18MCA34C

DESIGN AND ANALYSIS OF ALGORITHM

UNIT IV

Dynamic programming

FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

Dynamic programming general method

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Multistage graph

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k*> 1**) number of disjoint subsets $S = \{s_1, s_2, \ldots, s_k\}$ 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 $|s_1| = |s_k| = 1$.

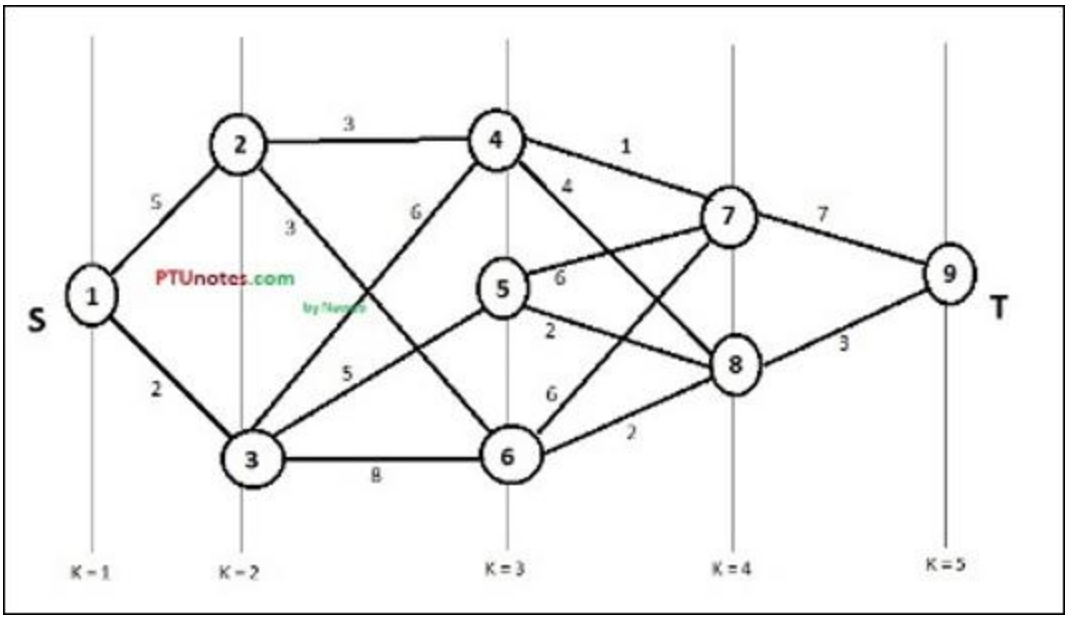
The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**.

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) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

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

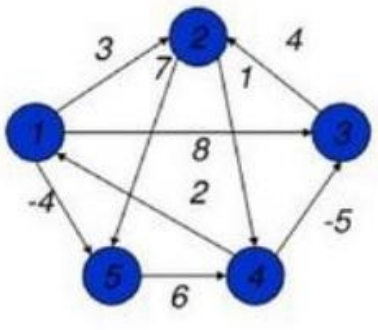
Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

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

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

All pair shortest path

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

Output − Matrix of all pair shortest path.

0 3 4 5 6 7 7

3 0 2 1 3 4 4

4 2 0 1 3 2 3

5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

Algorithm

floydWarshal(cost)

Input − The cost matrix of given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
  for k := 0 to n, do
    fori := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
```

done

done

display the current cost matrix

End

Optimal binary search tree

the Optimal Binary Search Tree Algorithm is presented. First, we build a BST from a set of provided **n** number of distinct keys < $k_1, k_2, k_3, ...k_n$>. Here we assume, the probability of accessing a key K_i is p_i. Some dummy keys ($d_0, d_1, d_2, ...d_n$) are added as some searches may be performed for the values which are not present in the Key set ***K***. We assume, for each dummy key d_i probability of access is q_i.

Optimal-Binary-Search-Tree(p, q, n)

e[1…n + 1, 0…n],

w[1…n + 1, 0…n],

root[1…n + 1, 0…n]

fori = 1 to n + 1 do

e[i, i - 1] := q_i - 1

w[i, i - 1] := q_i - 1

for l = 1 to n do

fori = 1 to n – l + 1 do

j = i + l – 1 e[i, j] := ∞

w[i, i] := w[i, i -1] + p_j + q_j

for r = i to j do

t := e[i, r - 1] + e[r + 1, j] + w[i, j]

if t < e[i, j]

e[i, j] := t

root[i, j] := r

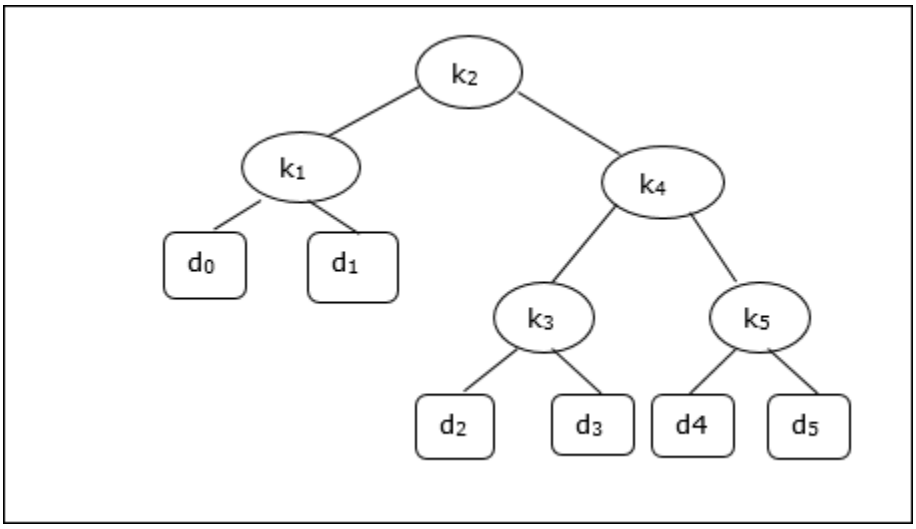
return e and root

Analysis

The algorithm requires **O (n^3)** time, since three nested **for** loops are used. Each of these loops takes on at most **n** values.

Example

Considering the following tree, the cost is 2.80, though this is not an optimal result.



Node Depth Probability Contribution

k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20

d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

To get an optimal solution, using the algorithm discussed in this chapter, the following tables are generated.

In the following tables, column index is **i** and row index is **j**.

e	**1**	**2**	**3**	**4**	**5**	**6**
5	2.75	2.00	1.30	0.90	0.50	0.10
4	1.75	1.20	0.60	0.30	0.05	
3	1.25	0.70	0.25	0.05		
2	0.90	0.40	0.05			
1	0.45	0.10				
0	0.05					

w	**1**	**2**	**3**	**4**	**5**	**6**
5	1.00	0.80	0.60	0.50	0.35	0.10
4	0.70	0.50	0.30	0.20	0.05	
3	0.55	0.35	0.15	0.05		
2	0.45	0.25	0.05			
1	0.30	0.10				

0 0.05

root 1 2 3 4 5

5 2 4 5 5 5

4 2 2 4 4

3 2 2 3

2 1 2

1 1

From these tables, the optimal tree can be formed.

0/1 knapsack problem

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 ($\boldsymbol{p_i/w_i}$), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$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 $W = 60$ 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$_i$mal weight of W into his knapsack. There are n items and weight of i^{th} 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 max$_i$mum weight **w**.

The algorithm takes the following inputs

- The max$_i$mum weight **W**
- The number of items **n**
- The two sequences $\mathbf{v} = \langle v_1, v_2, \ldots, v_n \rangle$ and $\mathbf{w} = \langle w_1, w_2, \ldots, w_n \rangle$

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

for w = 0 to W do
c[0, w] = 0
fori = 1 to n do
c[i, 0] = 0
for w = 1 to W do
if$w_i \leq$ 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 $\theta(n, w)$ times as table *c* has $(n + 1).(w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Travelling salesman problem

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!**number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S Є {1, 2, 3, ... , n}*** that includes ***1***, and ***j Є S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$C(S,j)=minC(S-\{j\},i)+d(i,j)wherei\in Sandi\neq jc(S,j)=minC(s-\{j\},i)+d(i,j)wherei\in Sandi\neq j$

Algorithm: Traveling-Salesman-Problem

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

Analysis

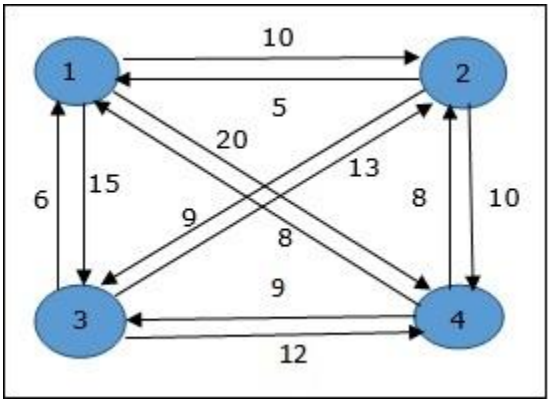
There are at the most $2n.n$

sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2n.n2)$

.

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

$Cost(2,\Phi,1)=d(2,1)=5Cost(2,\Phi,1)=d(2,1)=5$

$Cost(3,\Phi,1)=d(3,1)=6Cost(3,\Phi,1)=d(3,1)=6$

$Cost(4,\Phi,1)=d(4,1)=8Cost(4,\Phi,1)=d(4,1)=8$

S = 1

$Cost(i,s)=min\{Cost(j,s–(j))+d[i,j]\}Cost(i,s)=min\{Cost(j,s)–(j))+d[i,j]\}$

$Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15cost(2,\{3\},1)=d[2,3]+cost(3,\Phi,1)=9+6=15$

$Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18cost(2,\{4\},1)=d[2,4]+cost(4,\Phi,1)=10+8=18$

$Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18cost(3,\{2\},1)=d[3,2]+cost(2,\Phi,1)=13+5=18$

$Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20 cost(3,\{4\},1)=d[3,4]+cost(4,\Phi,1)=12+8=20$

$Cost(4,\{3\},1)=d[4,3]+Cost(3,\Phi,1)=9+6=15 cost(4,\{3\},1)=d[4,3]+cost(3,\Phi,1)=9+6=15$

$Cost(4,\{2\},1)=d[4,2]+Cost(2,\Phi,1)=8+5=13 cost(4,\{2\},1)=d[4,2]+cost(2,\Phi,1)=8+5=13$

S = 2

$Cost(2,\{3,4\},1)=\{d[2,3]+Cost(3,\{4\},1)=9+20=29 d[2,4]+Cost(4,\{3\},1)=10+15=25=25 Cost(2,\{3,4\},1)\{d[2,3]+cost(3,\{4\},1)=9+20=29 d[2,4]+Cost(4,\{3\},1)=10+15=25=25$

$Cost(3,\{2,4\},1)=\{d[3,2]+Cost(2,\{4\},1)=13+18=31 d[3,4]+Cost(4,\{2\},1)=12+13=25=25 Cost(3,\{2,4\},1)\{d[3,2]+cost(2,\{4\},1)=13+18=31 d[3,4]+Cost(4,\{2\},1)=12+13=25=25$

$Cost(4,\{2,3\},1)=\{d[4,2]+Cost(2,\{3\},1)=8+15=23 d[4,3]+Cost(3,\{2\},1)=9+18=27=23 Cost(4,\{2,3\},1)\{d[4,2]+cost(2,\{3\},1)=8+15=23 d[4,3]+Cost(3,\{2\},1)=9+18=27=23$

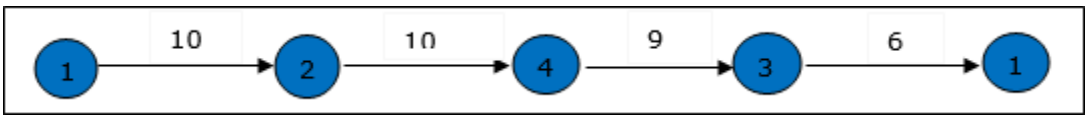
S = 3

$Cost(1,\{2,3,4\},1)=\{d[1,2]+Cost(2,\{3,4\},1)=10+25=35 d[1,3]+Cost(3,\{2,4\},1)=15+25=40 d[1,4]+Cost(4,\{2,3\},1)=20+23=43=35 cost(1,\{2,3,4\}),1) d[1,2]+cost(2,\{3,4\},1)=10+25=35 d[1,3]+cost(3,\{2,4\},1)=15+25=40 d[1,4]+cost(4,\{2,3\},1)=20+23=43=35$

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.