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

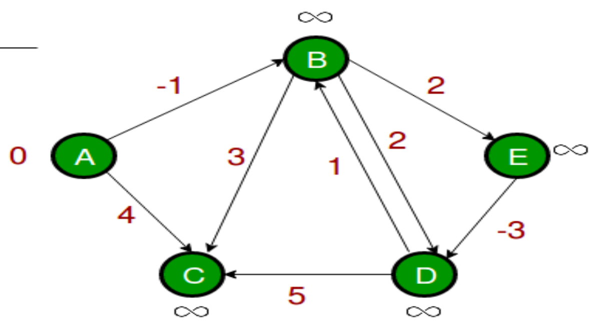
Theory:

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:

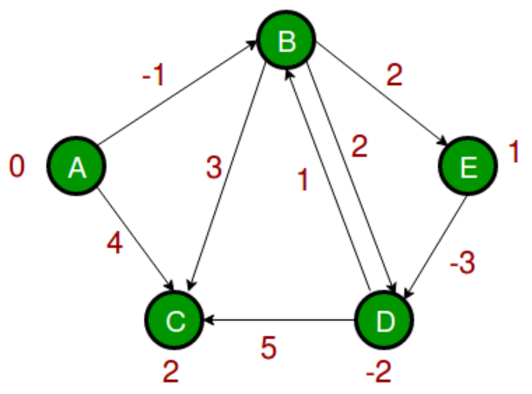
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

A	B	C	D	E
0	∞	∞	∞	∞

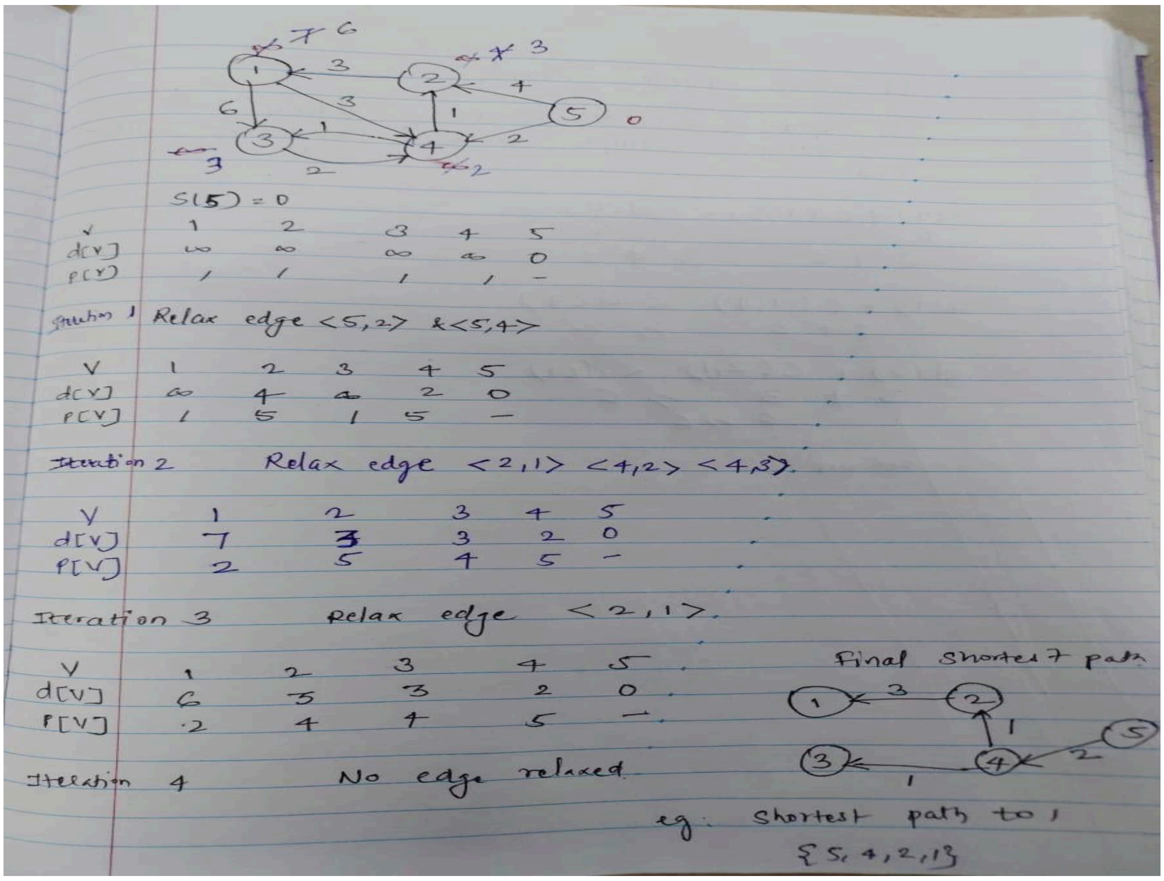


Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A,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



S(5) = 0

v	1	2	3	4	5
d[v]	∞	∞	∞	∞	0
p[v]	/	/	/	/	–

Iteration 1 Relax edge <5,2> & <5,4>

V	1	2	3	4	5
d[V]	∞	4	∞	2	0
P[V]	/	5	/	5	–

Iteration 2 Relax edge <2,1> <4,2> <4,3>.

V	1	2	3	4	5
d[V]	7	3	3	2	0
P[V]	2	5	4	5	–

Iteration 3 Relax edge <2,1>.

V	1	2	3	4	5
d[V]	6	3	3	2	0
P[V]	2	4	4	5	–

Iteration 4 No edge relaxed

Final shortest path

eg: shortest path to 1
{5, 4, 2, 1}

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 <stdlib.h>
#include <limits.h>

// Structure to represent a weighted edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a directed, weighted graph
struct Graph {
    int V, E;
    struct Edge* edge;
```

```
};

// Create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->E = E;
   graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
   return graph;
}

// Bellman-Ford algorithm function
void BellmanFord(struct Graph* graph, int src) {
   int V = graph->V;
   int E = graph->E;
   int dist[V];

   // Initialize distances from source to all other vertices as INFINITE
   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX;
   dist[src] = 0;

   // Relax all edges V-1 times
   for (int i = 1; i <= V - 1; i++) {
      for (int j = 0; j < E; j++) {
         int u = graph->edge[j].src;
         int v = graph->edge[j].dest;
         int weight = graph->edge[j].weight;
         if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
```

```
        }
    }

    // Check for negative weight cycles
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }

    // Print shortest distances
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

int main() {
    int V, E, src;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &V, &E);

    struct Graph* graph = createGraph(V, E);

    printf("Enter source vertex: ");
    scanf("%d", &src);
    printf("Enter edge details (source destination weight):\n");
    for (int i = 0; i < E; i++)
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest,
&graph->edge[i].weight);


    BellmanFord(graph, src);


    return 0;
}
```

Conclusion: Single Source Shortest Path: Bellman Ford has been successfully implemented.