Unit-3

Dynamic Programming.

Que- What is Dynamic programming and characteristics of dynamic programming.

⇒

Dynamic programming :-

- Dynamic programming is a method used to solve complex problems by breaking them down into simpler subproblems.
- By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

- Approaches of Dynamic programming (DP) :-

1) Top-Down Approach (Memoization) :-

- In the top-down approach, also known as memoization, we start with the final solution and recursively break it down into smaller subproblems.
- To avoid redundant calculations, we store the results of solved subproblems in a memoization table.
- Let's breakdown Top down approach :-

1) Starts with the final solution and recursively breaks it down into smaller subproblems.
2) Stores the solutions to subproblems in a table to avoid redundant calculations.
3) Suitable when the number of subproblems is large and many of them are reused.

2) Bottom-up Approach (Tabulation):-
- In the bottom-up approach, also known as tabulation, we start with the smallest subproblems and gradually build up to the final solution.
- we store the results of solved subproblems in a table to avoid redundant calculations.
- Let's breakdown. Bottom-up approach:-
1) starts with the smallest subproblems and gradually builds up to the final solution.
2) fills a table with solutions to subproblems in a bottom-up manner.
3) suitable when the number of subproblems is small and the optimal solutions can be directly computed from the solutions to smaller subproblems.

• characteristics of Dynamic Programming:-

1) overlapping subproblems:-
- DP is used when a problem can be broken into subproblems that are solved repeatedly.

2) optimal substructure:-
- A problem exhibits optimal substructure if its optimal solution can be constructed from the optimal solutions of its subproblems.

3) Memoization (Top-down):-
- Recursively solve subproblems and store the results to avoid redundant calculations.

4) Tabulation (Bottom-up):-
- solve subproblems iteratively, storing results in a table to build up the final solutions.

Que- Explain Bellman Ford algorithm with example.
⇒

Bellman-Ford algorithm:-
- Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph.
- This algorithm can be used on both weighted and unweighted graphs.
- The Bellman-Ford algorithm is used to find the shortest path from starting vertex to all other vertices in a graph, even when some edges have negative weights.
- It also helps detect if there is a negative weight cycle.
- works with negative weights
- can detect negative weight cycles.

How Bellman-Ford works:-

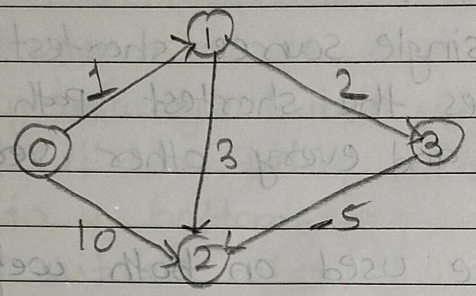
1) Initialize distances:-
- set the distance to the source vertex as 0 and all other vertices as infinity.

2) Relax all edges V-1 times:-
- for each edge, try to reduce the distance to the vertex it points to by checking if going

through another vertex offers a shorter path.

3] check for negative cycles :-
- After V-1 relaxations, if any edge can still be relaxed, it means there's a negative weight cycle.

• Example :-



Steps :-
1] Initialization :-
- Distance array (starting from vertex 0) :-
[0, ∞, ∞, ∞] (0 is source, others are ∞)

2] First iteration (relax all edges) :-
• Relax 0 → 1 :- distance to 1 becomes 1
[0, 1, ∞, ∞]
• Relax 1 → 2 :- distance to 2 becomes 4
[0, 1, 4, ∞]
• Relax 0 → 2 :- distance to 2 remains 4 (already smaller)

• Relax 1 → 3 :- distance to 3 becomes 3
[0, 1, 4, -3]

• Relax 3 → 2 :- distance to 2 becomes -2
[0, 1, -2, 3]

3] Second and third iterations:-
-No Further improvements, distances remain the same.

4] Check for negative cycle:-
-No edge can be relaxed Further, so no negative cycle is detected.

Final distances from vertex 0:-
[0, 1, -2, 3] (shortest paths to vertices 1, 2, 3).

• Advantages:-
1] Handles Negative weights
2] Detects Negative weight cycles
3] simple and Intuitive
4] works For Both Directed and undirected graphs.

• Disadvantages:-
1] slower compared to Dijkstra
2] inefficient for Dense Graphs
3] Redundant Relaxations.

• Applications:-
1] Routing in Networks
2] solving Problems with Negative weights
3] Railway or Airline Networks
4] Graph Problems with constraints.

Que- ~~Explain Reliability design with example~~

⇒

Que- Explain Traveling sales person Problem with example.

⇒

Traveling Sales Person Problem :-

- The Traveling Salesperson Problem (TSP) is a well-known optimization problem where a salesperson must visit a given set of cities, each exactly once and return to the starting city, all while minimizing the total travel distance.

• Dynamic programming Approach (Held-karp Algorithm)

- In Dynamic programming, we break the Problem down into smaller subproblems, solve each subproblem, and store the solutions to avoid redundant calculations.
- For TSP, we use the idea of recursively finding the minimum cost to visit subsets of cities and ending at a particular city.

• key concept :- Bitmasking + DP

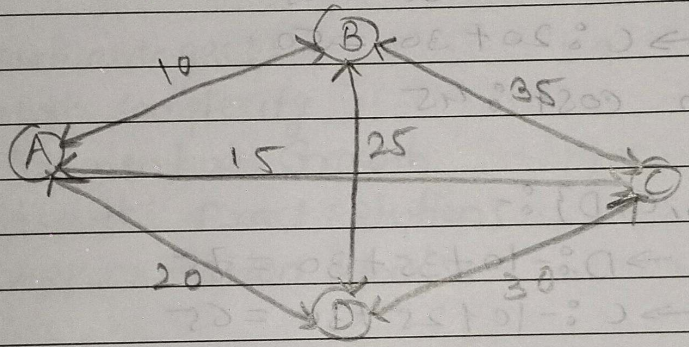
- In the DP approach to TSP, we track the cities that have been visited so far using a bitmask.
- Each bit in the bitmask represents whether a particular city has been visited.
- This allows us to efficiently represents and manage subsets of cities, which is essential For the DP solution.

• Terminology :-

1) n :- The total number of cities.
2) dist(i, j) :- The distance between city i and city j

3) dp[mask][i] :- The minimum cost to visit all cities represented in the bitmask mask and end at city i.
- The bitmask is a binary number where each bit represents whether a city is visited (1 if visited, 0 otherwise).

• Example :-
Find the shortest route starting from city A, visiting all cities (B, C, D) and returning to city A.



Steps :-
1] start at city A : dp[1][0] = 0 (cost is 0)

2] Calculate minimum Distance for Each subset
a) subset {B} :-
- cost to visit B from A :- 10

b] Subset {C} :-
- cost to visit C from A :- 15

c) Subset {D}
- cost to visit D from A :- 20

d) subset {B, C} :-
- A → B → C : 10 + 35 = 45
- A → C → B : 15 + 35 = 50
• minimum cost :- 45

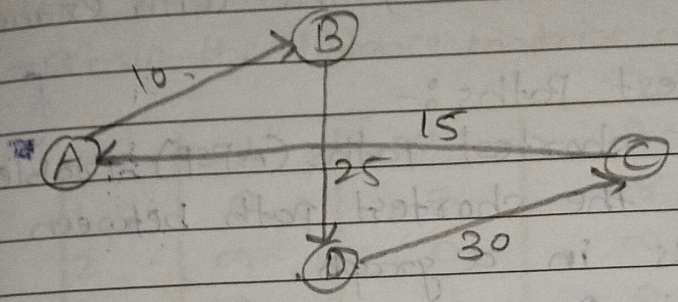
e) subset {B, D}
- A → B → D : 10 + 25 = 35
- A → D → B : 20 + 25 = 45
• minimum cost : 35

f) subset {C, D}
- A → C → D : 15 + 30 = 45
- A → D → C : 20 + 30 = 50
• minimum cost : 45

g) subset {B, C, D} :-
- A → B → C → D :- 10 + 35 + 30 = 75
- A → B → D → C :- 10 + 25 + 30 = 65
- A → C → B → D :- 15 + 35 + 25 = 75
- A → C → D → B :- 15 + 30 + 25 = 70
- A → D → B → C :- 20 + 25 + 35 = 80
- A → D → C → B :- 20 + 30 + 35 = 85
• minimum cost : 65

3) Return to starting city
For the best route A → B → D → C
• Add cost to return to A :- C → A = 15
• Total cost :- 65 + 15 = 80

Solution :-
The shortest route is A → B → D → C → A with total cost of 80 units.



- Advantages :-
1) optimization Benchmark
2) Exact Solutions For small Instances
3) Inspires Heuristics

- Disadvantages :-
1) High complexity
2) Exponential Growth
3) Limited Exact solutions
4) Approximation Limits.

- Applications :-
1) Logistics and transportation
2) Manufacturing
3) circuit design
4) Travel planning.

Que- Explain All pair shortest paths with example.

⇒

All pair shortest Paths :-

- The all Pair shortest Paths (APSP) Problem involves finding the shortest path between every pair of vertices in a graph.
- using dynamic programming, the Floyd-Warshall algorithm efficiently solves this problem by iteratively updating the shortest paths.

• Floyd - Warshall Algorithm :-

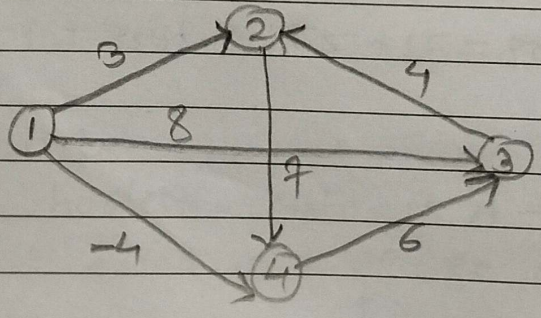
- The Floyd - warshall algorithm iteratively improves the shortest path distance by considering each vertex as an intermediate point.
- It checks whether going through this intermediate vertex provides a shorter path between every pair of vertices.

• Algorithm :-

$$d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$$

where, $d[i][j]$ is the distance from vertex i to vertex j and k is the intermediate vertex.

• Example :-



step 1 :- Initial Distance matrix
-set up the distance matrix with initial values

	1	2	3	4
1	0	3	8	-4
2	∞	0	∞	7
3	∞	4	0	∞
4	∞	∞	6	0

-where ∞ means no direct path between the nodes.

step 2 :- Apply the Floyd-warshall Algorithm
-We iteratively use each vertex as an intermediate node and update the matrix.

• Iteration 1 :- using vertex 1 (k=1) :
-No changes occurs in this iteration

• Iteration 2 :- using vertex 2 (k=2)

1] Pair (3,4) :-
d[3][4] = min (d[3][4], d[3][2] + d[2][4])
= min (∞, 4+7)
= 11

	1	2	3	4
1	0	3	8	-4
2	∞	0	∞	7
3	∞	4	0	11
4	∞	∞	6	0

• Iteration 3 :- using vertex 3 (k=3)
1] Pair (1,3) :-
d[1][3] = min (d[1][3], d[1][4] + d[4][3]) = min (8, -4+6)
= 2

2] Pair (2,3) :-

$d[2][3] = \min(d[2][3], d[2][4] + d[4][3]) =$

$= \min(\infty, 7+6)$

$= 13$

	1	2	3	4
1	0	3	2	-4
2	∞	0	13	7
3	∞	4	0	11
4	∞	∞	6	0

• Iteration 4 : using Vertex 4 (k=4)

1] Pair (4,2) :-

$d[4][2] = \min(d[4][2], d[4][3] + d[3][2])$

$= \min(\infty, 6+4)$

$= 10$

	1	2	3	4
1	0	3	2	-4
2	∞	0	13	7
3	∞	4	0	11
4	∞	10	6	0

Final shortest paths :-

- 1 → 3 : 2
- 2 → 3 : 13
- 4 → 2 : 10

• Advantages :-

1] Simple and easy to implement

2] Handles negative edge weights.

3) Suitable for dense graphs with $O(V^2)$ edges.
4) Works for both directed and undirected graphs.

- Disadvantages:-
1) Time complexity is $O(V^3)$
2) Not efficient for sparse graphs

- Applications:-
1) Network routing
2) Navigation systems
3) Urban planning
4) Game theory.

Que- Explain 0/1 knapsack Problem with example.
=>

0/1 knapsack Problem:-
-The 0/1 knapsack problem is a classic optimization problem where we are given:-
- A set of items, each with a weight and a value
- A knapsack with a maximum capacity.

-The goal is to maximize the total value of items in the knapsack without exceeding its weight capacity.
-The "0/1" aspect of the problem means you either take an item or leave it - no fractions of items are allowed.

• Example:-
A knapsack with a capacity of 5 units and 3 items.

Item	Value	weight
1	60	2
2	100	3
3	120	4

Goal :- Maximize value within the 5 unit capacity

Step 1 :- Create a Table.
-We create a table where rows represent items and columns represent knapsack capacities (0 to 5)

i/j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	60	60	60	60
2	0	0	60	100	100	160
3	0	0	60	100	120	160

Step 2 :- Fill the Table

1) Item 1 (value = 60, weight = 2) :- Fits when capacity ≥ 2, filling with value 60.

2) Item 2 (value = 100, weight = 3) :- Fits when capacity ≥ 3. At capacity 5, both item 1 and item 2 can fit, yielding value 160.

3) Item 3 (value = 120, weight = 4) :- Fits when capacity

≥ 4 but does not exceed value from item combinations at capacity 5.

step 3 :-
maximum value in the knapsack with a capacity of 5 is 160.

- Advantages :-
1) optimal solutions
2) Flexibility
3) Applicability

- Disadvantages :
1) Scalability
2) Complexity of Implementation
3) Time complexity

- Applications :-
1) Resource Allocation
2) Finance
3) Travel and logistics
4) Production

Que- Explain multistage Graphs with example.

⇒

Multistage Graphs :-

- A Multistage graph is a directed, weighted graph where vertices are divided into multiple stages, and each stage contains a set of vertices.
- The goal in many multistage graph Problem is to find the shortest or longest Path from the source node in the first stage to the sink node in the last stage.
- Dynamic programming (DP) is commonly used to solve these Problems efficiently.

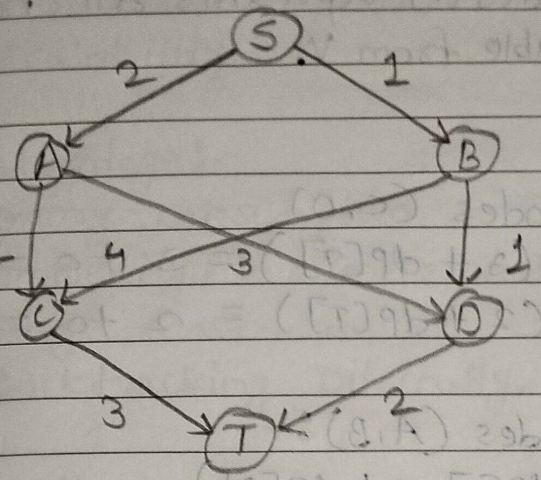
• key concept :-

1) Stages :- The graph is divided into stages and nodes in a stage can only connect to nodes in the next stage.
2) Source and sink :- The first stage has the source node and the last stage has the sink node.
3) Cost/weight :- Each edge between nodes has an associated cost or weight.

• Example :-

consider a multistage graph with four stages as follows :-

stage 1	stage 2	stage 3	stage 4
S	A, B	C, D	T
(source)			(sink)



Problem :- Find the shortest Path from S TO T

-To solve this Dynamic programming, we'll work backward from the sink node, computing the minimum cost to reach the sink node from each preceding node.

•steps :-

1) Define state :-

- Let dP[V] represent the minimum cost to reach the sink node T from node V.

2) Initialization :-

- The cost to reach the sink node from itself is zero : dP[T] = 0

3) Recurrence Relation :-

- For any node V in stages 1, 2 or 3, calculate the minimum cost using :-

$$dp[V] = \min_{u \in next\_nodes(V)} (cost(V,u) + dP[u])$$

- Here next_nodes(v) represents all nodes directly reachable from V.

4) Calculation :-

- For stage 3 nodes (C, D)
  - dp[C] = min (3 + dp[T]) = 3 + 0 = 3
  - dp[D] = min (2 + dp[T]) = 2 + 0 = 2

- For stage 2 nodes (A, B) :-
  - dp[A] = min (2 + dp[C], 3 + dp[D])
    = min (2+3, 3+2) = 5
  - dp[B] = min (4 + dp[C], 1 + dp[D])
    = min (4+3, 1+2)
    = 3

- For stage 1 node (S)
  - dp[S] = min (2 + dp[A], 1 + dp[B])
    = min (2+5, 1+3)
    = 4

- Final Solution
  - The shortest path cost from S to T is dp[S] = 4

Shortest Path :-

- The Path S → B → D → T gives the shortest path with cost of 4.

- Advantages :-
  1) optimal solution
  2) Efficiency.

3] structured Approach
4] scalability.

• Disadvantage :-
1] memory consumption
2] complexity in Implementation
3] stage constraint
4] Backtracking Difficulty.

• Application :-
1] Network Routing.
2] Resource Allocation
3] Game Theory
4] Production planning.

que- Explain optimal Binary search Tree (OBST) with example.

⇒

optimal Binary search Tree :-
- An optimal Binary search Tree (OBST) is a binary search tree constructed to minimize the total search cost for given access Probabilities of elements.
- The goal is to structure the tree in a way that frequently accessed nodes are closer to the root, minimizing the overall search time.

• Problem definition :-
- Given a set of keys and their associated Probabilities, the objective is to build a binary search tree with the minimum expected search cost.

• key points :-
1) keys (k) :- A sorted sequence of keys
2) Probability of searching (P) :- Probability of accessing each key.
3) Dummy keys (Q) :- Probabilities of searches for keys that don't exist (i.e. external nodes).

• Example :-
Given keys and probabilities as :-
• keys :- k_1, k_2, k_3
• probabilities :- P = [0.2, 0.5, 0.3]
• Dummy Probabilities :- Q = [0.1, 0.05, 0.05, 0.1]

step 1 :- Initialize costs for empty trees

c[i][i-1] = q[i-1]
c[1][0] = 0.1, c[2][1] = 0.05, c[3][2] = 0.05, c[4][3] = 0.1

step 2 :- compute w[i][i]
- For single elements :-
• w[1][1] = p[1] + q[0] + q[1] = 0.2 + 0.1 + 0.05 = 0.35
• w[2][2] = p[2] + q[1] + q[2] = 0.5 + 0.05 + 0.05 = 0.6
• w[3][3] = p[3] + q[2] + q[3] = 0.3 + 0.05 + 0.1 = 0.45

- For ranges :-
• w[1][2] = w[1][1] + p[2] + q[2] = 0.35 + 0.5 + 0.05 = 0.9
• w[2][3] = w[2][2] + p[3] + q[3] = 0.6 + 0.3 + 0.1 = 1.0
• w[1][3] = w[1][2] + p[3] + q[3] = 0.9 + 0.3 + 0.1 = 1.3

Step 3 :- compute the cost c[i][j] using recurrence relation

1) For C[1][1] :-

C[1][1] = min(C[1][0] + C[2][1] + W[1][1])

= 0.1 + 0.05 + 0.35

= 0.5

2) For C[2][2]

C[2][2] = min(C[2][1] + C[3][2] + W[2][2])

= 0.05 + 0.05 + 0.6

= 0.7

3) For C[1][2] :-

C[1][2] = min(C[1][0] + C[2][2] + W[1][2], C[1][1] + C[3][2] + W[1][2])

C[1][2] = min(0.1 + 0.7 + 0.9, 0.5 + 0.05 + 0.9)

= 1.5

4) For C[1][3] :-

C[1][3] = min(C[1][0] + C[2][3] + W[1][3], C[1][1] + C[3][3] + W[1][3],
C[1][2] + C[4][3] + W[1][3])

= min(0.1 + 1.0 + 1.3, 0.5 + 0.45 + 1.3, 1.5 + 0.1 + 1.3)

= 2.4

optimal cost :- The optimal cost of binary search tree is C[1][3] = 2.4

• Advantages :-

1) minimized search cost
2) Efficient for static data
3) Predictable performance

• Disadvantages :-

1) High Preprocessing Time
2) Memory Intensive
3) Not self-Balancing.

• Applications :-

1) Database Indexing
2) Search engines
3) Decision Tree
4) Financial systems.