Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **IT-403**

Semester: **4th**

Unit-3: Dynamic Programming

Introduction:

Dynamic programming (DP) is a general algorithm design technique for solving problems with overlapping sub-problems. This technique was invented by American mathematician "Richard Bellman" in 1950s.

Key Idea

The key idea is to save answers of overlapping smaller sub-problems to avoid re-computation.

Dynamic Programming Properties

- An instance is solved using the solutions for smaller instances.
- The solutions for a smaller instance might be needed multiple times, so store their results in a table.
- Thus each smaller instance is solved only once.
- Additional space is used to save time.

Rules of Dynamic Programming

1. OPTIMAL SUB-STRUCTURE: An optimal solution to a problem contains optimal solutions to sub-problems
2. OVERLAPPING SUB-PROBLEMS: A recursive solution contains a "small" number of distinct sub-problems repeated many times
3. BOTTOM UP FASHION: Computes the solution in a bottom-up fashion in the final step

Example Problems that can be solved using Dynamic Programming method

1. Computing binomial co-efficient
2. Compute the longest common subsequence
3. Warshall's algorithm for transitive closure
4. Floyd's algorithm for all-pairs shortest paths
5. Some instances of difficult discrete optimization problems like
6. knapsack problem
7. traveling salesperson problem

0/1 knapsack:

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same. 0-1 Knapsack cannot be

solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.
The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (***p_i/w_i***), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max¡mum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is W = 30 and the items are as shown in the following table.

Item	**A**	**B**	**C**
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a max¡mal weight of ***W*** into his knapsack. There are ***n*** items and weight of ith item is ***w_i*** and the profit of selecting this item is ***p_i***. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then ***$S' = S - \{i\}$*** is an optimal solution for ***W - w_i*** dollars and the value to the solution ***S*** is ***V_i*** plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if $w_i \le w$ then
      if $v_i$ + c[i-1, w-$w_i$] then
        c[i, w] = $v_i$ + c[i-1, w-$w_i$]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes θ(n, w) times as table c has (n + 1).(w + 1) entries, where each entry requires θ(1) time to compute.

Multi Stage Graph

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets ***S = {s_1,s_2,…,s_k}*** such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

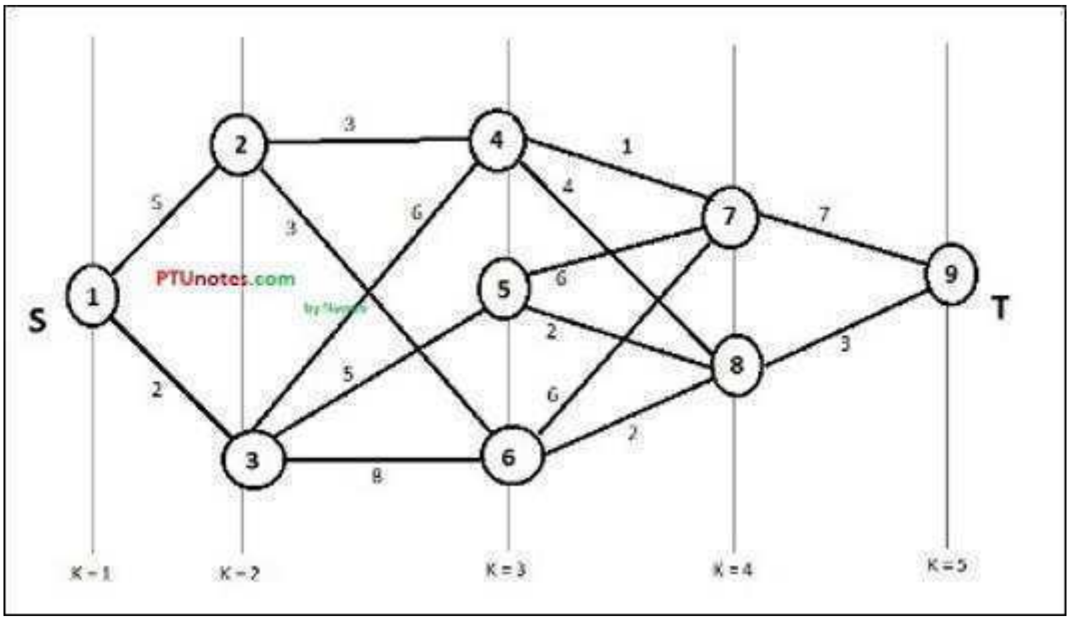
The vertex ***s* ∈ *s_1*** is called the **source** and the vertex ***t* ∈ *s_k*** is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = min {c(3, 4) + Cost(4, 9) + Cost(8, 9),c(3, 5) + Cost(8, 9)} = 7

Step-3: Cost (K-4, j)

Cost (1, 1) = min {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9),

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1→ 2→ 6→ 8→ 9**.

Reliability Design

In this section, we present the dynamic programming approach to solve a problem with multiplicative constraints. Let us consider the example of a computer return in which a set of nodes are connected with each other. Let r i be the reliability of a node i.e. the probability at which the node forwards the packets correctly in r i . Then the reliability of the path connecting from one node s to k∏ r where k is the number of intermediate node. Similarly, we can also consider a another node d isi i = 1 system with n devices connected in series, where the reliability of device i is r i .

The reliability of the k system is ∏ r . For example if there are 5 devices connected in series and the reliability of each device i i = 1 is 0.99 then the reliability of the system is 0.99 × 0.99 × 0.99 × 0.99 × 0.99=0.951. Hence, it is desirable to connect multiple copies of the same devices in parallel through the use of switching circuits. The switching circuits determine the devices in any group functions properly. Then they make use of one such device at each stage.Let m i be the number of copies of device D i in stage i. Then the probability that all m i have malfunction i.e. (1-r i) mi .Hence, the reliability of stage i becomes 1-(1-r i) mi .

Thus, if r i =0.99 and m i =2, the reliability of stage i is 0.9999. However, in practical situations it becomes less because the switching circuits are not fully reliable. Let us assume that the reliability of stage i in ϕ i (m i), i≤n. Thus the reliability k of the system is ∏ ϕ (m) .

The reliability design problem is to use multiple copies of the devices at each stage to increase reliability. However, this is to be done under a cost constraint. Let c i be the cost of each unit of device D i and let c be the cost constraint. Then the objective is to maximize the reliability under the condition that the total cost of the system m i c i will be less than c.

Mathematically, we can write maximize
∏ ϕ (m) i1 ≤ i ≤ n is subject to ∑ c m ≤ c 1 ≤ i ≤ n m i ≥ 1 and 1 ≤ i ≤n

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm works based on a property of *intermediate* vertices of a shortest path. An *intermediate* vertex for a path $p = <v_1, v_2, ..., v_j>$ is any vertex other than v_1 or v_j.

If the vertices of a graph *G* are indexed by {1, 2, ..., *n*}, then consider a subset of vertices {1, 2, ..., *k*}. Assume *p* is a minimum weight path from vertex *i* to vertex *j* whose intermediate vertices are drawn from the subset {1, 2, ..., *k*}. If we consider vertex *k* on the path then either:

k is **not** an intermediate vertex of *p* (i.e. is not used in the minimum weight path)

⇒all intermediate vertices are in {1, 2, ..., *k*-1}

k is an intermediate vertex of *p* (i.e. is used in the minimum weight path)

⇒we can divide *p* at *k* giving two subpaths p_1 and p_2 giving $v_i \leadsto k \leadsto v_j$

⇒ by Lemma 24.1 (subpaths of shortest paths are also shortest paths) subpaths p_1 and p_2 are shortest paths with intermediate vertices in {1, 2, ..., k-1}

Thus if we define a quantity $d^{(k)}_{ij}$ as the minimum weight of the path from vertex i to vertex j with intermediate vertices drawn from the set {1, 2, ..., k} the above properties give the following recursive solution

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Thus we can represent the optimal values (when $k = n$) in a matrix as

$$D^{(n)} = \left[d_{ij}^{(n)}\right] = \left[\delta(i, j)\right]$$

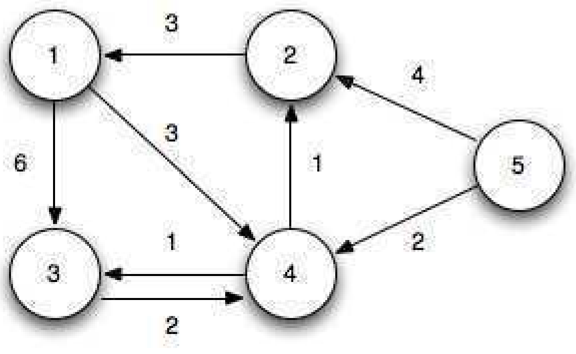
Algorithm FloydWarshall(D, P)

```
1.for k in 1 to n do
      2.for i in 1 to n do
            3.for j in 1 to n do
                  4.if D[i][j] > D[i][k] + D[k][j] then
                        5.D[i][j] = D[i][k] + D[k][j]
                        6.P[i][j] = P[k][j]
   7.return P
```

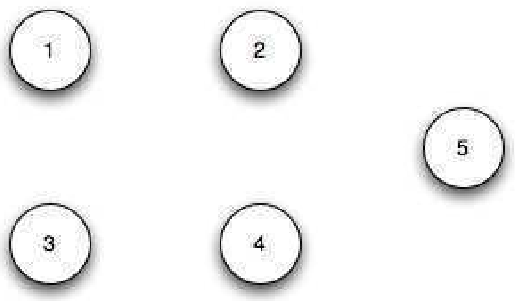
Basically the algorithm works by repeatedly exploring paths between every pair using each vertex as an intermediate vertex. Since Floyd-Warshall is simply three (tight) nested loops, the run time is clearly O(V^3).

Example

Using the directed graph



Initialization: (k = 0)



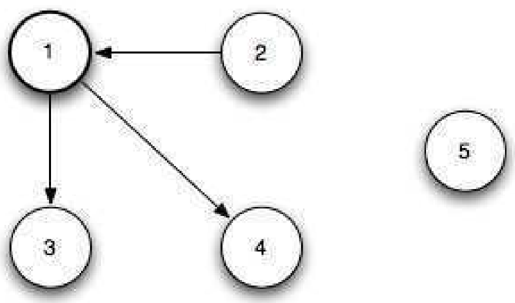
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 1: (*k* = 1) Shorter paths from 2 ⇝3 and 2 ⇝4 are found through vertex 1



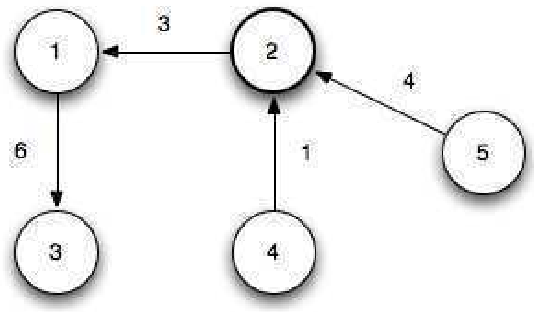
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 2: (*k* = 2) Shorter paths from 4 ⇝1, 5 ⇝1, and 5 ⇝3 are found through vertex 2



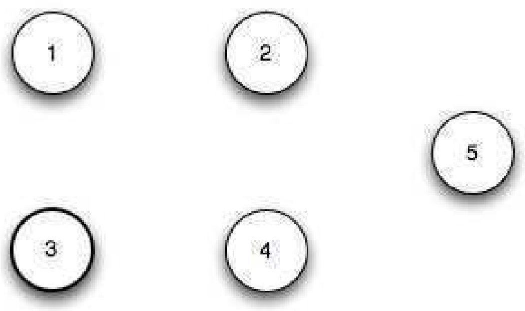
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**2**	5	/

Iteration 3: (*k* = 3) No shorter paths are found through vertex 3



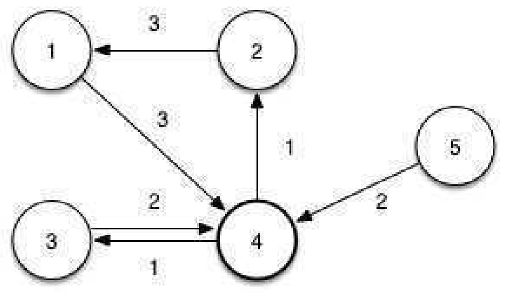
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	2	5	/

Iteration 4: (*k* = 4) Shorter paths from 1 ⇝2, 1 ⇝3, 2 ⇝3, 3 ⇝1, 3 ⇝2, 5 ⇝1, 5 ⇝2, 5 ⇝3, and 5 ⇝4 are found through vertex 4



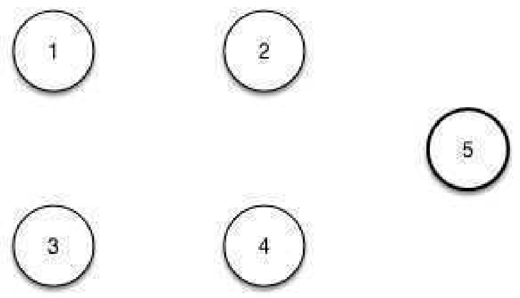
D

	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Π

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**2**	**4**	/	3	/
4	2	4	4	/	/
5	2	**4**	**4**	5	/

Iteration 5: (*k* = 5) No shorter paths are found through vertex 5



D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

The final shortest paths for all pairs is given by

D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/