Date
Page

Q. Compare Dijkstra's & Bellman Ford's Algorithm.

Ans.

Dijkstra's Algorithm	Bellman Ford's Algorithm
(1.) It may or maynot work when there is negative weight edge. But will not work when there is negative weight cycle.	It works when there is negative weight edge, it also detects the negative weight cycle.
(2.) The result contain vertices containing whole info. about the network.	The result only contains vertices which contains info. about other vertices they are connected to.
(3.) It can't be implemented easily in a distributed way.	It can easily be implemented in a distributed way.
(4.) It is less time consuming	It is more time consuming than Dijkstra's.
(5.) Time complexity is O(E logV)	Time complexity is O(VE).
(6.) Greedy approach is taken to implement the algorithm.	Dynamic Programming approach is taken to implement the algorithm.

Q. Compare Prim's & Kruskal's Algorithm.

Prim's Algorithm	Kruskal's Algorithm
(1) It starts to build minimum spanning tree from any vertex in graph.	(1) It starts to build the minimum spanning tree from vertex carrying minimum weight in graph.
(2.) It traverses one node more than one time to get the minimum distance.	(2) It traverses one node only once.
(3.) Its time complexity is $O(E \log V)$.	(3) Its time complexity is $O(E \log E)$.
(4) Gives connected component as well as it works only on connected graph.	(4) Can generate forest at any instant as well as it can work on disconnected components.
(5) Runs faster in dense graphs.	(5) Runs faster in sparse graphs.

x———x

Q. Discuss & give example of greedy method to solve Knapsack problem.

An. → In this, we can break items for maximizing total value of knapsack. Fraction is allowed.

→ A brute force solution would be to try all possible subset with all different fraction that will be too much time taking.

=> An efficient solution is to use greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each items & sort the items on basis of this ratio.

=> Then take the item ~~on basis~~ of with the highest ratio & add them until we can't find add the next item as a whole & at the end add next item as much as we can.

=> This will always give an optimal solution.

=> The Knapsack problem can be stated as:-

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \qquad \text{— (1)}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \qquad \text{— (2)}$$

$$\text{and } 0 \le x_i \le 1 \;, \; 1 \le i \le n \qquad \text{— (3)}$$

=> A feasible soln. is any set satisfying (2) & (3).

=> An optimal solution is a feasible solution for which (1) is maximized.

For example:- Item as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Knapsack capacity, m = 50.

Output :- maximum possible value = 240, by taking full items of 10kg, 20kg and 2/3rd of last item of 30kg.

* Jdo greedy algo. n/ ayi tn simple knapsack

* Jdo dynamic tn 0/1 knapsack

Page

Q9. Demonstrate how dynamic programming can be used to solve knapsack problem. [0/1 knapsack]

Ans 9. The 0/1 knapsack problem can be stated as:-

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$ [optimal soln & objective fnc.]

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$ [constraint]

x_i is either 0 or 1 [feasible soln.]

=> Using dynamic programming [such that the principle of optimality holds]:-

$$f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\}$$

=> Generalizing the eqn:-

$$f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}$$

To represent $f_i(y)$ an ordered set S^i is used such that:-

$$S^i = \{f(y_j), y_j\}$$

Each member of S^i is a pair (P, W)

Compute S^i, then S^i_1, S^{i+1} ..... for all elements.

For example: $n = 3$ $(P_1, P_2, P_3) = (1, 2, 5)$
$m = 6$ $(w_1, w_2, w_3) = (2, 3, 4)$

Solve using dynamic programming.

$$\begin{cases} S^0 = \{(0,0)\} \\ S^0_1 = \{(1,2)\} \end{cases}$$

$$\begin{cases} S^1 = \{(0,0),(1,2)\} \\ S^1_1 = \{(2,3),(3,5)\} \end{cases}$$

$$\begin{cases} S^2 = \{(0,0),(1,2),(2,3),(3,5)\} \\ S^2_1 = \{(5,4),(6,6),(7,7),(8,9)\} \end{cases}$$

$S^3 = \{(0,0),(1,2),(2,3),(3,5),(5,4),$
$(6,6),(7,7),(8,9)\}$

$(3,5)$ is rejected as $w_j > w_k \mid P_j \le P_k$
$(7,7)$ $(8,9)$ is rejected as
these are purged as $w > m$.

So,
$S^3 = \{(0,0),(1,2),(2,3),(5,4),(6,6)\}$

Trace Back : Max $\to (6,6)$

$P_1 = 1$	$P_2 = 2$	$P_3 = 5$
$w_1 = 2$	$w_2 = 3$	$w_3 = 4$
$i = 0$	$i = 1$	$i = 2$

first appeared for S^2_1
$\therefore$ x_1 x_2 x_3
$\downarrow$
$(5,4)$

So $(6,6)$
$-(5,4)$
$(1,2)$

Date
Page

=> (1,2) is first appeared at S^0_1

$$\therefore \begin{array}{ccc} x_1 & x_2 & x_3 \\ | & & | \\ (1,2) & & (5,4) \end{array}$$

Now (1,2)
$-(1,2)$
$\overline{\quad 0 \quad}$ — (Rest are not included)

Hence,

$$\begin{array}{ccc} x_1 & x_2 & x_3 \\ 1 & 0 & 1 \\ (1,2) & & (5,4) \end{array}$$

Solution : $\begin{pmatrix} x_1, & x_2, & x_3 \\ 1, & 0, & 1 \end{pmatrix}$ for 0/1 knapsack problem.

Que. Elaborate how dynamic prog. is used by Floyd-Warshal Algorithm. evaluate the efficiency of Floyd-Warshal Algorithm.

Ans. Floyd-Warshall Algorithm is an algorithm for finding shortest path between all pairs of vertices in a weighted graph. This algorithm follows dynamic programming approach to find shortest paths.

-> It involves sequence of decisions :-

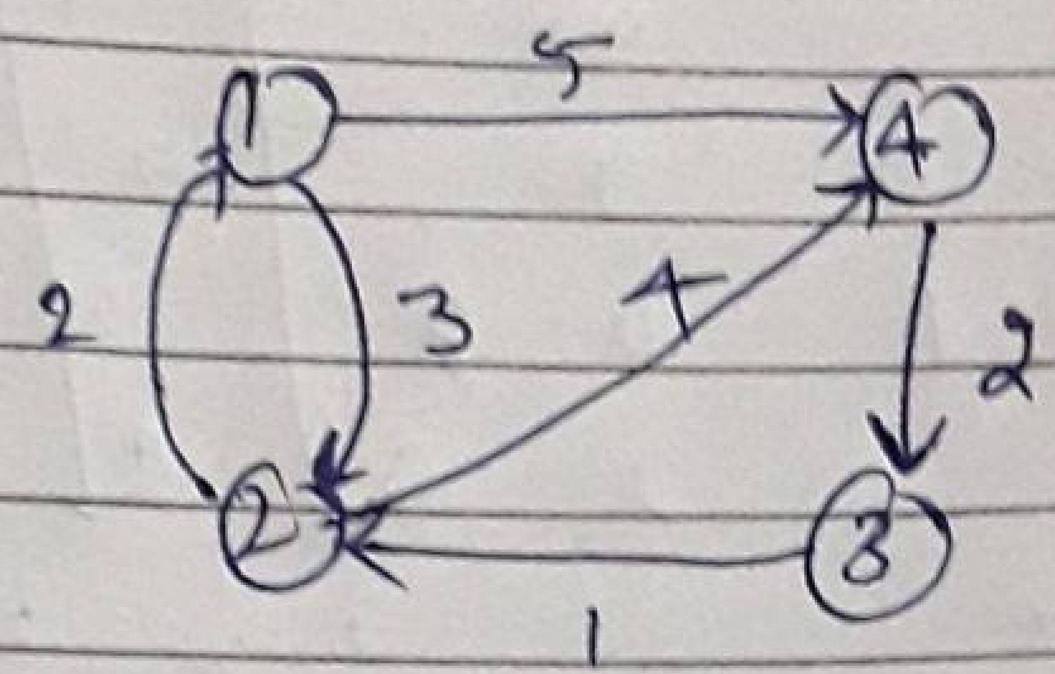
$$A^k(i,j) = \min\{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\}$$

(for $k \geq 1$)

=> Graph should have no cycles with negative length for Floyd-Warshal Algorithm.

Scanned by TapScanner

Lets solve Floyd-Warshall Algorithm using dynamic programming with an example:-



→ follow the steps below:-

(1.) Create a matrix A^0 of dimension $n*n$. Each cell $A[i][j]$ is filled with distance from i^{th} vertex to j^{th} vertex.

$$A^0 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{array}\right] \end{array}$$

(2) Now create a matrix $[A']$ using A^0. The elements in first column & first row are left as they are. Here '1' is intermedia[te] vertex.

$$A' = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{array}\right] \end{array} \longrightarrow \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{array}\right] \end{array}$$

(3) Similarly A^2 is created using A'.

$$A^2 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{array}\right] \end{array} \longrightarrow \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{array}\right] \end{array}$$

(4) Similarly A^3, A^4 is also created.

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 4 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(5.) A^4 gives shortest path between each pair of vertices.

⇒ Floyd-Warshall Algorithm

n = no. of vertices
A = matrix of dimension $n*n$
for $k=1$ to n
    for $i=1$ to n
        for $j=1$ to n
            $A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$
return A.

* ⇒ It has time complexity as $O(n^3)$.