P, NP, NP Complete, NP Hard problems & Approximation Algorithm

By

Ajitha Shenoy K B

Optimization Problem

- Optimization problems are those for which the objective is to maximize or minimize some values.

For example,

- 0/1 Knapsack Problem.
- Finding the shortest path between two vertices in a graph.
- Travelling Sales Person Problem (TSP)

Decision Problem

- There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**.

For example,

- Whether a given graph can be colored by only 4-colors?
- Decision version of optimization problem: example:
  - Is there exist a tour in TSP of cost less than or equal to M (for some M).
  - Is there exist a solution to 0/1 Knapsack problem which has profit greater than or equal to M?

Class P Problem

- Class P Problems: A decision Problem X is said to be in class P, if it can be solved using deterministic Turing machine (deterministic algorithm) in polynomial time.

- Example: Sorting, Searching, Finding shortest path, container loading problem etc.

Class NP Problems

- Class NP problems: **NP stands for Non deterministically Polynomial**

A decision problem X is said to be in class NP if it can be solved in polynomial time using non-deterministic Turing machine (Probabilistic algorithm, "guess").

Equivalently: A problem X is said to be in NP if the solution is **verified** in polynomial time using deterministic algorithm.

Example: Decision version of 0/1 knapsack problem, TSP

Not in NP: Optimization problem TSP, 0/1 Knapsack problem etc.

P Vs NP

- Million Dollar question ! P = NP ?   OR P != NP?
- Clearly class P is subset of class NP (Since any problem which can be solved in polynomial time using deterministic machine can also be solved using non-deterministic algorithm.
- But whether NP is subset of P or not is a big research problem!
- Why this research question is important? (Discussed after explainin NP complete and NP hard problems)

NP-Complete

- A problem is NP-complete if it is both **NP-hard** and in **NP**.

- i.e., A problem X is said to be NP-complete if
  - X is in NP (verifiability) AND
  - X is NP–hard (reducibility)

NP-hard

Polynomial-time reductions: If problem A can be polynomial-time reduced to problem B, then it stands to reason B is at least as hard as A.

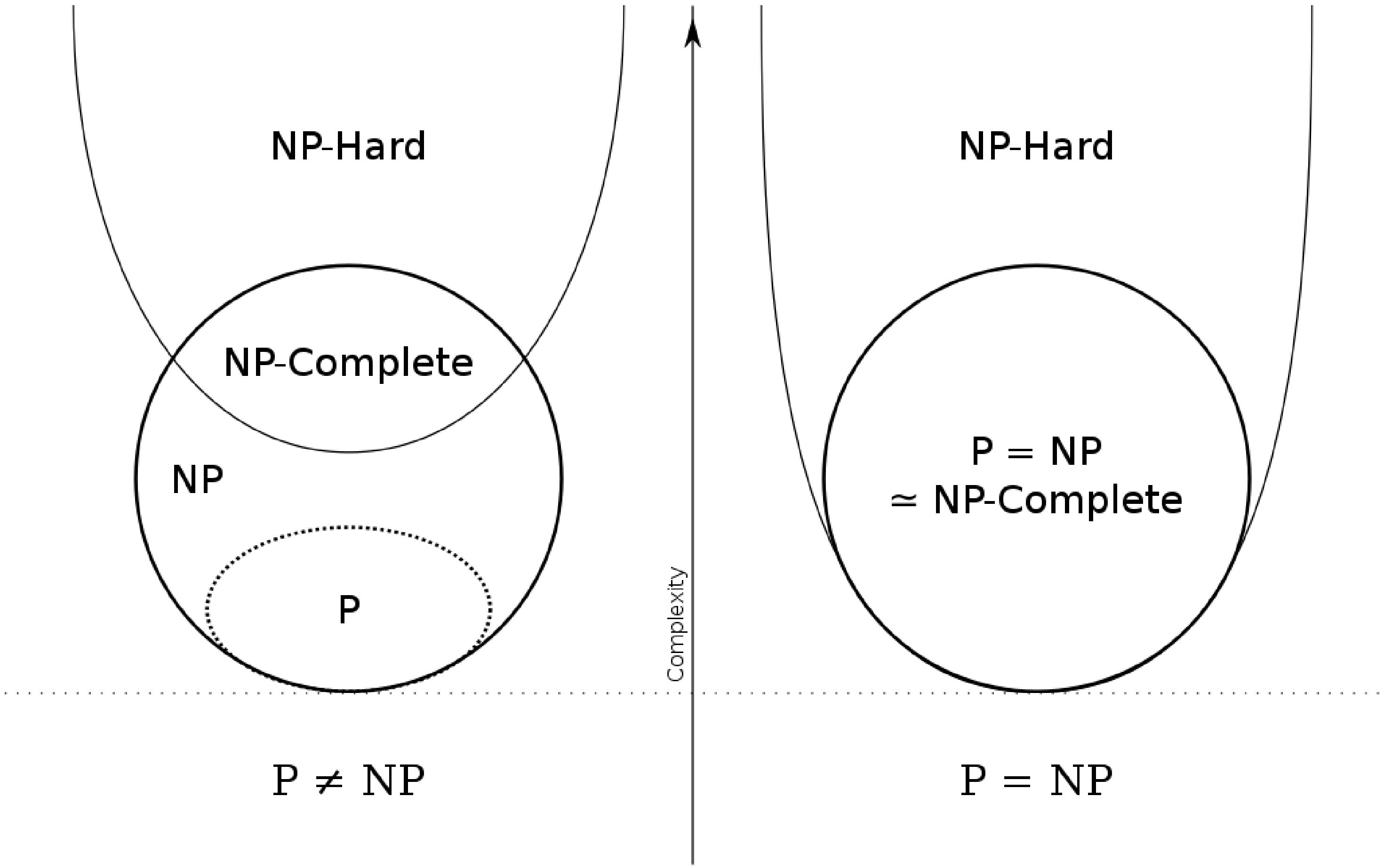
NP-hard: A problem is NP-hard if every problem in NP can be polynomial time reduced to it.

In Practice: To prove a problem X is NP-hard, take a **known NP-complete problem** Y and polynomial time reduce it to X.

Examples of NP-Complete Problems

1. Knapsack decision version
2. 3-Partition: given n integers, can you divide them into triples of equal sum?
3. Traveling Salesman Problem decision version
4. Minesweeper, Sudoku, and most puzzles
5. SAT: given a Boolean formula (and, or, not), is it ever true? x and not x → NO
6. 3-coloring a given graph.

P = NP and P!= NP Scenario



Approximation Algorithm

- If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly.

- but this does not imply that all hope is lost.

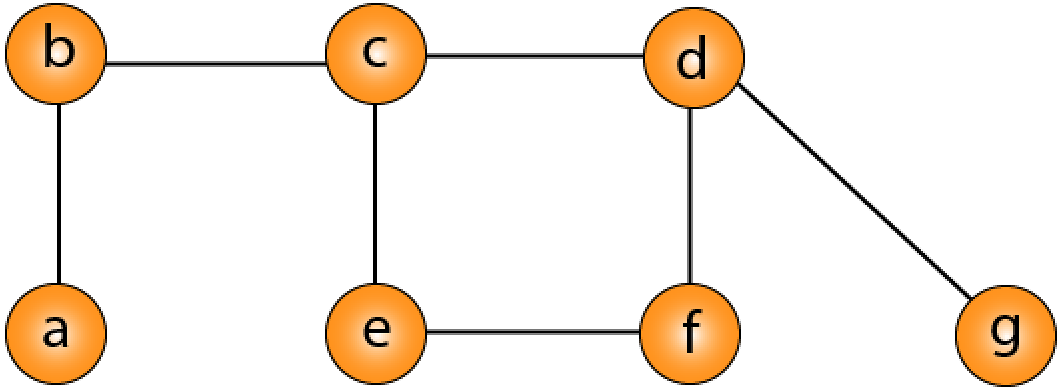
- There are two approaches to getting around NP-completeness.
  - If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.

  - It may still be possible to find near-optimal solutions in polynomial time. In practice, near-optimality is often good enough.

  An algorithm that returns near-optimal solutions is called an **approximation algorithm.**

Approximation Algorithm for Vertex Cover Problem

- Vertex-cover: A vertex cover of undirected graph G = (V, E) is a subset V’ of V such that if <u, v> is an edge in G, then either u or v in V’ (or both). The size of vertex cover is number of vertices contained in V’
- The ***vertex-cover problem*** is to find a vertex cover of minimum size.
- We call such a vertex cover an ***optimal vertex cover***. This problem is NP-hard, since the related decision problem is NP-complete.



V1=V V2 = {b, c, d, f, g} V3={ c, d, e, f, g}
V4 ={a, b, c, e, f} V5={c, d, b, f}
V6={b, e, d} V7={c, d, e, a}

Vertex Cover: V1, V2, V5, V6 V7
Min. Vertex Cover: V6
Not vertex cover: V3 and V4

Approximation Algorithm for Vertex Cover

Simple approach (Nixon's Algorithm): Repeatedly select a vertex of highest degree, and remove all of its incident edges

Using this approach for the graph given in previous slide we get

a, b, c, d, e, f, g

Degree[]=[1, 2, 3, 3, 2, 2, 1]

First select c or d: V'={c}

Updated degree[]=[1, 1, 0, 2, 1, 2, 1]

Select d or f: V'={c, d} updated degree[] = [1, 1, 0, 0, 1, 1, 0]

Select any one vertex arbitrarily from a, b, e, f: V'={c, d, a}

Updated degree[]=[0, 0, 0, 0, 1, 1, 0] Select any vertex e or f : V'={c, d, a, e}

Complexity : $O(n^2)$ [adjacency matrix representation]

Selecting highest degree vertex: $O(n)$, even if all vertices selected complexity of this step is $O(n^2)$

Updating degree: $O(n)$ for each vertex, so n vertex $O(n^2)$

Approximation algorithm for Travelling Salesperson Problem(TSP)

Euclidean TSP Approximation Algorithm:

1. Compute a minimum spanning tree T connecting the cities. (Prim’s algorithm)
2. Visit the cities in order of a preorder traversal of T.
3. Get the Hamiltonian cycle and find the tour cost.

Complexity Analysis:

Prim’s Algorithm Complexity : $O(n^2)$ (Adjacency matrix representation)

Preorder Traversal of Tree : $O(n)$

Cost of tour calculation: $O(n)$

Complexity of the approximation algorithm: $O(n^2)$

Thank You