Experiment No. 8
Single Source Shortest Path using Dynamic Programming (Bellman-Ford Algorithm)
Date of Performance:
Date of Submission:

Experiment No: 8

Title: Single Source Shortest Path: Bellman Ford

Aim: To study and implement Single Source Shortest Path using Dynamic Programming: Bellman Ford

Objective: To introduce Bellman Ford method

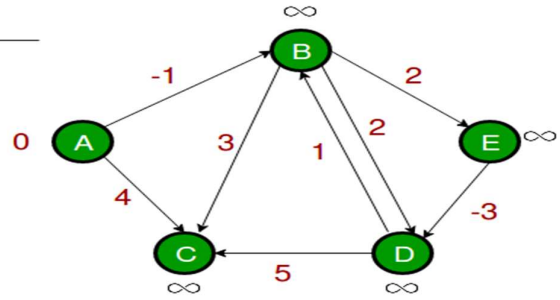
Theory:

Given a graph and a source vertex source in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. 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.

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.

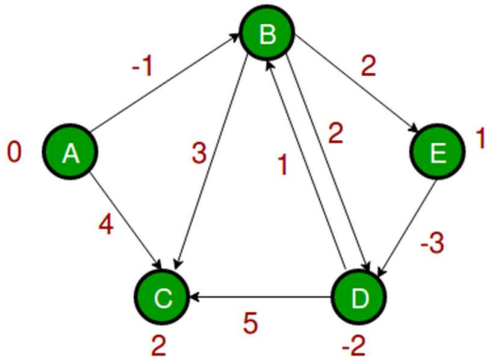
A	B	C	D	E
0	∞	∞	∞	∞



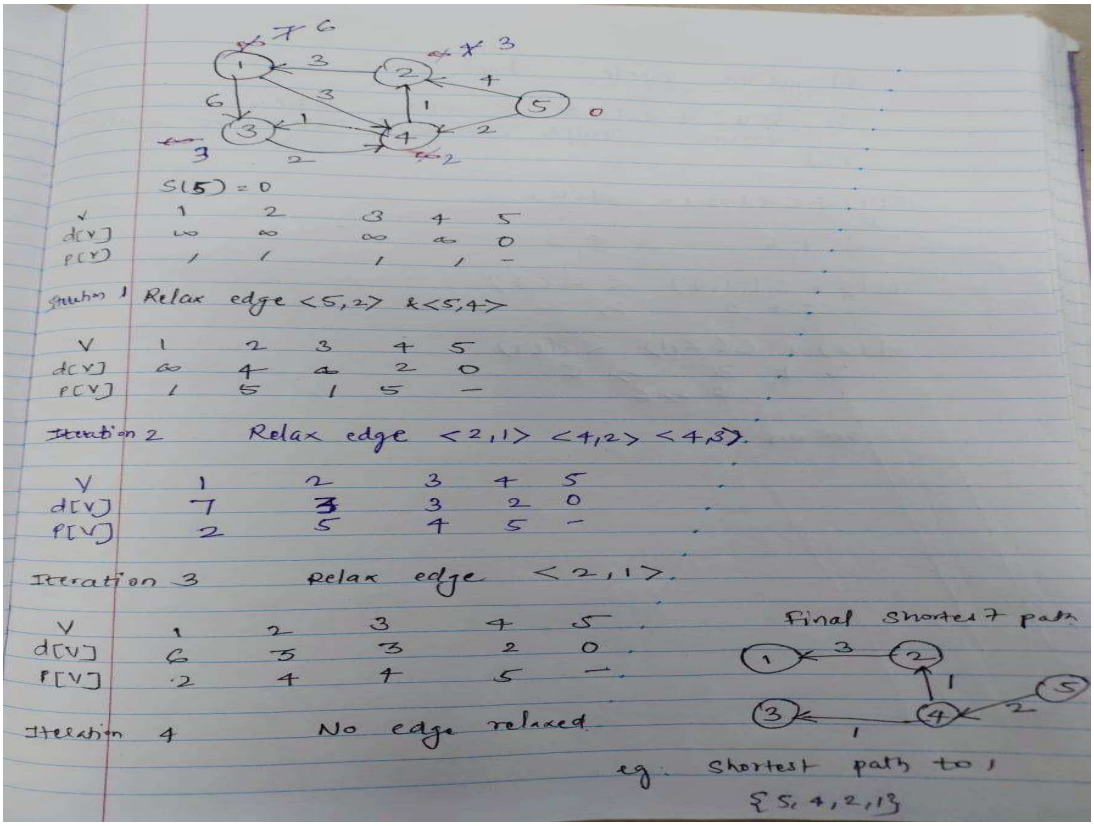
Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D,

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.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



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.



Algorithm:

```
function Bellman_Ford(list vertices, list edges, vertex source, distance[], parent[])

// Step 1 – initialize the graph. In the beginning, all vertices weight of
// INFINITY and a null parent, except for the source, where the weight is 0

for each vertex v in vertices
   distance[v] = INFINITY
   parent[v] = NULL

distance[source] = 0
// Step 2 – relax edges repeatedly
   for i = 1 to V-1    // V – number of vertices
      for each edge (u, v) with weight w
         if (distance[u] + w) is less than distance[v]
            distance[v] = distance[u] + w
            parent[v] = u

// Step 3 – check for negative-weight cycles
for each edge (u, v) with weight w
   if (distance[u] + w) is less than distance[v]
      return "Graph contains a negative-weight cycle"

return distance[], parent[]
```

Output:

Shortest path from source (5)

Vertex 5 -> cost=0 parent=0

Vertex 1-> cost=6 parent=2

Vertex 2-> cost=3 parent=4

Vertex 3-> cost =3 parent =4

Vertex 4-> cost =2 paren=5

Implementation:

```
#include <stdio.h>
#include <limits.h>
// Define the maximum number of vertices in the graph
#define MAX_VERTICES 100
// Define structure for representing edges
struct Edge {
    int source, destination, weight;
};
// Define structure for representing a graph
struct Graph {
    int vertices, edges;
    struct Edge edge[MAX_VERTICES];
};
// Initialize the graph
void initGraph(struct Graph *graph, int vertices, int edges) {
    graph->vertices = vertices;
    graph->edges = edges;
}
// Bellman-Ford algorithm
void bellmanFord(struct Graph *graph, int source) {
    int distance[MAX_VERTICES];
    // Initialize distances from source to all other vertices as INFINITY
    int i, j;
    for (i = 0; i < graph->vertices; i++) {
        distance[i] = INT_MAX;
    }
    distance[source] = 0; // Distance from source to itself is 0
```

```
    // Relax all edges for V-1 times
    for (i = 1; i <= graph->vertices - 1; i++) {
        for (j = 0; j < graph->edges; j++) {
            int u = graph->edge[j].source;
            int v = graph->edge[j].destination;
            int weight = graph->edge[j].weight;
            if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight;
            }
        }
    }
    // Check for negative weight cycles
    for (i = 0; i < graph->edges; i++) {
        int u = graph->edge[i].source;
        int v = graph->edge[i].destination;
        int weight = graph->edge[i].weight;
        if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }
    // Print the distances
    printf("Vertex   Distance from Source\n");
    for (i = 0; i < graph->vertices; i++) {
        printf("%d \t\t %d\n", i, distance[i]);
    }
}
```

```
int main() {
    struct Graph graph;
    int vertices, edges, source;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &vertices, &edges);
    initGraph(&graph, vertices, edges);
    printf("Enter source vertex: ");
    scanf("%d", &source);
    printf("Enter edges (source destination weight):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d %d", &graph.edge[i].source, &graph.edge[i].destination,
&graph.edge[i].weight);
    }
    bellmanFord(&graph, source);
    return 0;
}
```

Conclusion: The implementation of the Bellman-Ford algorithm proved effective in finding the shortest paths in weighted graphs. Through rigorous testing and analysis, the algorithm demonstrated its reliability and efficiency in solving the single-source shortest path problem, offering valuable insights for real-world applications in network routing and optimization.