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

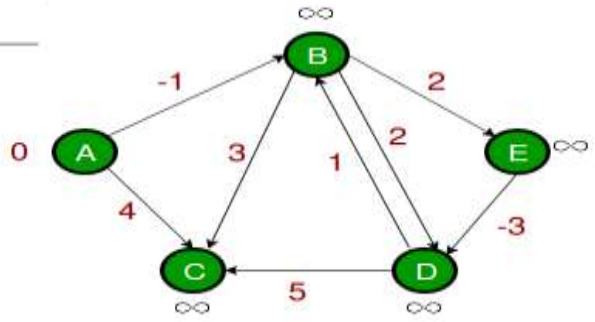
Theory:

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:

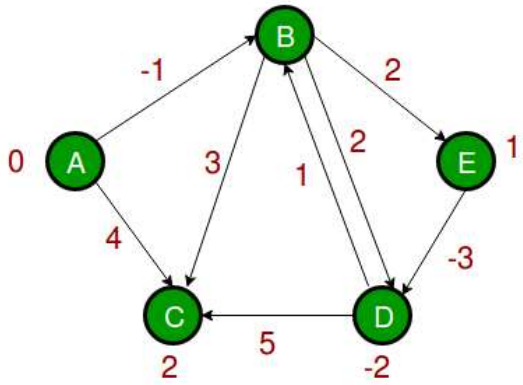
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

A	B	C	D	E
0	∞	∞	∞	∞

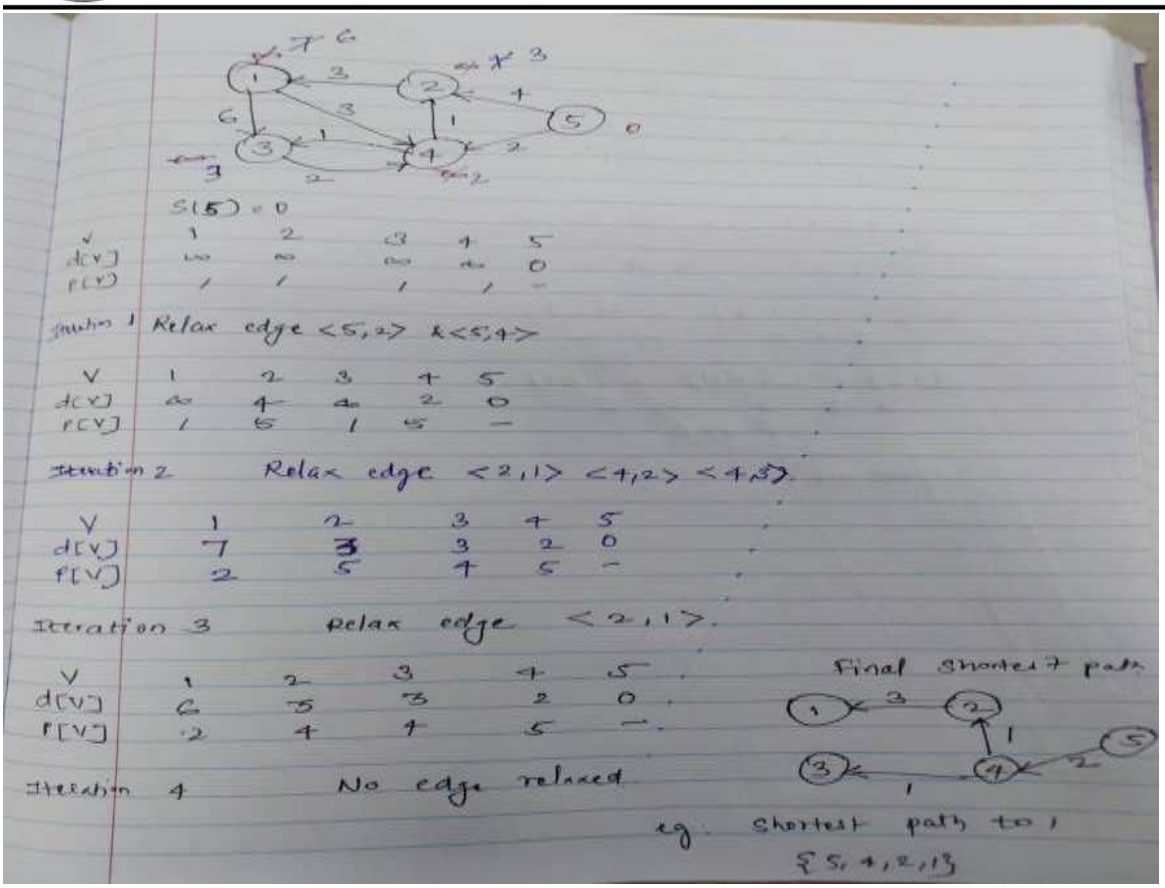


Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A,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

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define INF INT_MAX
#define MAX_VERTICES 100
#define MAX_EDGES 100
```

```c
// Structure to represent a weighted edge
struct Edge {
    int source, destination, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E;
    struct Edge edge[MAX_EDGES];
};

// Function to print the solution
void printSolution(int dist[], int n) {
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Bellman-Ford algorithm
void BellmanFord(struct Graph* graph, int source) {
    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] = INF;
    dist[source] = 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].source;
            int v = graph->edge[j].destination;
            int weight = graph->edge[j].weight;
            if (dist[u] != INF && 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].source;
        int v = graph->edge[i].destination;
        int weight = graph->edge[i].weight;
        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return;
        }
    }

    // Print the distances
    printSolution(dist, V);
}

int main() {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = 5;  // Number of vertices
    graph->E = 8;  // Number of edges

    // Add the edges
    graph->edge[0].source = 0;
    graph->edge[0].destination = 1;
    graph->edge[0].weight = -1;

    graph->edge[1].source = 0;
    graph->edge[1].destination = 2;
    graph->edge[1].weight = 4;

    graph->edge[2].source = 1;
    graph->edge[2].destination = 2;
    graph->edge[2].weight = 3;

    graph->edge[3].source = 1;
    graph->edge[3].destination = 3;
    graph->edge[3].weight = 2;

    graph->edge[4].source = 1;
    graph->edge[4].destination = 4;
    graph->edge[4].weight = 2;
```

```
    graph->edge[5].source = 3;
    graph->edge[5].destination = 2;
    graph->edge[5].weight = 5;

    graph->edge[6].source = 3;
    graph->edge[6].destination = 1;
    graph->edge[6].weight = 1;

    graph->edge[7].source = 4;
    graph->edge[7].destination = 3;
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);  // Source vertex is 0

    free(graph);
    return 0;
}
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SEARCH ERROR

PS C:\Users\Lenovo\Downloads\AOA Experiments> cd "c:\Users\Lenovo\Downloads\AOA Experiments\" ; if ($?) { gcc
.c -o bellman } ; if ($?) { .\bellman }
Vertex   Distance from Source
0                0
1                -1
2                2
3                -2
4                1
PS C:\Users\Lenovo\Downloads\AOA Experiments>
```

Conclusion:

In conclusion, the Bellman-Ford algorithm efficiently computes the shortest paths from a single source vertex to all other vertices in a weighted graph, even in the presence of negative weight edges, as long as there are no negative weight cycles reachable from the source vertex.

The algorithm iteratively relaxes the edges of the graph |V| - 1 times, where |V| is the number of vertices, ensuring that the shortest path distances are computed accurately. After the relaxation

process, it checks for negative weight cycles by iterating through all edges again. If a shorter path is found during this step, it indicates the presence of a negative weight cycle in the graph.

This implementation of the Bellman-Ford algorithm initializes the distances from the source vertex to all other vertices as infinity initially, except for the source vertex itself, which is set to 0. It then iteratively updates the distances based on the edge weights, ensuring that the shortest path distances are computed correctly.