**ASSIGNMENT 2**

**Title:** Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity

**Software Requirement:** Ubantu, C++ Compiler

**Theory:**

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The sub-problems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

**1. Dynamic Programming:**

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler sub-problems, solving each sub-problem just once, and then storing their solutions to avoid repetitive computations.

The following are the steps that the dynamic programming follows:

* It breaks down the complex problem into simpler subproblems.
* It finds the optimal solution to these sub-problems.
* It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
* It reuses them so that same sub-problem is calculated more than once.
* Finally, calculate the result of the complex problem.

**2. Working of Dynamic Programming**

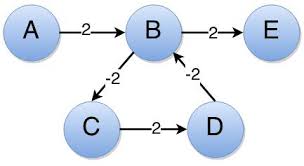
Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

* **Optimal Sub-structure**: A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming.
* **Principle of Optimality:** Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them. This technique of storing the value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

**3. Bellman Ford Algorithm**

Bellman Ford algorithm used to find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights. Solves single shortest path problem in which edge weight may be negative but no negative cycle exists.

This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination. It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges. This algorithm detects the negative cycle in a graph and reports their existence.



**Problem Statement:** Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

* Based on the "***Principle of Relaxation***" in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.
* Given a weighted directed graph G = (V, E) with source s and weight function w: E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source.
* If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative - weight cycles that are reachable from the source.

**Steps to solve Bellman Ford**

* **Input:** Graph and a source vertex src
* **Output:** Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1. This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0.

* Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

1. This step calculates shortest distances.

Do following |V|-1 times where |V| is the number of vertices in given graph.

for each edge u-v

If dist[v] > dist[u] + weight of edge uv, then update dist[v]

dist[v] = dist[u] + weight of edge uv

1. This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

* If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

**Algorithm: Bellman Ford**

BELLMAN -FORD (G, w, s)

INITIALIZE - SINGLE - SOURCE (G, s)

for i ← 1 to |V[G]| - 1

do for each edge (u, v) ∈ E [G]

do RELAX (u, v, w)

for each edge (u, v) ∈ E [G]

do if d [v] > d [u] + w (u, v)

then return FALSE.

return TRUE.

**Example:**

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

**4. Complexity Analysis**

* Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.
* The one for loop is used for initialization, which runs in O(V) times. The another for loop runs |V - 1| passes over the edges, which takes O(E) times.
* Hence, Bellman-Ford algorithm runs in O(V, E) time.

**Conclusion:**

Bellman-Ford Algorithm using Dynamic Programming is implemented successfully.