UNIT-5: Branch and Bound and NP-Completeness

Table of Contents

Course Topics

This unit covers:

- **Branch and Bound**: General Method and applications including Travelling Salesperson Problem and 0/1 Knapsack Problem with LC Branch and Bound Solution and FIFO Branch and Bound Solution
- **NP-Hard and NP-Complete Problems**: Basic concepts, non-deterministic algorithms, NP-Hard and NP-Complete classes, and Cook's theorem

Part 1: Branch and Bound Algorithm

1.1 General Method

The **Branch and Bound Algorithm** is a method used in combinatorial optimization problems to systematically search for the best solution. Branch and Bound is commonly used in problems like the Travelling Salesman and job scheduling.

Key Characteristics:

- It involves breaking the problem into smaller sub-problems (branching) and using bounds to eliminate sub-problems that cannot contain the optimal solution (bounding)
- This approach explores a state-space tree where nodes represent sub-problems
- Pruning is done to avoid exhaustive search, thus improving efficiency

General Method Components:

1. **Branching**: Split the problem into smaller subproblems that represent subsets of the solution space
2. **Bounding**: Calculate an upper or lower bound on the objective value for each subproblem
3. **Pruning**: Discard subproblems whose bounds indicate they cannot contain a better solution than already found
4. **Selection**: Explore the remaining subproblems systematically until all possibilities are evaluated or pruned

1.2 Applications of Branch and Bound

The branch and bound method is applied mainly to solve combinatorial optimization problems optimally and efficiently by reducing the search space. Key applications in Design and Analysis of Algorithms include:

1. Travelling Salesman Problem / Traveling Salesperson Problem (TSP)

- Optimally finding the shortest route visiting all cities using systematic state-space tree search and pruning

2. 0/1 Knapsack Problem

- Selecting items to maximize value without exceeding capacity by branching on inclusion/exclusion and bounding via fractional knapsack relaxations

3. Job Scheduling

- Assigning jobs to resources in ways to optimize makespan or minimize cost using branching on job assignments and bounding partial schedules

4. Integer Programming

- Solving discrete optimization problems by branching on integer variable values and using bounds from linear relaxations for pruning

5. Constraint Satisfaction Problems

- Efficient exploration of search space by pruning infeasible partial solutions

6. Resource Allocation Problems

- Optimal distributing limited resources among competing tasks or agents with bounding functions improving search efficiency

2. Traveling Salesperson Problem (TSP)

2.1 Problem Definition

The **Traveling Salesman Problem (TSP)** using the Branch and Bound technique is a combinatorial optimization problem where the goal is to find the shortest possible route that visits a set of cities exactly once and returns to the starting city.

The Branch and Bound algorithm solves this by systematically exploring potential routes in a search tree, while pruning suboptimal paths using a lower bound estimate on the route cost to avoid unnecessary computations.

2.2 Key Points

1. The algorithm starts at the root node representing the initial city
2. At each node in the search tree, it calculates a lower bound on the minimum possible total cost to complete the route from that node
3. If the calculated bound exceeds the cost of the best known solution, that branch is pruned (ignored)
4. Otherwise, the algorithm recursively explores deeper nodes (routes)
5. The cost bound is computed by considering minimum costs of leaving and entering each city to estimate a least possible completion cost
6. The best solution found through the search is the shortest route for the salesman

This approach significantly reduces the search space compared to brute force enumeration, making it more efficient though still computationally expensive for very large numbers of cities.

2.3 Salesman Rules

- Visit each city at least once
- Do not visit any city more than once
- Starting city and ending city must be the same

2.4 Example: TSP with 4 Cities (A, B, C, D)

Step 1: Initial Cost Matrix

	A	B	C	D	Min
A	∞	10	5	3	(3)
B	8	∞	9	7	(7)
C	1	6	∞	9	(1)
D	2	3	8	∞	(2)

Step 2: Row Reduction

Subtract minimum value from each row:

	A	B	C	D
A	∞	7	2	0
B	1	∞	2	0
C	0	5	∞	8
D	0	1	6	∞

Row reduction total = 3 + 7 + 1 + 2 = **13**

Step 3: Column Reduction

For each column, find minimum and subtract from all entries in that column:

Column minimums: A=0, B=1, C=2, D=0
Column reduction total = 0 + 1 + 2 + 0 = **3**

After column reduction:

	A	B	C	D
A	∞	6	0	0
B	1	∞	0	0
C	0	4	∞	8
D	0	0	4	∞

Total reduction = 13 + 3 = 16 (This is our initial lower bound)

This becomes **Matrix M1**.

2.5 Branch and Bound Tree Exploration

Node 1 (Root): Lower Bound = **16**

From Node 1, we branch on three possible paths from A:

Path A→B:

- Set A row to ∞, B column to ∞, B→A to ∞
- Remaining values from M1
- Additional reduction needed = 6
- Total Cost = 16 + 6 + 0 = **22**

Path A→C:

- Set A row to ∞, C column to ∞, C→A to ∞
- Additional reduction needed = 4
- Total Cost = 16 + 4 + 0 = **20**

Path A→D:

- Set A row to ∞, D column to ∞, D→A to ∞
- Additional reduction needed = 0
- Total Cost = 16 + 0 + 0 = **16** ← **Best so far**

Continuing from A→D (Node with cost 16):

Path (A→D)→B:

- Set A, D rows to ∞, B column to ∞, B→D and B→A to ∞
- Additional reduction = 0
- Total Cost = 16 + 0 + 0 = **16**

Path (A→D)→C:

- Set A, D rows to ∞, C column to ∞, appropriate cells to ∞
- Additional reduction = 5 + 4 = 9
- Total Cost = 16 + 9 + 0 = **25**

Continuing from (A→D→B):

Path (A→D→B)→C:

- All remaining cells properly set
- Additional reduction = 0
- Total Cost = 16 + 0 + 0 = **16**

2.6 Branch and Bound Tree Diagram

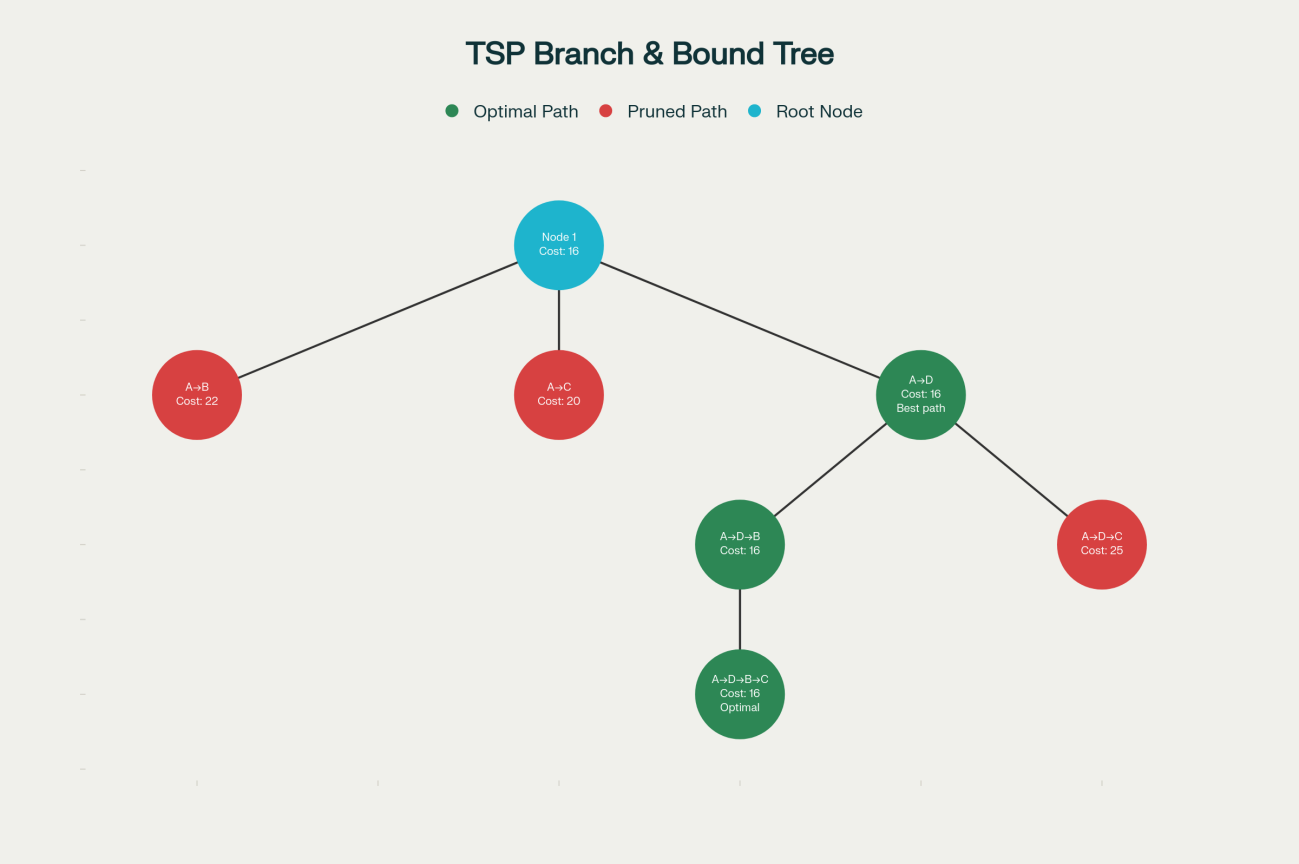

Figure 1: Branch and Bound State Space Tree for TSP - The green path shows the optimal solution: A → D → B → C → A with total cost 16. Red/gray paths represent pruned branches with higher costs.

2.7 Final Solution

Optimal Path: A → D → B → C → A

Path costs:

- A → D: 3
- D → B: 3
- B → C: 9
- C → A: 1

Total Cost = 3 + 3 + 9 + 1 = 16

3. 0/1 Knapsack Problem - Branch and Bound

3.1 Problem Formulation

The **0/1 Knapsack problem** can be efficiently solved using the branch and bound technique, which systematically explores possible item combinations while pruning branches that cannot lead to better solutions based on calculated bounds.

Given:

- A set of n items, each with weight w_i and value v_i
- A knapsack with maximum capacity W
- **Goal**: Maximize the total value in the knapsack without exceeding its weight limit
- Each item can be either **included (1)** or **excluded (0)** – hence the name "0/1 Knapsack"

3.2 Method for Solving the Problem

1. **Sort items** by value/weight ratio in descending order to prioritize more valuable items per unit weight
2. **Start with a root node** representing the empty knapsack
3. **At each node, branch into two**:
    - One where the current item is included (if it fits)
    - One where it is excluded
4. **Calculate the upper bound** for each node, often using a greedy approach, sometimes taking fractions of remaining items for calculation purposes only (not as part of the solution)
5. **Use a priority queue** to repeatedly process the node with the best bound (for LC method) or queue for FIFO method
6. **Pruning**: Whenever a node's bound is worse than the best found feasible solution, it's pruned
7. The process continues until all promising nodes are explored, and the best feasible solution found so far is the answer

3.3 Advantages

- **Efficient** compared to brute-force or basic backtracking since it eliminates large subtrees that cannot yield better solutions
- Especially useful when weights and values are not integers or the state space is too large for dynamic programming

4. Least Cost Branch and Bound Solution

4.1 Concept

In **Least Cost (LC) Branch and Bound**, we use a priority queue that always expands the node with the best (least) bound first. This is also known as **Best-First Search**.

The objective is to place objects into a knapsack so that the profit is maximum.

4.2 Example Problem

Given:

- $n = 4$ items
- Knapsack capacity $m = 15$
- Profits: $P = (10, 10, 12, 18)$
- Weights: $W = (2, 4, 6, 9)$

Step 1: Convert all profits to negative values (for minimization framework)

$$P = (-10, -10, -12, -18)$$

Step 2: Calculate profit-to-weight ratios:

- Item 1: -10/2 = -5.0
- Item 2: -10/4 = -2.5
- Item 3: -12/6 = -2.0
- Item 4: -18/9 = -2.0

Items are already sorted by profit-to-weight ratio (most negative first).

4.3 Node Calculations

For each node, we calculate:

- **Lower Bound (LB)**: No fractional values allowed (actual feasible solution)
- **Upper Bound (UB)**: Fractional values allowed (optimistic estimate)

Node 1 (Root):

Upper Bound:

- Include items: 1, 2, 3, 4 (fractional)
- Weight: 2 + 4 + 6 + 3 (partial of item 4, 3/9 fraction) = 15
- Profit: -10 - 10 - 12 - (18 × 3/9) = -10 - 10 - 12 - 6 = **-38**

Lower Bound:

- Include items fitting completely: 1, 2, 3
- Weight: 2 + 4 + 6 = 12
- Profit: -10 - 10 - 12 = **-32**

Node 2 ($x_1 = 1$):

Including item 1:

- LB = **-38**
- UB = **-32**

Node 3 ($x_1 = 0$):

Excluding item 1:

- LB = **-32**
- UB = **-22**

Node 4 ($x_1 = 1, x_2 = 1$):

Including items 1 and 2:

- LB = **-38**
- UB = **-32**

Node 5 ($x_1 = 1, x_2 = 0$):

Including item 1, excluding item 2:

- LB = **-22**
- UB = **-36**

Node 6 ($x_1 = 1, x_2 = 1, x_3 = 1$):

Including items 1, 2, and 3:

- Weight: 2 + 4 + 6 = 12
- LB = **-38** (can fit partial item 4)
- UB = **-38**

Node 7 ($x_1 = 1, x_2 = 1, x_3 = 0$):

Including items 1 and 2, excluding item 3:

- LB = **-38**
- UB = **-38**

Node 8 ($x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$):

Including items 1, 2, and 4:

- Weight: 2 + 4 + 9 = 15
- Profit: -10 - 10 - 18 = **-38**
- LB = **-38**
- UB = **-38 ★ OPTIMAL SOLUTION**

Node 9 ($x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$):

Including only items 1 and 2:

- Weight: 2 + 4 = 6
- Profit: -10 - 10 = -20
- LB = **-20**
- UB = **-20**

4.4 Branch and Bound Tree Diagram (LC Method)

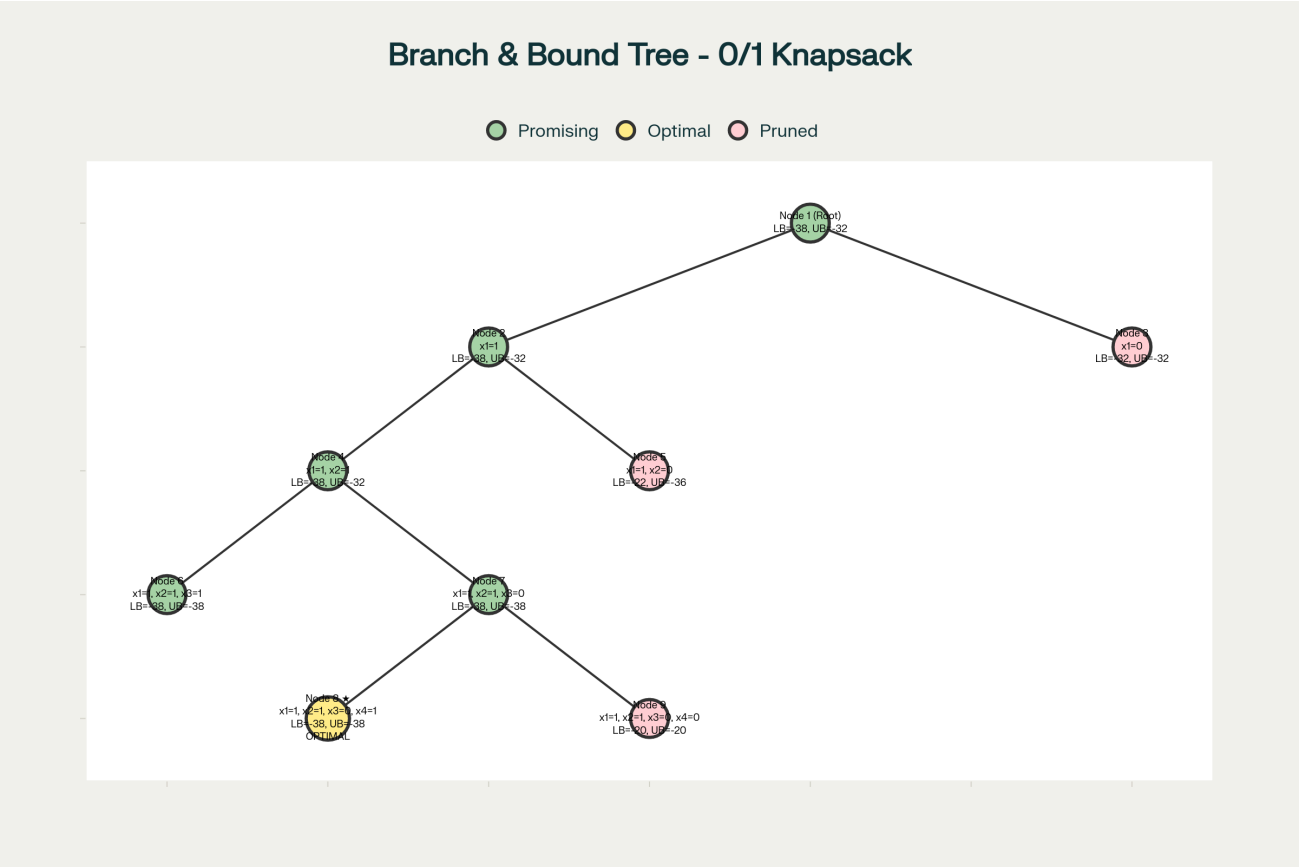

Figure 2: Least Cost Branch and Bound Tree - The tree shows best-first exploration with lower bounds (LB) and upper bounds (UB). The optimal solution is highlighted, selecting items 1, 2, and 4 for maximum profit of 38.

4.5 Solution

Comparing all nodes, the best solution is at Node 8.

Final Path: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$

Elements placed in knapsack: Items 1, 2, and 4

Total profit: 10 + 10 + 18 = **38**

Total weight: 2 + 4 + 9 = **15** (exactly at capacity)

5. FIFO Branch and Bound Solution

5.1 Concept

The **FIFO (First-In-First-Out)** method in Branch and Bound is a strategy used in optimization and combinatorial problems efficiently. It uses a **Breadth-First Search (BFS)** technique.

Key Features:

- Uses a **queue data structure** to manage the list of live nodes (nodes that have been generated but not yet explored)
- When a node is expanded (E-node), all its children are generated and added to the end of the queue
- The oldest node (inserted first) is selected next for exploration, maintaining FIFO order
- Ensures a level-wise (breadth-first) search of the state space tree
- Uses a **global upper bound** that gets updated as better solutions are found

5.2 Key Terms

- **E-node (Expanding node)**: The node currently being explored
- **Live node**: A generated node whose children are yet to be explored
- **Dead node**: A node that is fully explored or pruned using bounding functions
- **Global Upper Bound**: The best feasible solution found so far, used for pruning

The process continues until the optimal or goal node is found or until no live nodes remain.

5.3 Example: FIFO Branch and Bound

Problem Setup (Same as LC example):

- $n = 4$, $m = 15$
- $P = (10, 10, 12, 18)$, $W = (2, 4, 6, 9)$
- Convert to negative: $P = (-10, -10, -12, -18)$

Key Difference: FIFO uses a **global upper bound** that gets updated as we find better solutions, and explores nodes level by level.

5.4 Node Exploration (BFS Order)

Level 0:

Node 1 (Root):

- LB = **-38**
- UB = **-32**
- Global UB = **-32**

Level 1:

Node 2 ($x_1 = 1$):

- LB = **-38**
- UB = **-32**

Node 3 ($x_1 = 0$):

- LB = **-32**
- UB = **-22**
- **Update Global UB to -22** (better feasible solution found)

Level 2:

Node 4 ($x_1 = 1, x_2 = 1$):

- LB = **-38**
- UB = **-32**
- Explore (UB ≤ Global UB)

Node 5 ($x_1 = 1, x_2 = 0$):

- LB = **-22**
- UB = **-32**
- Don't expand (UB > Global UB = -22)

Node 6 ($x_1 = 0, x_2 = 1$):

- LB = **-32**
- UB = **-22**
- Don't expand

Node 7 ($x_1 = 0, x_2 = 0$):

- LB = **-30**
- UB = **-30**
- Don't expand (worse than global)

Level 3:

Node 8 ($x_1 = 1, x_2 = 1, x_3 = 1$):

- LB = **-38**
- UB = **-32**

Node 9 ($x_1 = 1, x_2 = 1, x_3 = 0$):

- LB = **-38**
- UB = **-38**

Level 4:

Node 10 ($x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1$):

- **Not feasible** (weight exceeds capacity: 2+4+6+9=21 > 15)

Node 11 ($x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0$):

- LB = **-32**
- UB = **-32**
- Don't expand (UB > Global UB)

Node 12 ($x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$):

- LB = **-38**
- UB = **-38 ★ OPTIMAL SOLUTION**

Node 13 ($x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$):

- LB = **-20**
- UB = **-20**
- **Update Global UB to -20**

5.5 Branch and Bound Tree Diagram (FIFO Method)

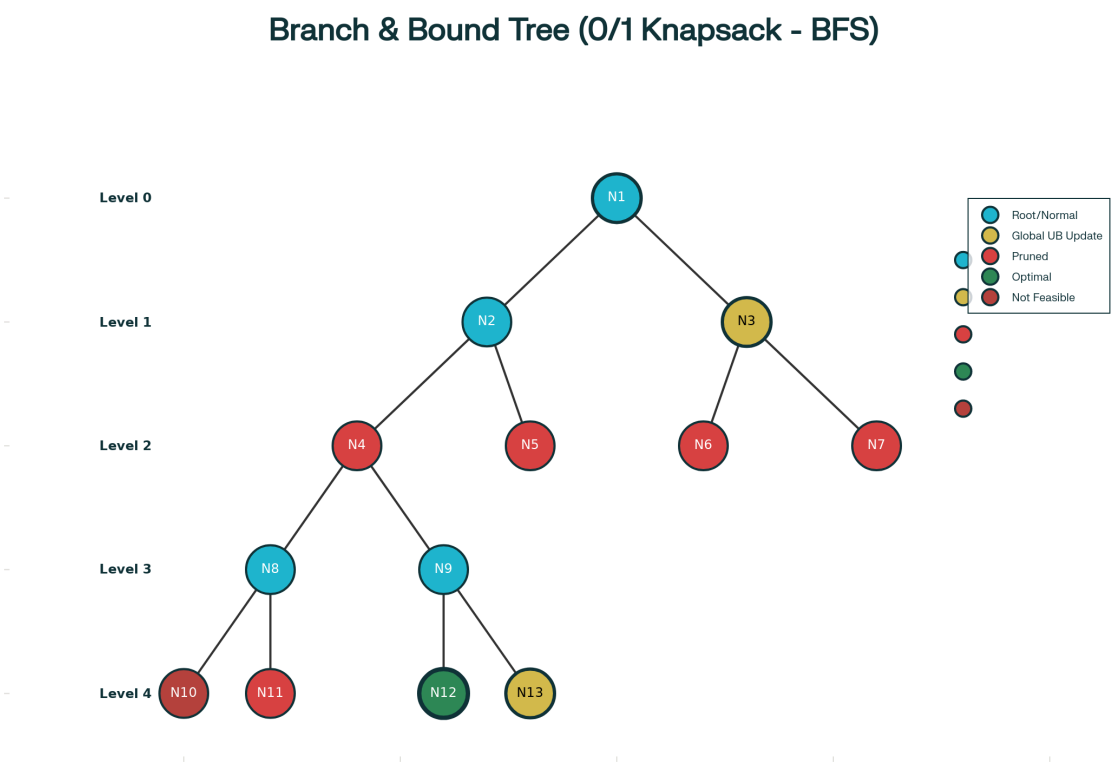

Figure 3: FIFO Branch and Bound Tree - The tree shows breadth-first level-wise exploration with global upper bound updates. Pruned branches are marked, and the optimal solution path is highlighted selecting items 1, 2, and 4.

5.6 Final Solution

After exploring all nodes in FIFO order:

Final Path: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$

Elements in knapsack: Items 1, 2, and 4

Total profit: 10 + 10 + 18 = **38**

Total weight: 2 + 4 + 9 = **15**

Part 2: NP-Hard and NP-Complete Problems

6. Nondeterministic Algorithms

A **nondeterministic algorithm** is one that may produce different outcomes even when given the same input. In theoretical computer science, this concept is modeled using a **nondeterministic Turing machine**, which can make choices between multiple possible next moves.

Comparison:

Deterministic Algorithm:

- Always follows the same sequence of steps for a given input
- Examples: Merge Sort, Binary Search

Nondeterministic Algorithm:

- Can conceptually explore many possible execution paths simultaneously

- Chooses one that leads to a correct answer if it exists

In complexity theory, a nondeterministic algorithm is said to solve a problem in **polynomial time** if there exists at least one computation path that reaches a correct solution in polynomial time.

This model gives rise to the complexity class **NP (Nondeterministic Polynomial time)** — problems whose solutions can be verified quickly (in polynomial time) on a deterministic machine.

7. Complexity Classes P and NP

7.1 Class P (Polynomial Time)

Definition: Problems that can be solved efficiently by a deterministic algorithm.

Characteristics:

- Solutions can be found in polynomial time
- Verification is also polynomial

Example: Sorting a list with Merge Sort

- Time complexity: $O(n \log n)$

7.2 Class NP (Nondeterministic Polynomial Time)

Definition: Problems whose solutions can be verified efficiently once a potential solution is given.

Characteristics:

- Verification is polynomial
- Finding the solution may or may not be polynomial

Example: Subset Sum Problem

- If you are told which subset sums to the target, you can check it easily
- Finding it from scratch might be hard

7.3 The P vs NP Problem

The famous **P vs NP problem** asks whether all problems that have verifiable solutions (NP) are also efficiently solvable (P).

Question: Does P = NP?

This is one of the most important unsolved problems in computer science and mathematics.

8. NP-Hard Problems

8.1 Definition

A problem is **NP-Hard** if all problems in NP can be reduced to it in polynomial time.

8.2 Characteristics

- They are **at least as hard** as the hardest NP problems
- They **do not have to be in NP** themselves (meaning we might not be able to verify their solutions efficiently)
- No known polynomial-time algorithm exists for NP-Hard problems (assuming P ≠ NP)

8.3 Examples

1. **Traveling Salesman Problem (TSP)** - optimization form
    - Finding the shortest tour visiting all cities
2. **The Halting Problem**
    - Determining whether a program will halt on a given input
    - This problem is undecidable (not even solvable in theory)

9. NP-Complete Problems

9.1 Definition

A problem is **NP-Complete** if it satisfies **two conditions**:

1. **It belongs to NP** (its solution can be verified quickly)
2. **It is NP-Hard** (every NP problem can be reduced to it)

Thus, **NP-Complete problems represent the hardest problems within NP**.

9.2 Significance

If any NP-Complete problem can be solved in polynomial time, then P = NP.

This means:

- Solving one NP-Complete problem efficiently would solve all NP problems efficiently
- NP-Complete problems are equally difficult

9.3 Examples

1. **Boolean Satisfiability Problem (SAT)**
    - First proven NP-Complete problem
    - Determining if a Boolean formula can be satisfied
2. **3-SAT**
    - SAT with clauses limited to 3 literals
3. **Hamiltonian Cycle Problem**
    - Finding a cycle that visits each vertex exactly once
4. **Subset Sum Problem**
    - Finding a subset of numbers that sum to a target value

10. Cook's Theorem

10.1 Overview

Cook's Theorem was discovered by Stephen Cook in 1971 and was the first formal proof that an NP-Complete problem exists.

10.2 Statement

"The Boolean Satisfiability Problem (SAT) is NP-Complete."

10.3 Details

SAT asks whether there exists a truth assignment to Boolean variables that makes a formula true.

Cook's Contribution:

- Showed that **every problem in NP can be reduced to SAT in polynomial time**
- This established SAT as the **first NP-Complete problem**
- Proved that every NP problem can be represented as a Boolean formula

10.4 Impact

Cook's Theorem laid the **foundation for reductions**:

Reduction Technique:

- If we can reduce any NP problem to another problem efficiently
- And that second problem is already known to be NP-Complete
- Then the second problem must also be NP-Complete

This technique has been used to prove thousands of problems are NP-Complete.

11. Complexity Class Relationships

Figure 4: Venn diagram showing relationships between complexity classes P, NP, NP-Complete, and NP-Hard. The diagram illustrates that $P \subseteq NP$, NP-Complete is the intersection of NP and NP-Hard, and the P vs NP question asks whether $P = NP$.

11.1 Summary Table

Category	Definition	Example	Verification in Polynomial Time	Solvable in Polynomial Time
P	Problems solvable efficiently	Merge Sort	Yes	Yes
NP	Solutions verifiable efficiently	Subset Sum	Yes	Unknown
NP-Complete	Verifiable & as hard as any NP problem	SAT	Yes	Unknown
NP-Hard	At least as hard as NP problems	TSP (optimization)	Not necessarily	Unknown

11.2 Key Relationships

1. **P ⊆ NP**: All problems in P are also in NP (if you can solve quickly, you can verify quickly)
2. **NP-Complete ⊆ NP**: All NP-Complete problems are in NP
3. **NP-Complete ⊆ NP-Hard**: All NP-Complete problems are NP-Hard
4. **P = NP?**: Unknown - this is the central open question in computer science

Conclusion

This unit covered two major topics in algorithm design and complexity theory:

Branch and Bound provides a systematic approach to solving combinatorial optimization problems by:

- Intelligently exploring the solution space through state-space trees
- Using bounds to prune unpromising branches and avoid exhaustive search
- Applications in TSP, Knapsack, scheduling, and integer programming

- Two main strategies: Least Cost (Best-First Search) and FIFO (Breadth-First Search)

NP-Completeness Theory helps us understand computational complexity:

- Classification of problems by difficulty (P, NP, NP-Complete, NP-Hard)
- The central P vs NP question - one of the most important unsolved problems
- NP-Complete problems as the "hardest" problems in NP
- Cook's Theorem establishing SAT as the first NP-Complete problem
- Reduction techniques for proving NP-Completeness of new problems

Understanding these concepts is crucial for:

- Choosing appropriate algorithms for different problems
- Recognizing when problems are computationally intractable
- Developing approximation algorithms or heuristics for hard problems
- Advancing theoretical computer science and practical algorithm design