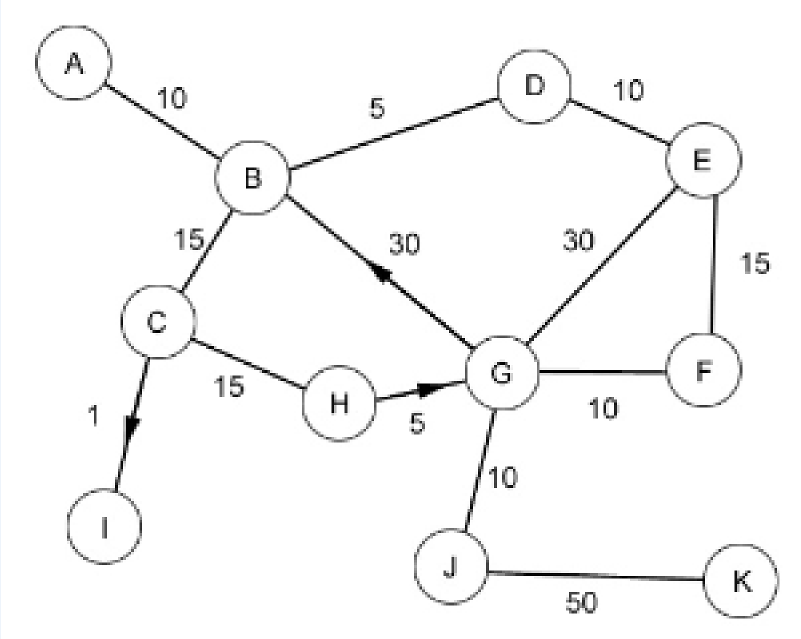
- **Multiplying two *n* x *n* matrices:**

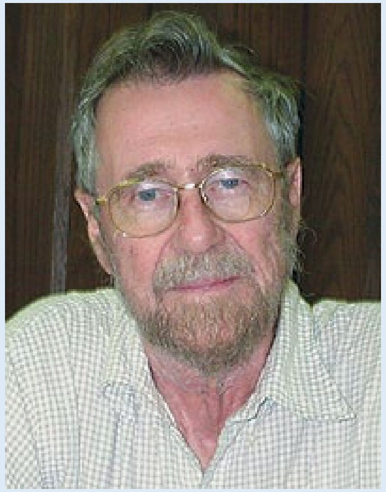
$$
n \underset{n}{\begin{pmatrix} 3 & 5 & 2 & 7 \\ 1 & 6 & 8 & 9 \\ 2 & 4 & 6 & 10 \\ 9 & 3 & 2 & 12 \end{pmatrix}} \underset{n}{\begin{pmatrix} 1 & 5 & 5 & 4 \\ 5 & 12 & 8 & 6 \\ 7 & 6 & 1 & 5 \\ 9 & 23 & 5 & 8 \end{pmatrix}} = \underset{n}{\begin{pmatrix} \quad & \quad & \quad & \quad \\ & & & \\ & & & \\ & & & \end{pmatrix}} n
$$

↳ "Easy" Problems(cont.)

- The Shortest Path Problem (i.e. "Google Maps")

Depending on implementation:
$O(|V|^2)$ or $O(|E| + |V|\text{Log}|V|)$





Edsgar Dijkstra

https://www.cs.hmc.edu/~cs5grad/cs5/LectureSlides/class07-black-16-functional5.pptx

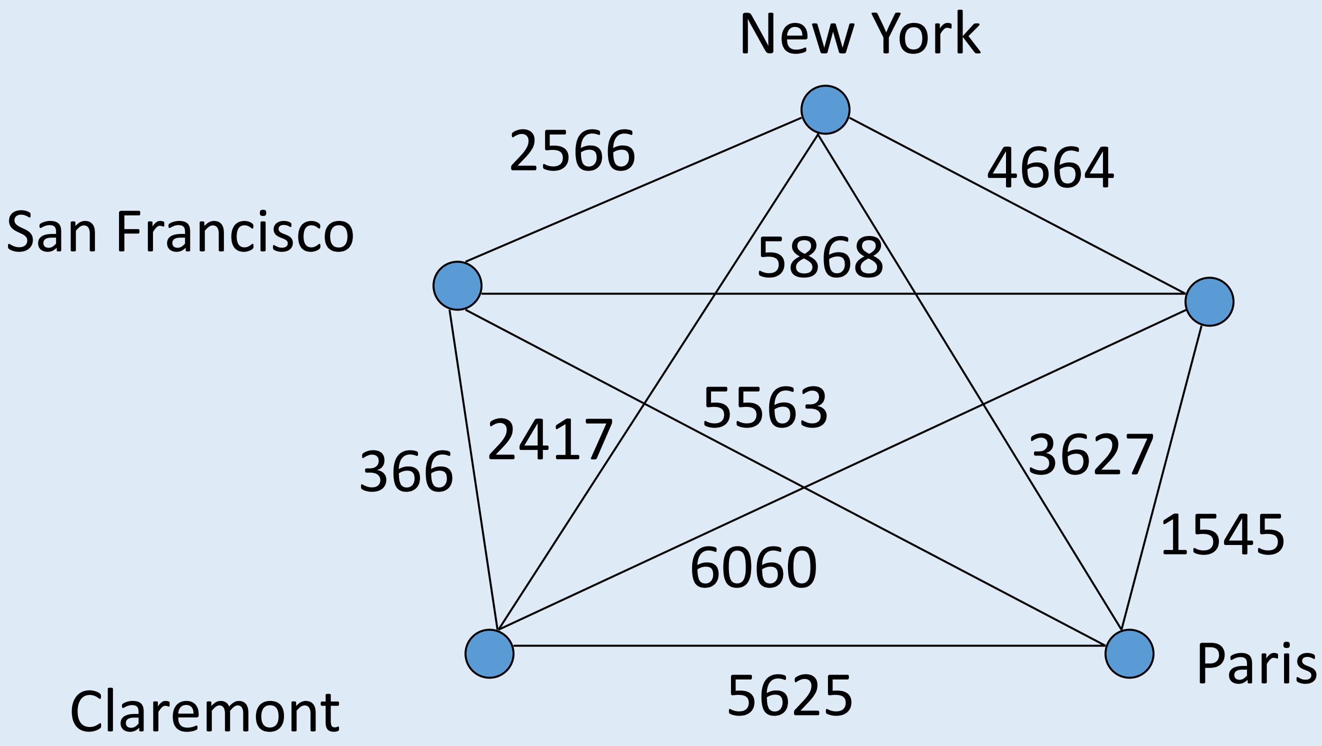
- "Polynomial Time" = "Efficient"

  $n, n^2, n^3, n^4, n^5, \ldots$

The "class" P

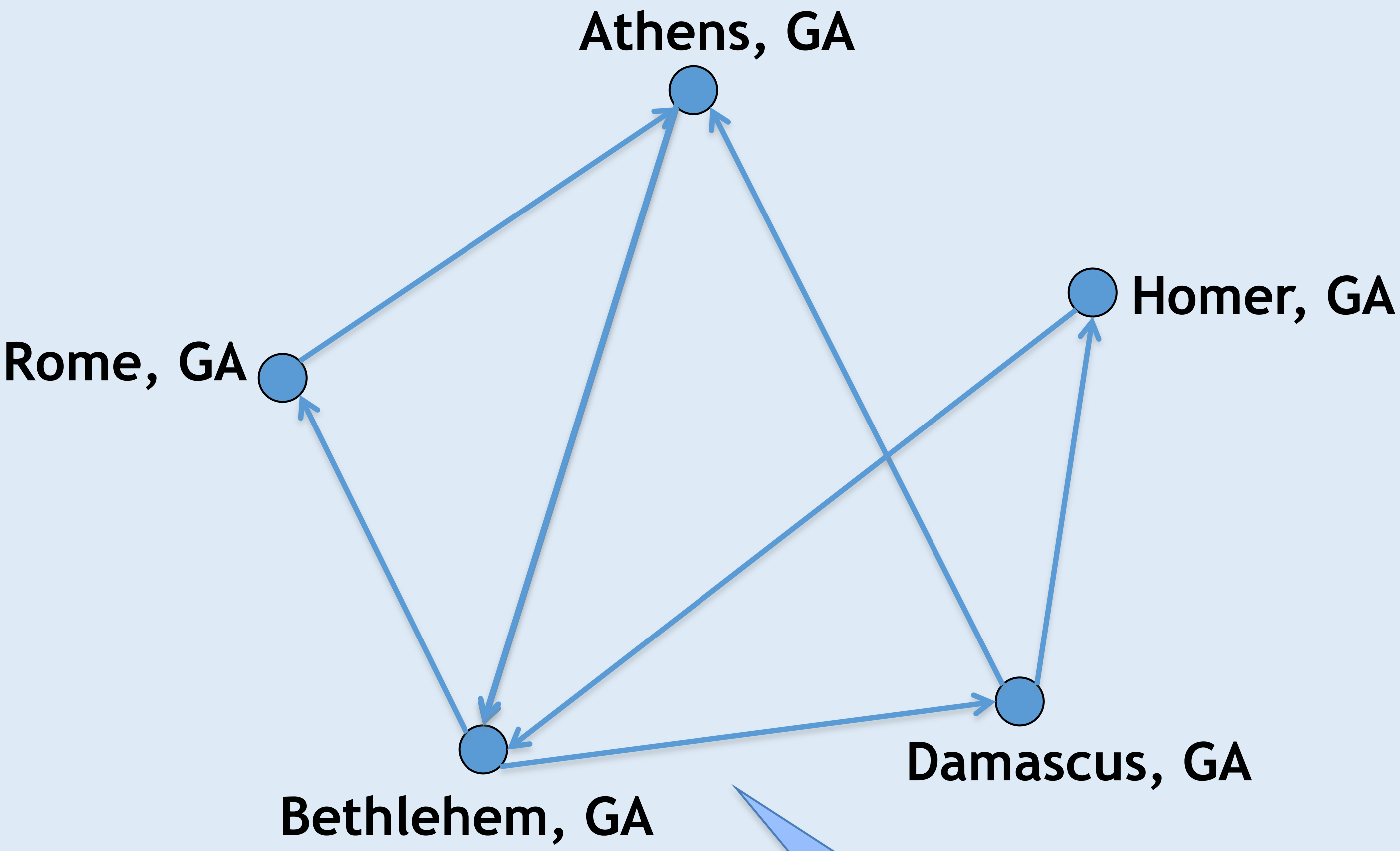
How about something like $n \log_2 n$?

- The Travelling Salesperson Problem



Brute Force? Greed?

↳ The Hamiltonian Path Problem



Those are some peachy names!

- The Geoff-O-Matic performs 10^9 operations/sec

	n = 10	n = 30	n = 50	n = 70
n^2	100 < 1 sec	900 < 1 sec	2500 < 1 sec	4900 < 1 sec
2^n	1024 < 1 sec	10^9 1 sec	10^{15} 11.6 days	**10^{21}** **31,688 years**
$n!$	< 1 sec	10^{16} years	10^{57} years	10^{93} years

- Some problems are ***intractable***:
  as they grow large, we are unable to solve them in reasonable time
  - Not in polynomial time: $O(2^n)$, $O(n!)$, $O(n^n)$,

- What constitutes reasonable time?
  - Standard working definition: ***polynomial time***
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - Polynomial time: $O(1)$, $O(n \lg n)$, $O(n^2)$, $O(n^3)$,

- Optimization Problems
  - An optimization problem is one which asks, “What is the optimal solution to problem X?”
  - Examples:
    - Maximal Matching
    - Traveling Salesperson
    - Minimum Spanning Tree
- Decision Problems
  - An decision problem is one with yes/no answer
  - Examples:
    - Does a graph G have a MST of weight $\leq$ W?

↳ Optimization/Decision Problems (cont.)

- An ***optimization problem*** tries to find an optimal solution
- A ***decision problem*** tries to answer a yes/no question
- Many problems will have decision and optimization versions
  - Eg: Traveling salesman problem
    - optimization: find hamiltonian cycle of minimum weight
    - decision: is there a hamiltonian cycle of weight $\leq k$
- Some problems are decidable, but ***intractable***:
  as they grow large, we are unable to solve them in reasonable time
  - ***Is there a polynomial-time algorithm that solves the problem?***

- The *class P* consists of those problems that are solvable in polynomial time.
- More specifically, they are 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.
- The key is that ***n*** is the **size of input.**

“Easy” Problems

- ***P***: the class of decision problems that have polynomial-time deterministic algorithms.
  - That is, they are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n
  - A deterministic algorithm is (essentially) one that always computes the correct answer

- Why polynomial?
  - if not, very inefficient
  - nice closure properties
    - *the sum and composition of two polynomials are always polynomials too*

- Deterministic in nature
- Solved by conventional computers in polynomial time
  - $O(1)$ Constant
  - $O(\log n)$ Sub-linear
  - $O(n)$ Linear
  - $O(n \log n)$ Nearly Linear
  - $O(n^2)$ Quadratic
- Polynomial upper and lower bounds

- Shortest Path Dijkstra **algorithm** $O(n^2)$.
- Eulerian path O(E)
- MST O(ElogV)
- Merge Sort
- **Huffman Algorithm:** Constructing the ***Optimal Binary (Huffman) Tree***.
- Others

Single-Source Bottleneck Path Algorithm Faster than Sorting for Sparse Graphs

Ran Duan *1, Kaifeng Lyu †1, Hongxun Wu ‡1, and Yuanhang Xie §1

[1]Institute for Interdisciplinary Information Sciences, Tsinghua University

Abstract

In a directed graph $G = (V, E)$ with a capacity on every edge, a *bottleneck path* (or *widest path*) between two vertices is a path maximizing the minimum capacity of edges in the path. For the single-source all-destination version of this problem in directed graphs, the previous best algorithm runs in $O(m + n \log n)$ ($m = |E|$ and $n = |V|$) time, by Dijkstra search with Fibonacci heap [Fredman and Tarjan 1987]. We improve this time bound to $O(m\sqrt{\log n})$, thus it is the first algorithm which breaks the time bound of classic Fibonacci heap when $m = o(n\sqrt{\log n})$. It is a Las-Vegas randomized approach. By contrast, the s-t bottleneck path has an algorithm with running time $O(m\beta(m, n))$ [Chechik et al. 2016], where $\beta(m, n) = \min\{k \geq 1 : \log^{(k)} n \leq \frac{m}{n}\}$.

- **NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**
- NP means verifiable in polynomial time
- Verifiable?
  - If we are somehow given a ‘certificate’ of a solution we can verify the legitimacy in polynomial time

- MST
- Maximal matching
- Hamiltonian Cycle (Traveling Salesman)
- Graph Coloring

- Determining whether a directed graph has a Hamiltonian cycle does not have a polynomial time algorithm (yet!)
- However if someone was to give you a sequence of vertices, determining whether or not that sequence forms a Hamiltonian cycle can be done in polynomial time.
- **Therefore Hamiltonian cycles are in NP.**

“Hard” Problem?

- Graph theory has these fascinating (annoying?) pairs of problems
  - Shortest path algorithms?
  - Longest path is NP complete (we'll define NP complete later)
  - Eulerian tours (visit every vertex but cover every edge only once, even degree etc). Solvable in polynomial time!
  - Hamiltonian tours (visit every vertex, no vertices can be repeated). NP complete

- **P** = set of problems that can be solved in polynomial time
- **NP** = set of problems for which a solution can be verified in polynomial time
- Clearly **P** $\subseteq$ **NP**
- Open question: Does **P = NP**?
  - Most suspect not
  - An August 2010 claim of proof that P ≠ NP, by Vinay Deolalikar, researcher at HP Labs, Palo Alto, has flaws

- A decision problem D is NP-complete iff

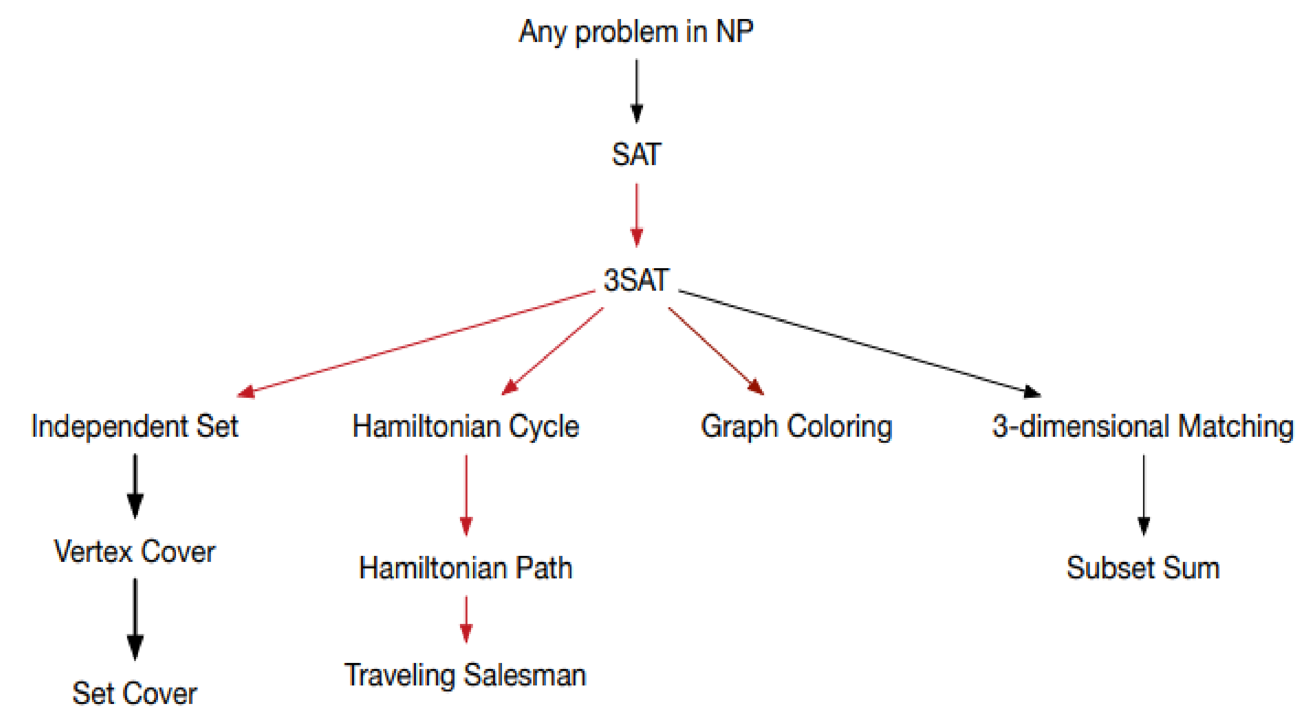
  1） $D \in NP$

  2） Every problem in NP is polynomial-time reducible to D

- A problem **R** can be ***reduced*** to another problem **Q** if any instance of **R** can be rephrased to an instance of **Q**, the solution to which provides a solution to the instance of **R**
  - This rephrasing is called a ***transformation***
- ***Intuitively***: If **R** reduces in polynomial time to **Q**, **R** is "no harder to solve" than **Q**
- Example: lcm(m, n) = m * n / gcd(m, n),

  lcm(m,n) problem is reduced to gcd(m, n) problem

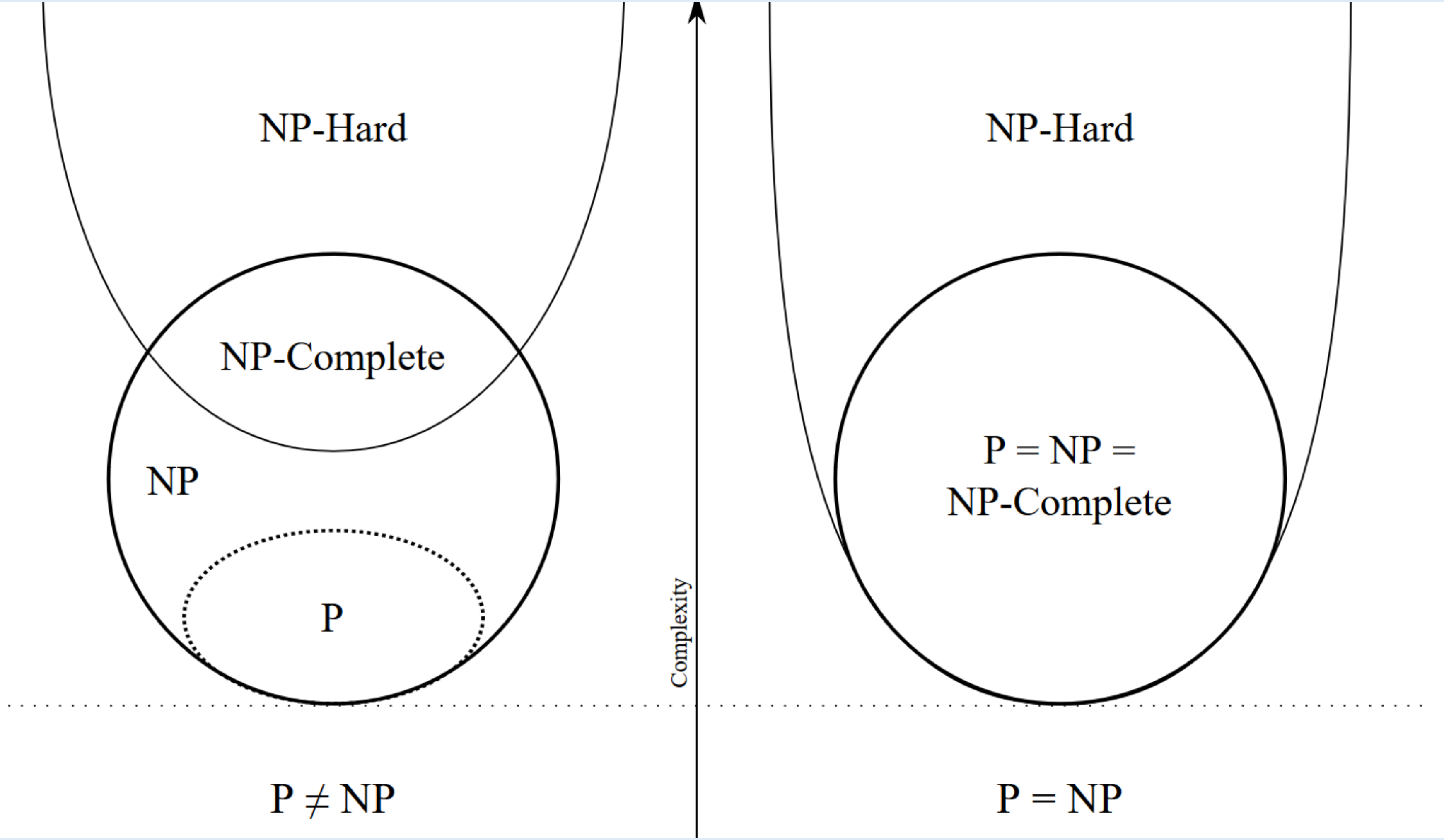
- Language **L** is polynomial-time reducible to language **M** if there is a function computable in polynomial time that takes an input **x** of **L** and transforms it to an input ***f*(*x*)** of **M**, such that **x** is a member of **L** if and only if ***f*(*x*)** is a member of **M**.



- If R is *polynomial-time reducible* to Q, we denote this $R \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in$ **NP** are *polynomial-time* reducible to Q, then Q is *NP-Hard*

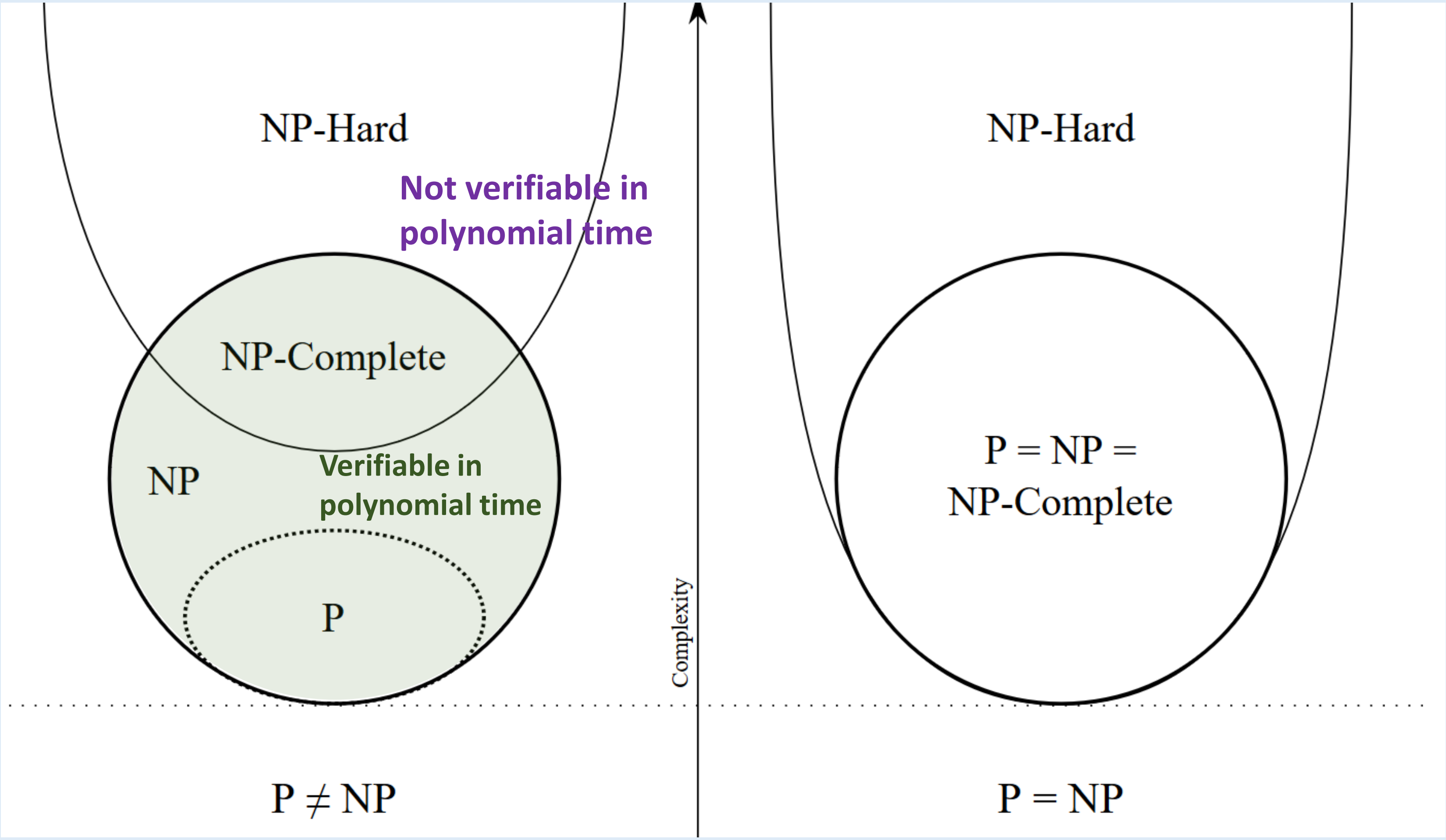
    Note: An NP-Hard problem need not be NP.
  - An NP-Hard problem is at least as hard as the NP-complete problems.
  - We say Q is *NP-Complete* if Q is NP-Hard and $Q \in$ **NP**
- If $R \leq_p Q$ and R is NP-Hard, Q is also NP-Hard (why?)

↳ NP-Hard and NP-Complete (cont.)



https://upload.wikimedia.org/wikipedia/commons/a/a0/P_np_np-complete_np-hard.svg

↳ NP-Hard and NP-Complete (cont.)



https://upload.wikimedia.org/wikipedia/commons/a/a0/P_np_np-complete_np-hard.svg

Objective :

Key Concepts :