

NP-Completeness

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen



OUTLINE

- Problem Complexity
- P-problem and NP-problem
- NP-hard Problem and NPC Problem
- Common Resolution Strategies for NPC Problem



THE COMPLEXITY OF THE PROBLEM

- **1.1 Introduction**
- **1.2 Polynomial Time**
- **1.3 Decision-making and Optimization Issues**



INTRODUCTION

- Faced with a problem, we **first** consider whether it can be solved with existing algorithms. **Second**, consider whether the existing algorithm is fast enough.
- An algorithm with time complexity is $O(n^k)$, which is completely acceptable. Because it can still get results in a reasonable amount of time with a very large input.
- So how do we define the complexity of the problem?



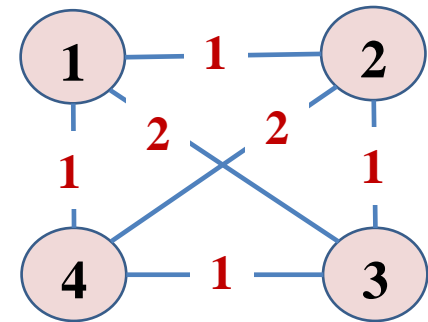
POLYNOMIAL TIME

- We use **polynomial time** limits to define how easy a problem is.
- When we determine whether a question can find an algorithm for polynomial time limits, there are three cases:
 - Yes (just **find** it)
 - No (**prove**)
 - ❑ All algorithms that prove the problem are exponentially timed.
 - ❑ Algorithms that prove that there is no polynomial time limit exist to solve the problem.
 - Not sure (**Not found** at the moment, doesn't mean no, that's the next NPC problem)

POLYNOMIAL TIME

Traveling Salesman Problem (TSP)

- Find a shortest path through all towns and back to the origin, $1 \rightarrow 2 \rightarrow 3$ ($1 \rightarrow 3 \rightarrow 2$) $\rightarrow \dots$. The distance between the two paths is not the same.



- Possible routes are: $n!$
- A simple idea is to traverse all paths and need $O(n!)$ time.
- The time complexity is $O(2^n)$.
- The dynamic programming time complexity is $O(2^n * n^2)$.

$$d(i, V) = \begin{cases} c_{is}, & V = \emptyset, i \neq s \\ \min\{c_{ik} + d(k, V - \{K\})\}, & k \in V, V \neq \emptyset \end{cases}$$

POLYNOMIAL TIME

- The investigation of an algorithm in polynomial time is a "difficult problem" for a computer.
- A problem **can be solved** in a polynomial time, saying it is **easy** to solve.
- A problem **cannot be solved** in polynomial time, saying that the problem is **not easy** to solve.

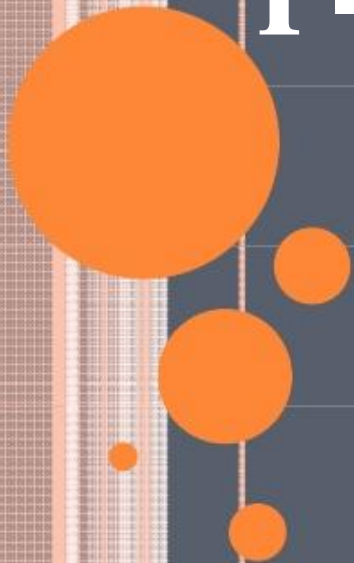


DECISION-MAKING AND OPTIMIZATION ISSUES

- **Decision Question:** A computational problem with the expected output of Yes or No, that is, 1 or 0.
- **Optimization Problem:** We try to construct a solution to **maximize** or **minimize** the computational problem of certain values.
- **Example** (Traveling Salesman Problem, TSP)
 - **Optimization problem:** Given a band map, find a shortest path through all points and back to the origin.
 - **Determine the problem:** Given a weight chart and a number of k , whether there is a path through all points and back to the origin with a total weight less than or equal to k .



P-problem and NP Problems



P-PROBLEM AND NP PROBLEMS

- **2.1 P-problem**
- **2.2 NP-problem**
- **2.3 P-problem & NP-problems**
- **2.4 The meaning of $P = NP$**



P-PROBLEM

- **Question**

- Find the *median* of the sequence $\{3, 1, 2, 4, 7\}$ (the number of $n/2$ positions after sorting)
- *Determine* whether $k-3$ is the median of the above sequence (**thinking**)

- Obviously, an algorithm can be found to solve the above problems in polynomials.



P-PROBLEM

- Definition of a **P** problem: A problem that can be solved in polynomial time is called a **P-problem**.
- What are the other **P** questions?
 - Find the *maximum* value of the sequence of {3, 1, 2, 4, 7}.
 - *Determines* whether $k = 4$ is the maximum value of the above sequence.



NP-PROBLEM

- **Question:** Do mass factor decomposition of 41607317.
- **Thinking:** If you give a possible answer of 8699 and 4783, can you **verify** that these two numbers are correct in a polynomial time?
- **Question:** Bool collection $(x_1, x_2, x_3, x_4, x_5)$, whether there is a set of assignments:
$$(x_1 \parallel x_2 \parallel x_3) \&\& (x_1 \parallel !x_2 \parallel !x_3) \&\& (!x_1 \parallel x_4 \parallel x_5) == \text{true?}$$
- 3-SAT problem



NP-PROBLEM

- **Question:** Bool collection $(x_1, x_2, x_3, x_4, x_5)$, whether there is a set of assignments $(x_1 \parallel x_2 \parallel x_3) \&\& (x_1 \parallel !x_2 \parallel !x_3) \&\& (!x_1 \parallel x_4 \parallel x_5) == \text{true}$? (3-SAT problem)

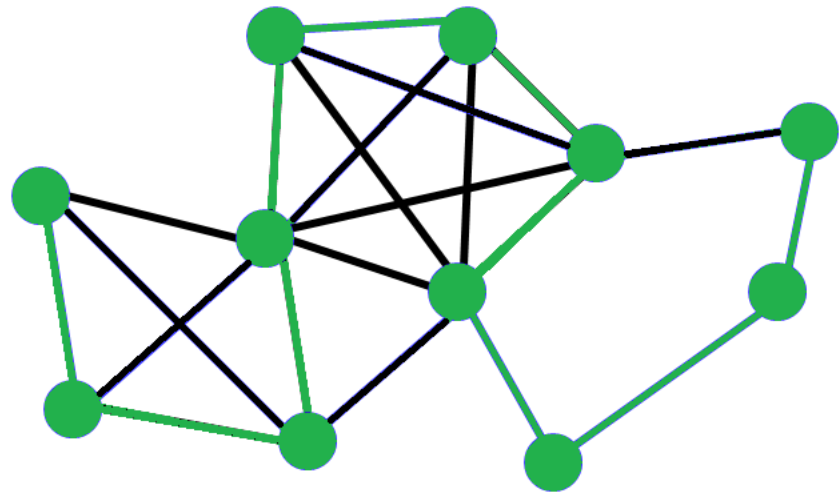
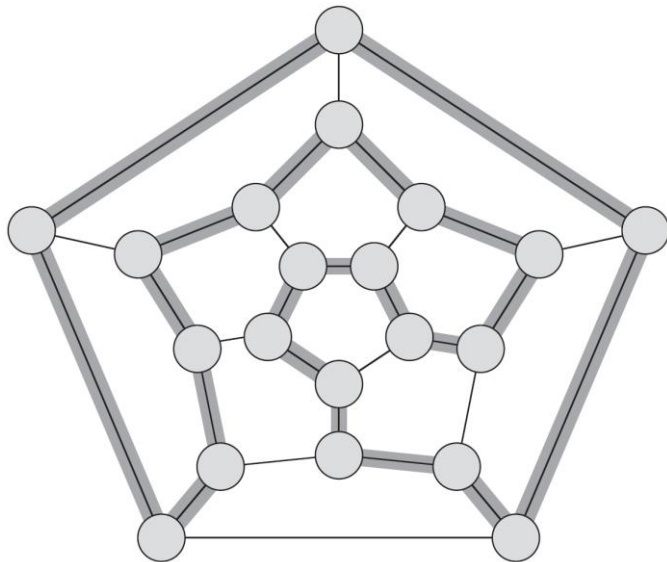
x_1	x_2	x_3	x_4	x_5
0	0	0	0	0
0	0	0	0	1
...

- **Thinking:**

- How to solve the above problems? (2^5)
- If there is given a set of solutions to $x_i = \{0, 1, 1, 0, 1\}$, is there an algorithm to verify the correctness of the solution in polynomial time?
- Whether there is an algorithm to solve the above problem in polynomial time? (Not found so far)

NP-PROBLEM: HAMILTON CIRCUIT PROBLEM

- Hamilton circuit problem
 - Given the directionless graph $G(V, E)$, determine **whether** it passes through the loop of each vertex in the graph **only once**.
 - **Existence:** NP-problem
 - **Nonexistence:** non NP-Problem



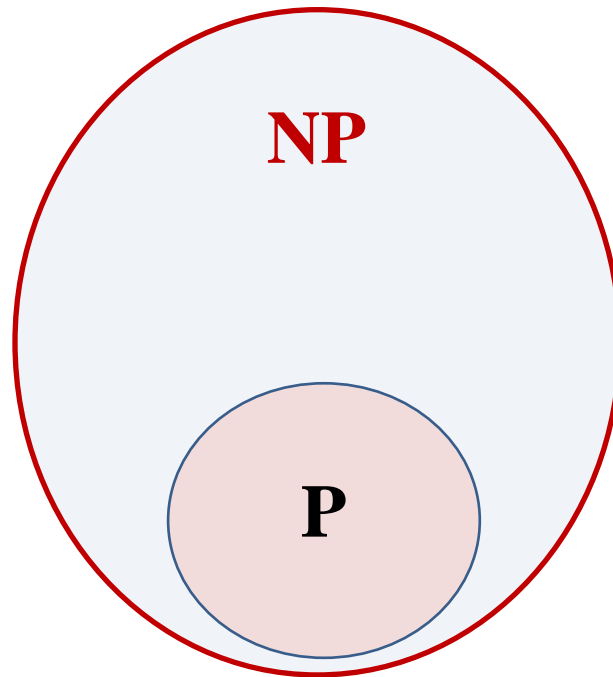
NP-PROBLEM

- **Definition** of an NP problem: A problem that can be **validated** in polynomial time.
- **Thinking**
 - Is the *median* NP problem? (We already know it's a P problem)
 - Is the 3-SAT problem an NP-problem and a P-problem?




P-PROBLEM & NP-PROBLEM

- **Obviously all P-problems must be NP-problems**
 - It can be solved in polynomial time, then it must be verifiable in polynomial time



P-PROBLEM & NP-PROBLEM

- So are NP-problem all P-problems?
 - The *median* is an NP-problem, and it is also a P-problem.
 - 3-SAT is an NP-problem, is it a P-problem?
- There is currently no evidence that $P=NP$, which is still an open question.
- **The meaning of $P=NP$**
 - If NP is verified, a **problem** can be solved in polynomial time, and an **algorithm** can be found to solve the problem in polynomial time.
 - Computers become more "smart" and can quickly find a  shortcut to a very chaotic situation.

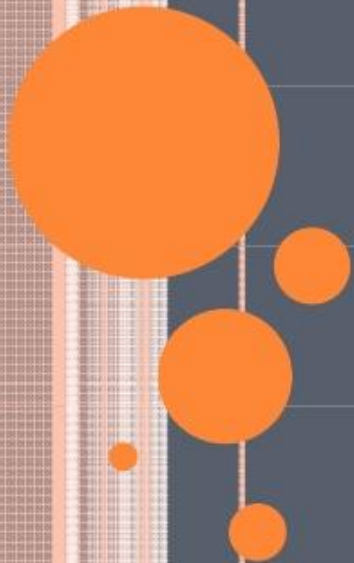
THE MEANING OF $P=NP$

- 1. Many of the challenges in operational planning will be easily solved, such as **integer planning**, traveling salesman, and so on.
- 2. When all the information is available, the computer can make very accurate predictions about the **stock market**, weather, and the results of the game in a very short period of time.
- 3. The issue of **protein folding** is also a NPC problem, and the new algorithm is undoubtedly a boon for the biological and medical community.

.....



NP-hard & NPC



NP-HARD & NPC


- 3.1 Polynomial time reduction
- 3.2 NP-hard & NPC
- 3.3 The distinguish of P, NP, NP-H, NP-C
- 3.4 The meaning of NPC problem



POLYNOMIAL TIME REDUCTION

- **Question A:** Solve a unitary equation at once
- **Question B:** Solve a one-dimensional secondary equation

$$ax + b = c \xrightarrow{\text{Reduce}} 0x^2 + ax + b = c$$

- If there is an algorithm that can effectively solve problem **B**, it can also be used as a sub-program to solve problem **A**, we call problem **A** "reduce" to problem **B**.
 - If I could turn an instance of problem **A** into an instance of problem **B** and then indirectly solve problem **A** by solving problem **B**, then I think **B** is more difficult than **A**.
- 

POLYNOMIAL TIME REDUCTION

- Consider a determination question A and hope to resolve the issue in a polynomial time, while there is a different determination question B , and we know how to resolve it in polynomial time.
- Finally, suppose there is a process that converts any instance α of A into an instance of B that has the following characteristics β :
 - The conversion operation takes polynomial time
 - The solution for both instances is the same: that is, the solution α is "Yes" and the solution that is only β is "Yes".



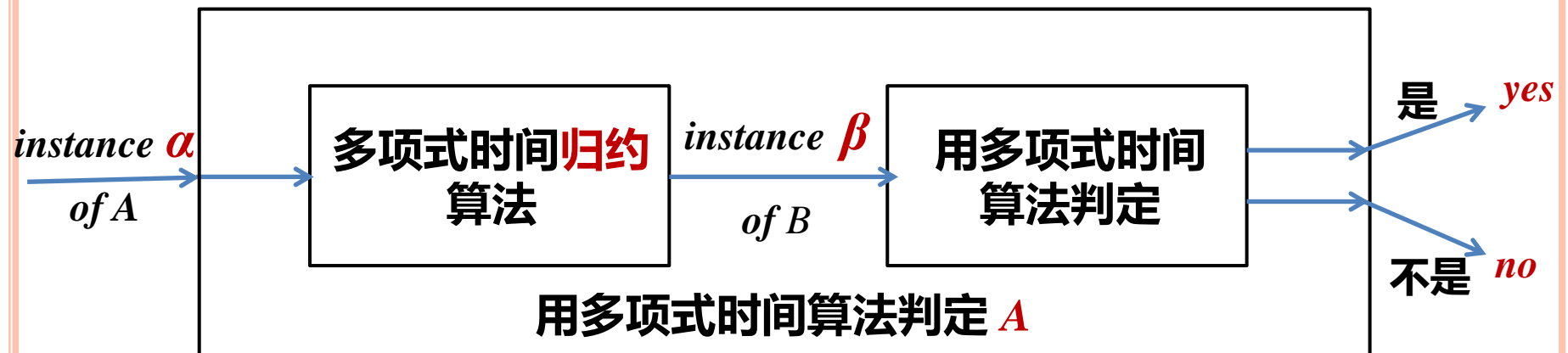
POLYNOMIAL TIME REDUCTION ALG.

- We call *this process* as a polynomial time contracting algorithm. We can solve any example of problem *A*, as shown in the following figure:
 - 1. Given an instance α of problem *A*, use a polynomial-time reduction algorithm to transform it to an instance β of problem *B*.
 - 2. Run the polynomial-time decision algorithm for *B* on the instance β .
 - 3. Use the answer for β as the answer for α .



POLYNOMIAL TIME REDUCTION ALG.

- We call *this process* as a polynomial time contracting algorithm. We can solve any example of problem *A*, as shown in the following figure:



- In other words, by “reducing” solving problem *A* to solving problem *B*, we use the “easiness” of *B* to prove the “easiness” of *A*.

POLYNOMIAL TIME REDUCTION ALG.

- ***Hamilton Circuit Problem***: Given the directionless graph $G(V, E)$, determine whether it passes through the loop of each vertex in the graph only once.
- ***Traveling Salesman Problem***: Find a shortest path through all towns and back to the origin, $1 \rightarrow 2 \rightarrow 3$ ($1 \rightarrow 3 \rightarrow 2$) $\rightarrow \dots$. The distance between the two paths is not the same.
- ***Thinking***: How do you put the HCP problem down to a traveler's problem?



POLYNOMIAL TIME REDUCTION ALG.

- Hamilton Circuit Problem can be reduced to Travelling Salesman Problem
 - In the Hamilton Circuit Problem, we assume that two points are connected, *i.e.*, the distance between the two points is **0**.
 - The distance between the two points is not directly connected to **1**, so the problem translates into whether there is a path with a length of 0 in the TSP problem.
 - Hamilton loops exist when and only if there is a loop with a length of **0** in the TSP problem.



POLYNOMIAL TIME REDUCTION ALG.

- By making a **contract** on some problems, we can find algorithms with high complexity but wide application to replace algorithms with low complexity but can only be applied to a very small class.



NP-HARD & NPC

- Informally, a problem is in the class **NPC**, if it is in NP and is as “**hard**” as any problem in NP.
- **NP-hard**: All NP problems can be attributed to **question** D within the polynomial time complexity, and then the problem D becomes the **NP-hard** problem.
- **NPC** (NP-completeness): All NP problems can be attributed to **NP** problem D within polynomial time complexity, then **NP** problem D becomes **NPC** problem.
- NPC question D should satisfy the following **two things**
 - The problem D is NP-hard
 - The problem D belong to NP-problem




NP-HARD & NPC

- **NPC problem:** As a general saying, a problem can be **verified** in polynomial time, but **no** polynomial time algorithm has been found to solve this problem, we call this kind of problem **NPC problem**.
- **The first proven NPC problem:** 3-SAT problem
- **The Hamilton Circuit Problem** is also an NPC problem.
- **Prove** that an NP-problem D is an NPC problem, just **reduce** a known NPC problem to question D in polynomial time. Then, D is also an NPC problem.

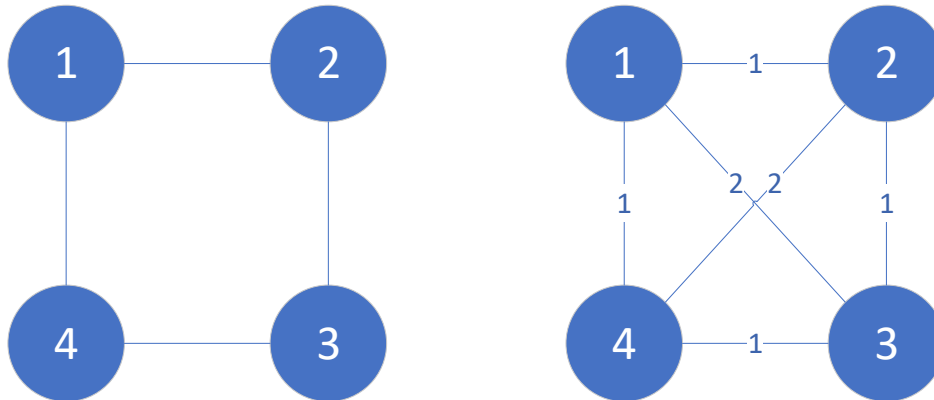


NP-HARD & NPC

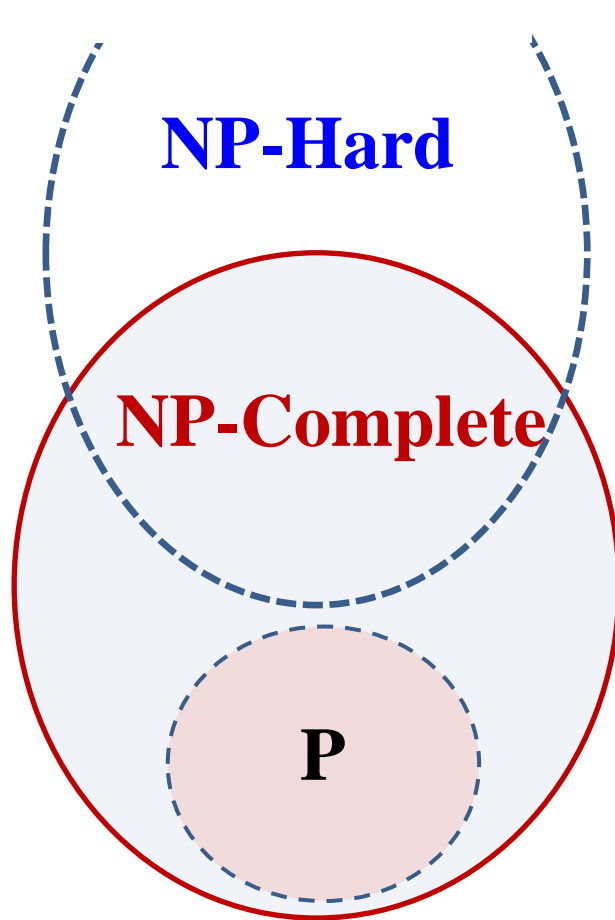
- **Prove** that an NP-problem D is an NPC problem, just **reduce** a known NPC problem to question D in polynomial time. Then, D is also an NPC problem.
 - Please prove that TSP is an NPC issue (HCP is known to be an NPC issue).
 - **Construct the reduction function T**
 - Given a unweighted graph G (HCP) and weighted complete graph G' which has same vertex (TSP).
 - If there are an edge between any two nodes in G , the weight of the edge between the two corresponding nodes in G' is **0**, otherwise it is **1**.
 - Then make $K = 0$.
- 

NP-HARD & NPC

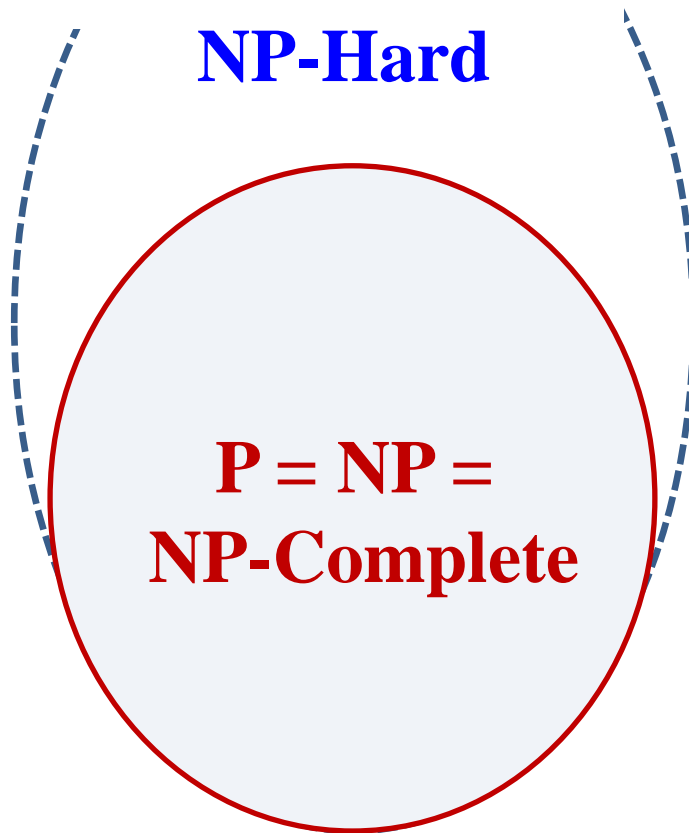
- The reduction function T can be implemented in polynomial time.
- The TSP problem itself is an NP problem.
- So the TSP problem is also an NPC problem.



THE DISTINGUISH OF P, NP, NP-H, NP-C



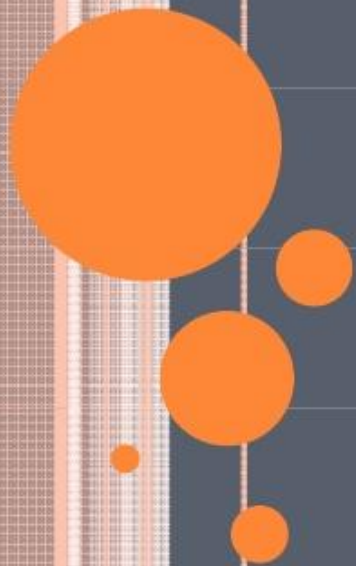
$P \neq NP$



$P = NP$



Common Resolution Strategies for NPC Issues



COMMON RESOLUTION STRATEGIES

- 4.1 Exponential algorithm
- 4.2 Approximate algorithm



EXPONENTIAL ALGORITHM

- The exponential algorithms commonly used to solve NPC problems are:
 - Dynamic planning
 - Branch boundary method
 - **Backtracking**



THE CONCEPT OF BACKTRACKING

- The backtracking algorithm is actually an enumerated search attempt, mainly in the search attempt to find the solution to the problem, when found that the solve conditions have not been met, "backtrack" back to try other paths.
- Many complex and large-scale problems can use the backtracking method, which has the reputation of "*universal problem-solving method*".



THE BASIC IDEA OF BACKTRACKING

- In the solution space **tree** that contains all the solutions to the problem, the solution space **tree** is explored in depth from the root node according to the strategy of **depth-first search**.
- When exploring a node, first determine whether the node contains a solution to the problem,
 - If so, continue to explore from that node.
 - If the node does not contain a solution to the problem, then backtrack to its ancestors layer by layer.
- In fact, backtracking is a depth-first search algorithm for implicit diagrams.



THE BASIC STEPS OF BACKTRACKING

- 1. Determine the space for solving the problem: The solution space for the problem should contain **at least one optimal solution** to the problem.
- 2. Determines the extended search rules for nodes.
- 3. Search for solution space in **depth-first manner** and use pruning functions to avoid **invalid** searches during the search.



EXAMPLES OF BACKTRACKING

— 0-1 BACKPACK PROBLEM

- There are N items and a backpack with a capacity of C . The volume of the i -th item is $c[i]$, and the value is $v[i]$.
- Find out which items are loaded into the backpack so that the total cost of these items does not exceed the capacity of the backpack, and the total value is the largest.
- This is the most basic knapsack problem, in general: to choose or not to choose, this is a question.



EXAMPLES OF BACKTRACKING

— 0-1 BACKPACK PROBLEM

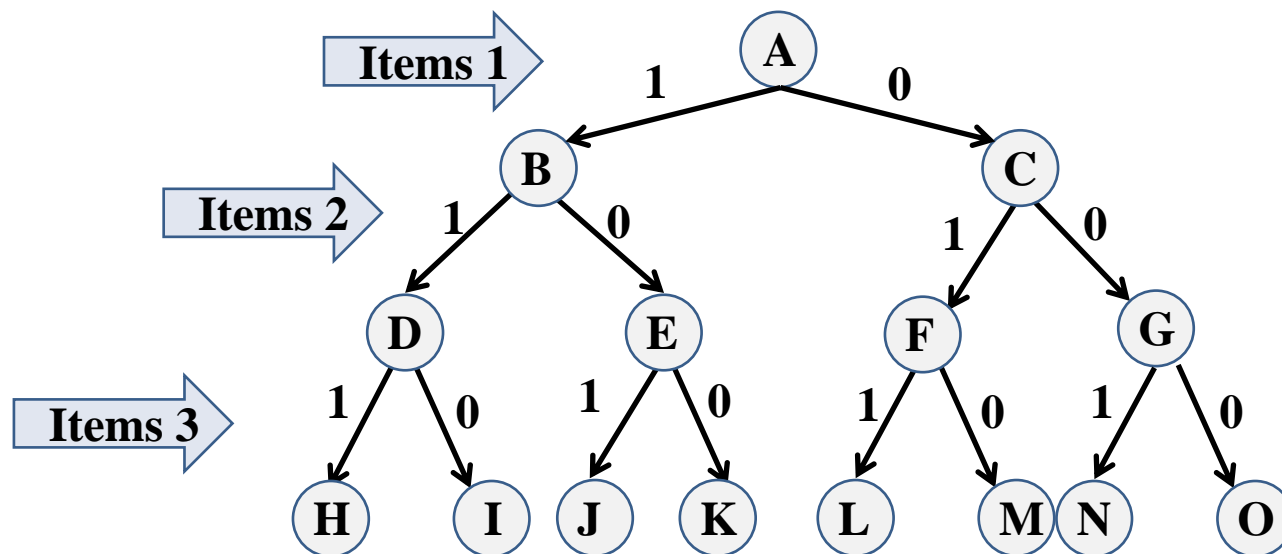
- Let $f[i][c]$ denote the maximum value that the *first i items* (top i -th goods) can get in a backpack with a capacity of C .
- For an item, there are only two cases
 - Case one: The i -th item is not put in, and the value obtained is: $f[i-1][C]+0$.
 - Case two: The i -th item is put in, and the value is: $f[i-1][C-c[i]]+v[i]$.



EXAMPLES OF BACKTRACKING

— 0-1 BACKPACK PROBLEM

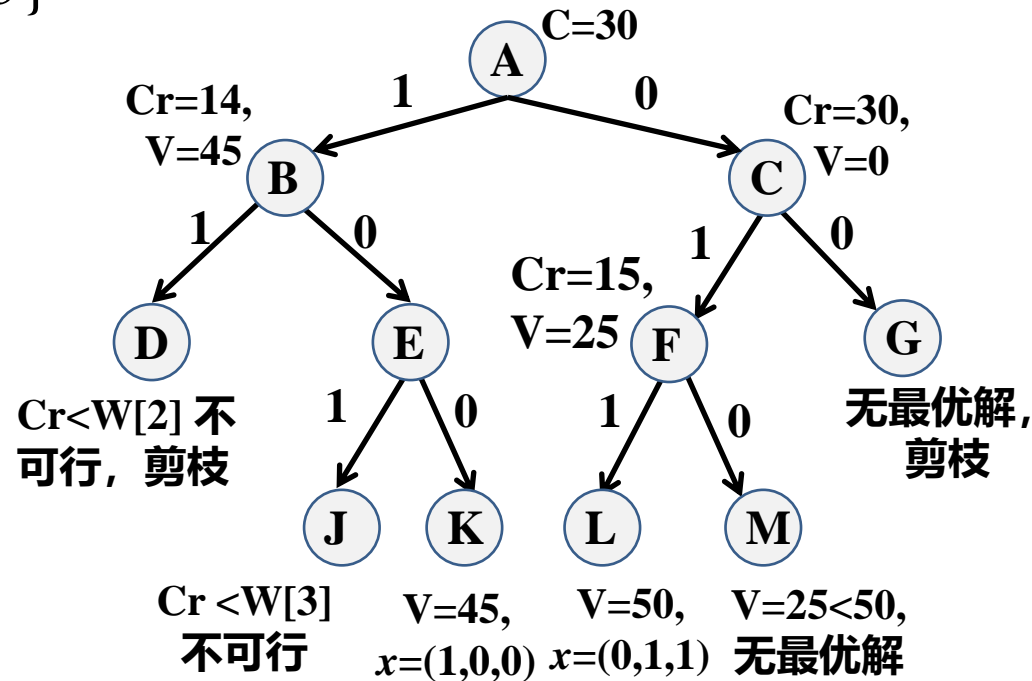
- When solving a problem with backtracking, the solution space of the problem should be clearly defined.
- The solution space for the problem contains at least one optimal solution to the problem. For the 0/1 backpack problem at $n=3$, a complete binary tree can be used to represent the solution space, as shown in the figure:



EXAMPLES OF BACKTRACKING

— 0-1 BACKPACK PROBLEM

- Assume that $n=3$, $C=30$ (maximum capacity), $c = \{16, 15, 15\}$, $v = \{45, 25, 25\}$



- Only 4 paths are finally searched, which is reduced by half compared to the original 8 paths in the solution space.
- It has been optimized to a certain extent, but its complexity is still $O(2^n)$.

APPROXIMATE ALGORITHM

- Approximate the design idea of the algorithm
 - Replace the optimal solution with the approximate optimal solution in exchange for:
 - ▣ Simplification of algorithm design
 - ▣ Reduced time complexity
 - Approximate algorithms are feasible:
 - ▣ The **input data** for the problem is approximate.
 - ▣ The **solution** to the problem allows for a degree of error.
 - ▣ The approximation algorithm can get an approximate solution to the problem in a **relatively short period of time**.



IDEAS AND APPROXIMATE ALGORITHM

- A measure of the performance of an approximate algorithm
 - **Time complexity:** Must be polynomial
 - **Approximate degree of solution:** Approximate ratio
- **Approximate ratio:** If the optimal value for an optimization problem is c^* , and an approximate optimal value is c , the approximate ratio of the approximation algorithm is:

$$\eta = \max \left\{ \frac{c}{c^*}, \frac{c^*}{c} \right\} \quad \begin{array}{l} \text{for minimize problem, } c \geq c^* \\ \text{for maximize problem, } c \leq c^* \end{array}$$

$\eta > 1$, the smaller η , the better approximate better.

- Typically, this approximation ratio is a function of the input size of the problem $\rho(n)$: $\max \left\{ \frac{c}{c^*}, \frac{c^*}{c} \right\} \leq \rho(n)$.
- If an algorithm's approximate ratio is reached $\rho(n)$, the algorithm is called $\rho(n)$ Approximate algorithm.



IDEAS AND APPROXIMATE ALGORITHM

- Approximate **the relative error** of the algorithm λ :

$$\lambda = \left| \frac{c - c^*}{c^*} \right|$$

- λ denotes the **degree** to which an approximate optimal solution **differs from** an optimal solution.

- If the input size of the problem is n , and exist a function $\varepsilon(n)$ where

$$\left| \frac{c - c^*}{c^*} \right| \leq \varepsilon(n)$$

$\varepsilon(n)$ is called the relative error bound of the approximation algorithm, and

$$\varepsilon(n) \leq \rho(n) - 1.$$



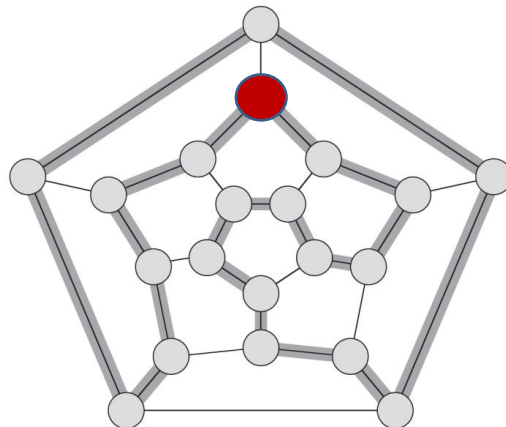
EXAMPLES OF APPROXIMATE ALGORITHMS

—VERTEX COVERAGE

- **Question**

- The vertex cover of the undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of its vertex set V , so that if (u, v) is an edge of G , then $v \in V'$ or $u \in V'$.
- The size of the vertex cover V' is the number of vertices it contains $|V'|$.

- **Vertex cover problem:** find the smallest vertex cover.



EXAMPLES OF APPROXIMATE ALGORITHMS

—VERTEX COVERAGE

- Vertex cover problem: find the smallest vertex cover

VertexSet ApproxVertexCover (Graph G)

```
{  Cset =  $\emptyset$ ;  
  E1 = E;  
  while (E1 !=  $\emptyset$ ) {  
    Take any edge e=(u,v) from E1 ;  
    Cset=Cset  $\cup$  {u,v};  
    Delete all edges associated with u and v from E1;  
  }  
  return Cset;  
}
```

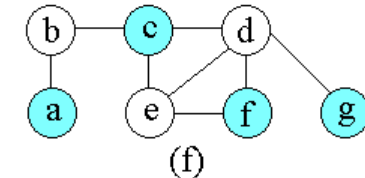
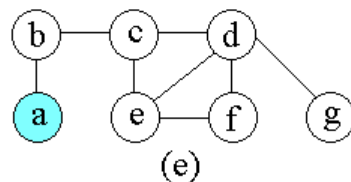
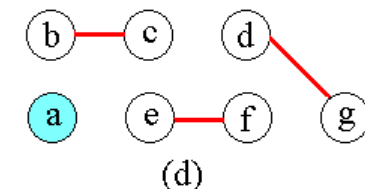
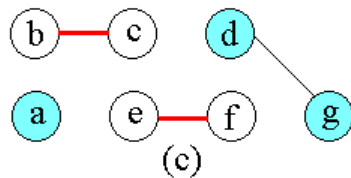
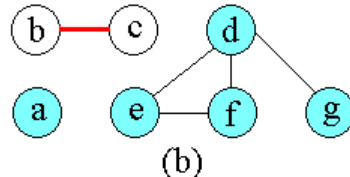
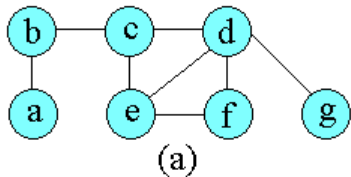
Cset is used to store the vertices in the vertex cover. **Initially empty**, continue to select one side (u, v) from the edge set E1, **add** the end points of the edge to Cset, and **delete** the edges covered by u and v in E1 **until** Cset has covered all edges. That is, E1 is empty.

- What is the time complexity of the algorithm? $O(V+E)$.



EXAMPLES OF APPROXIMATE ALGORITHMS

—VERTEX COVERAGE



Figures (a) ~ (e) illustrate the running process and results of the algorithm.


(e) represents the approximate optimal vertex **cover** Cset generated by the algorithm, which is composed of vertices *b*, *c*, *d*, *e*, *f*, and *g*.

(f) is a **minimum vertex cover** of graph G, which contains only 3 vertices: *b*, *d* and *e*.

- The approximate ratio of the algorithm **ApproxVertexCover** is **2**. ($c=6$ (e), $c^*=3$ (f))

EXAMPLES OF APPROXIMATE ALGORITHMS

—VERTEX COVERAGE

- **Theorem:** The algorithm ApproxVertexCover is a 2 approximation algorithm.
 - **Proof:** Let A denote a set of **edges** selected by the algorithm ApproxVertexCover. Let C^* denote the **optimal vertex cover**.
 - Then, C^* must contain at least one **endpoint** of each side in A . **Also**, since no two edges in A share endpoints, no two edges in A are covered by the same vertex of C^* .
 - Therefore, we have a lower bound on the scale of the optimal vertex cover: $|C^*| \geq |A|$.
 - On the other hand, the algorithm selects the two endpoints of each edge in A , with $|C| = 2|A|$.
 - **Therefore**, $|C| = 2|A| \leq 2|C^*|$.
- 

Ending

