AKTU · 3RD YEAR · SEMESTER 5 · BCS503

Design and Analysis of Algorithm

Unit 4 — Dynamic Programming, Backtracking & Branch and Bound

- Dynamic Programming with Examples Such as Knapsack
- All Pair Shortest Paths – Warshal's Algorithms
- All Pair Shortest Paths – Floyd's Algorithms
- Resource Allocation Problem
- Backtracking
- Branch and Bound with Examples Such as Travelling Salesman Problem
- Graph Coloring
- n-Queen Problem
- Hamiltonian Cycles
- Sum of Subsets



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

Dynamic Programming Paradigm

DYNAMIC PROGRAMMING (DP)

An algorithmic paradigm that solves complex optimization problems by breaking them into overlapping subproblems, solving each subproblem once, and storing their solutions to avoid redundant computations.

- Requires Optimal Substructure: Optimal solution to the problem contains optimal solutions to subproblems.
- Requires Overlapping Subproblems: The same subproblems are solved multiple times during computation.
- Bottom-Up Approach (Tabulation): Solves smaller subproblems first and fills a lookup table iteratively.
- Top-Down Approach (Memoization): Recursive approach that stores results of expensive function calls.
- Time complexity is reduced from exponential to polynomial by eliminating re-computation.

0/1 Knapsack Problem using Dynamic Programming

Given item weights $w = [2, 3, 4, 5]$, values $v = [3, 4, 5, 6]$, and knapsack capacity $W = 5$, find the maximum total value.

1. Recurrence relation: $V[i, w] = \max(V[i-1, w], v[i] + V[i-1, w - w[i]])$ if $w[i] \leq w$, else $V[i-1, w]$.
2. Base cases: $V[0, w] = 0$ for all w and $V[i, 0] = 0$ for all i.
3. Item 1 (w=2, v=3): $V[1, 2..5] = 3$.
4. Item 2 (w=3, v=4): $V[2, 3] = \max(3, 4) = 4$, $V[2, 5] = \max(3, 4 + 3) = 7$.
5. Item 3 (w=4, v=5): $V[3, 5] = \max(7, 5 + 3) = 7$.
6. Item 4 (w=5, v=6): $V[4, 5] = \max(7, 6 + 0) = 7$.

Maximum Value = 7 (achieved by selecting items with weights 2 and 3).

Warshall's Algorithm for Transitive Closure

Warshall's algorithm computes the reachability matrix (transitive closure) of a directed graph.

- Transitive closure matrix $W^{(n)}$ has entry 1 if a directed path exists between vertices i and j, else 0.
- Recurrence Relation
  - $W^{(k)}[i,j] = W^{(k-1)}[i,j] \vee (W^{(k-1)}[i,k] \wedge W^{(k-1)}[k,j])$
  - Vertex k acts as an intermediate candidate node.
- Initial matrix $W^{(0)}$ is the binary adjacency matrix of the given graph.
- Time Complexity: $\Theta(V^3)$ where V is the number of vertices.
- Space Complexity: $\Theta(V^2)$ to store the reachability matrix.

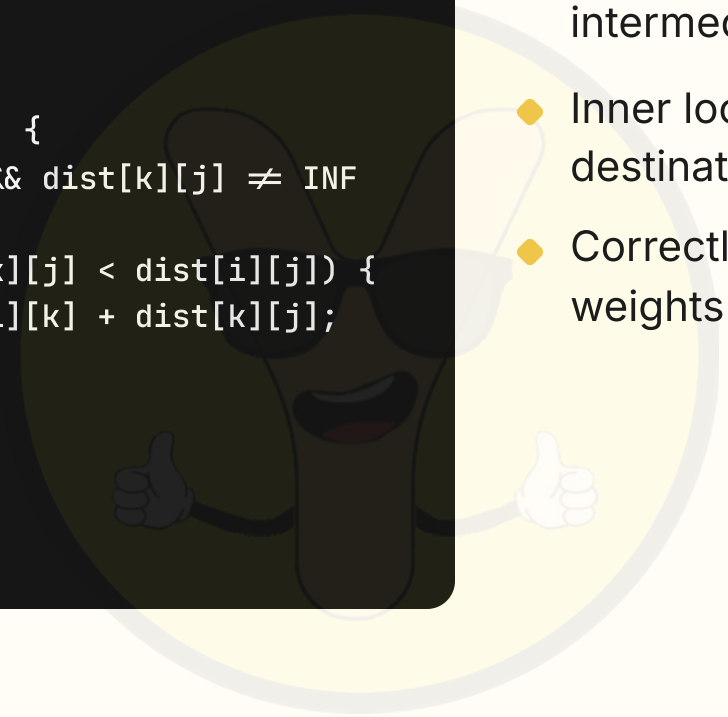
Floyd’s Algorithm for All-Pairs Shortest Paths

Floyd's algorithm finds shortest paths between all pairs of vertices in a weighted graph.

- Works on both directed and undirected graphs without negative weight cycles.
- Recurrence Formula
  - $D^{(k)}[i, j] = \min(D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j])$
  - $D^{(0)}[i, j]$ is initialized to edge weight $w(i, j)$ or ∞.
- Maintains a distance matrix D and a predecessor matrix P for path recovery.
- Time Complexity: $\Theta(V^3)$ using three nested loops.
- Space Complexity: $\Theta(V^2)$ memory footprint.

Floyd-Warshall Implementation in C

```
void floydWarshall(int dist[V][V]) {
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] ≠ INF && dist[k][j] ≠ INF
&&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}
```



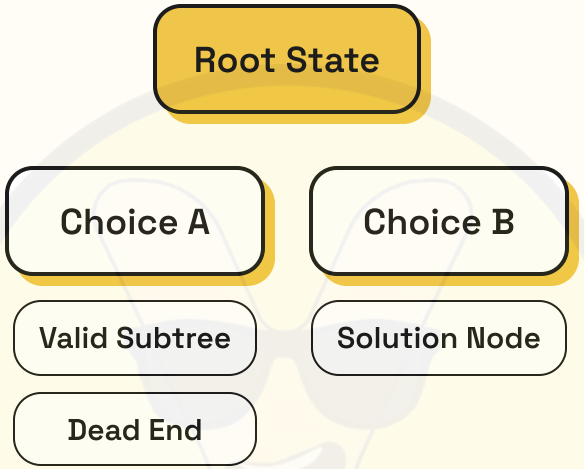
- Outer loop `k` considers vertices as intermediate nodes from 0 to $V - 1$.
- Inner loops `i` and `j` iterate over all source-destination pairs.
- Correctly handles graphs with negative edge weights as long as no negative cycle exists.

Resource Allocation Problem

Distributing resources among activities to maximize total return using Dynamic Programming.

- Problem Statement: Allocate M units of a discrete resource among N activities to maximize total return.
- Let $P(i, j)$ be the profit obtained from allocating j units of resource to activity i.
- DP Recurrence
  - $F_i(x) = \max_{0 \leq k \leq x}\{P(i, k) + F_{i-1}(x - k)\}$
  - $F_i(x)$ is maximum return allocating x units to the first i activities.
- Base Case: $F_1(x) = P(1, x)$ for all $0 \leq x \leq M$.
- Time Complexity: $\mathcal{O}(N \cdot M^2)$, significantly better than brute-force search.

Backtracking Paradigm & State Space Tree



Backtracking systematically explores the state space tree using Depth First Search (DFS), pruning invalid branches.

Algorithmic Paradigm Comparison

Feature	Dynamic Programming	Backtracking	Branch and Bound
Problem Domain	Optimization with overlapping subproblems	Constraint satisfaction & decision problems	Discrete optimization & NP-Hard problems
Search Technique	Bottom-up dynamic table construction	Depth-First Search (DFS) traversal	Breadth-First (BFS) / Best-First Search
Pruning Strategy	Reuses stored subproblem values	Bounding/feasibility rules prune branches	Lower/Upper cost bounds truncate nodes
Data Structures	Arrays / 2D Matrix Tables	Recursion Stack	Priority Queue / Min-Heap
Typical Examples	0/1 Knapsack, Floyd-Warshall	N-Queens, Graph Coloring	Travelling Salesman Problem (TSP)

N-Queens Problem using Backtracking

Place N non-attacking queens on an $N \times N$ chessboard.

- Constraint: No two queens can share the same row, column, or diagonal.
- Representation: Vector $X[1..N]$ where $X[i]$ stores the column position of the queen in row i.
- Diagonal Check Logic
  - Two queens at $(i, X[i])$ and $(j, X[j])$ conflict if:
  - $X[i] == X[j]$ (Same Column) or $|i - j| == |X[i] - X[j]|$ (Same Diagonal)
- Algorithmic Approach: Place queen row by row; if placement fails, backtrack to the previous row.
- Time Complexity: $\mathcal{O}(N!)$ worst case, heavily reduced by early pruning.

4-Queens Problem Worked Walkthrough

Place 4 queens on a 4×4 chessboard using backtracking.

1. Row 1: Place Queen 1 at Col 1 → State: [1, -, -, -]
2. Row 2: Col 1 conflict, Col 2 conflict, place at Col 3 → State: [1, 3, -, -]
3. Row 3: All columns (1 to 4) conflict! Dead End → Backtrack to Row 2.
4. Row 2: Try next available column, Col 4 → State: [1, 4, -, -]
5. Row 3: Place Queen at Col 2 → State: [1, 4, 2, -]
6. Row 4: All columns conflict! Backtrack to Row 1 and place Queen 1 at Col 2.
7. Row 1: Queen at Col 2 → Proceeding yields valid assignment [2, 4, 1, 3].

First valid solution vector X = [2, 4, 1, 3]

Sum of Subsets Problem

Find a subset of positive integers whose sum equals a given target value S.

- Input: Set $W = \{w_1, w_2, \ldots, w_n\}$ and target sum S.
- Assume set elements are sorted in non-decreasing order: $w_1 \leq w_2 \leq \cdots \leq w_n$.
- Bounding Functions
  - Kill child node if current sum $s + w_{k+1} > S$.
  - Kill child node if total remaining sum $s + \sum_{i=k+1}^{n} w_i < S$.
- State Space Tree: Binary tree where left edge includes element w_k and right edge excludes w_k.
- Time Complexity: $\mathcal{O}(2^n)$ worst case, significantly pruned using bounding bounds.

Graph Coloring Problem

M-COLORING PROBLEM

Assigning one of m colors to each vertex of a graph such that no two adjacent vertices share the same color.

- Chromatic Number $\chi(G)$: The minimum number of colors needed to color graph G.
- Decision Problem: Is the graph m-colorable for a given integer m?
- Optimization Problem: Find the exact chromatic number $\chi(G)$.
- Backtracking Strategy: Assign colors recursively to vertices from 1 to V.
- Safety check: Ensure no adjacent neighbor shares the newly assigned color.

Hamiltonian Cycles Problem

Find a closed loop in a graph that visits every vertex exactly once and returns to the start.

- Hamiltonian Path: A path that visits every vertex in the graph exactly once.
- Hamiltonian Cycle: A Hamiltonian path that forms a closed loop (edge exists from last vertex to first).
- Backtracking Validity Conditions
  - Vertex $x[k]$ must be adjacent to previous vertex $x[k-1]$.
  - Vertex $x[k]$ must not already be present in path $x[1..k-1]$.
  - For final vertex $x[n]$, an edge must connect $x[n]$ to starting vertex $x[1]$.
- NP-Complete decision problem with practical uses in routing and circuit design.
- Worst-case Time Complexity: $\mathcal{O}(N!)$.

Branch and Bound Paradigm

A systematic search technique for optimization problems that bounds subproblem values.

- Used exclusively for discrete optimization problems (minimization or maximization).
- Maintains a Global Upper/Lower Bound to truncate unpromising subtrees.
- Search Strategies
  - FIFO Branch & Bound: State space tree explored using a Queue (BFS).
  - LIFO Branch & Bound: State space tree explored using a Stack (D-search).
  - LC (Least Cost) B&B: Explores node with best cost metric using a Priority Queue.
- Pruning Condition: Kill node if lower bound $\geq$ current best upper bound (for minimization).

Branch and Bound Search Strategies

Parameter	FIFO Branch & Bound	LIFO Branch & Bound	LC (Least Cost) B&B
Data Structure	Queue	Stack	Priority Queue (Min-Heap)
Search Order	Breadth-First Search	Depth-First Search	Best-First Search
Selection Rule	First generated active node	Last generated active node	Node with minimum cost estimate $\hat{c}(x)$
Memory Demand	High (stores entire level)	Low (proportional to depth)	Moderate to High
Efficiency	Uninformed Search	Uninformed Search	Most efficient for optimal solution

Travelling Salesman Problem (TSP) using B&B

Find the minimum cost Hamiltonian cycle in a weighted cost matrix using LC Branch and Bound.

1. Step 1: Reduce initial matrix by subtracting row and column minimums to compute initial lower bound L_0.
2. Step 2: From root node (City 1), generate child nodes for unvisited cities.
3. Step 3: Set assigned row/col entries to ∞, reduce matrix, and calculate cost $L_i = L_{prev} + cost(u, v) + \text{reduction}$.
4. Step 4: Select node with lowest cost bound for further expansion (Least-Cost rule).
5. Step 5: Repeat expansion until a full tour is completed and non-explored bounds exceed tour cost.

Returns exact optimal TSP tour while pruning large portions of the state space tree.

Algorithmic Technique Selection Guide

Guidelines for analyzing and selecting algorithmic strategies in AKTU examinations.

- Use Dynamic Programming when subproblems overlap and optimal substructure exists.
- Use Backtracking for combinatorial constraint satisfaction problems (e.g., N-Queens, Sudoku).
- Use Branch and Bound for NP-Hard optimization problems (e.g., TSP, 0/1 Knapsack optimization).
- Dynamic Programming builds solutions bottom-up; Backtracking and Branch & Bound build top-down state trees.
- Branch & Bound uses cost bounds to prune, while Backtracking relies on feasibility constraints.

Exam Corner: High-Yield Questions

- **Q1** Compare Dynamic Programming, Backtracking, and Branch & Bound paradigms with suitable examples.
- **Q2** Explain Warshall's and Floyd's algorithms. Solve all-pairs shortest paths for a given 4-vertex weighted graph.
- **Q3** Formulate the 0/1 Knapsack problem using Dynamic Programming. State its time and space complexity.
- **Q4** Write a backtracking algorithm for the N-Queens problem. Draw the state space tree for N = 4.
- **Q5** Explain the Sum of Subsets problem using Backtracking. Illustrate with a set W = {3, 5, 6, 7} and S = 11.
- **Q6** Describe LC (Least-Cost) Branch and Bound. Explain how Travelling Salesman Problem (TSP) is solved using B&B.
- **Q7** Define Chromatic Number. Write a backtracking algorithm for the Graph Coloring problem.

Unit 4 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 5 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

