

Autonomous Institute of Government of Maharashtra)

Name :	PRN : 2241004
Class : T. Y. B.Tech Computer	Academic Year : 2024-25
Subject : DAA Lab	Course Teacher : Mr. Vinit Kakde
Date of Performance :	Date of Completion :

PRACTICAL NO.

Aim:

To implement the Bellman-Ford Algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph, including those with negative weights.

Theory:

What is the Bellman-Ford Algorithm?

The Bellman-Ford Algorithm is a shortest path algorithm that computes the shortest path from a single source vertex to all other vertices in a weighted graph, even when the graph contains negative weight edges.

It is based on the concept of Dynamic Programming and follows a bottom-up approach. The algorithm repeatedly relaxes all the edges of the graph to gradually update the distance values of each vertex.

Key Concepts:

- Relaxation Principle: Gradually update the shortest distance to each vertex.
 - Negative Weight Edges: Bellman-Ford can handle negative weights, unlike Dijkstra's algorithm.
 - Negative Weight Cycles: The algorithm can detect negative weight cycles, which cause the shortest path to become undefined due to infinite loop reduction.
-

Why Should You Be Cautious With Negative Weights?

- Negative weight cycles can cause incorrect results.
- Some algorithms like Dijkstra's cannot handle negative weights.
- Bellman-Ford detects such cycles and warns if they exist.

How Does the Bellman-Ford Algorithm Work?

1. Initialize the distance to the source node as 0 and all others as infinity.
 2. Relax all edges ($V - 1$) times, where V is the number of vertices.
 3. Check for negative weight cycles in one additional iteration.
 4. If any distance can still be minimized, the graph contains a negative weight cycle.
-

Steps to Implement Bellman-Ford:

1. List all edges of the graph.
 2. Perform $V - 1$ iterations:
 - For each edge (u, v) , if $\text{distance}[v] > \text{distance}[u] + \text{weight}(u, v)$, update $\text{distance}[v]$.
 3. Check for negative cycles in one more iteration:
 - If you can still relax any edge, report a negative weight cycle.
-

Pseudocode:

function BellmanFord(Graph, source):

$\text{distance}[] = \text{initialize distances to all vertices as infinity}$

$\text{distance}[\text{source}] = 0$

 for i from 1 to $|V| - 1$:

 for each edge (u, v) in Graph:

 if $\text{distance}[u] + \text{weight}(u, v) < \text{distance}[v]$:

$\text{distance}[v] = \text{distance}[u] + \text{weight}(u, v)$

 // Check for negative-weight cycles

 for each edge (u, v) in Graph:

 if $\text{distance}[u] + \text{weight}(u, v) < \text{distance}[v]$:

```
print("Graph contains a negative weight cycle")
```

```
return
```

```
return distance[]
```

Time Complexity:

Operation	Complexity
Initializing distances	$O(V)$
Relaxing all edges (V

Checking for negative-weight cycles	$O(E)$
Overall Time Complexity	$O(V * E)$

Space Complexity of Bellman-Ford Algorithm:

Component	Space Complexity
Distance array	$O(V)$
Edge list (if explicitly stored)	$O(E)$
Overall Space Complexity	$O(V + E)$

- V represents the number of vertices in the graph.
- E represents the number of edges in the graph.

PROGRAM:

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

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;
}

void BellmanFord(struct Graph* graph, int src) {
    int V = graph->V;
    int E = graph->E;
    int dist[V];
    int i, j;

    // Initialize distances
    for (i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Relax edges |V| - 1 times
    for (i = 1; i <= V - 1; i++) {
        for (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 (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]) {
        printf("Graph contains a negative weight cycle\n");
        return;
    }
}

// Print the shortest distance from source to each vertex
printf("\nVertex\tDistance from Source\n");
for (i = 0; i < V; i++) {
    if (dist[i] == INT_MAX)
        printf("%d\tINF\n", i);
    else
        printf("%d\t%d\n", i, dist[i]);
}
}

int main() {
    int V, E, src, i;

    // Take input from user
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);

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

    printf("Enter each edge as: source destination weight\n");
    for (i = 0; i < E; i++) {
        printf("Edge %d: ", i + 1);

```

```

        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph->edge[i].weight);
    }

    printf("Enter the source vertex: ");
    scanf("%d", &src);

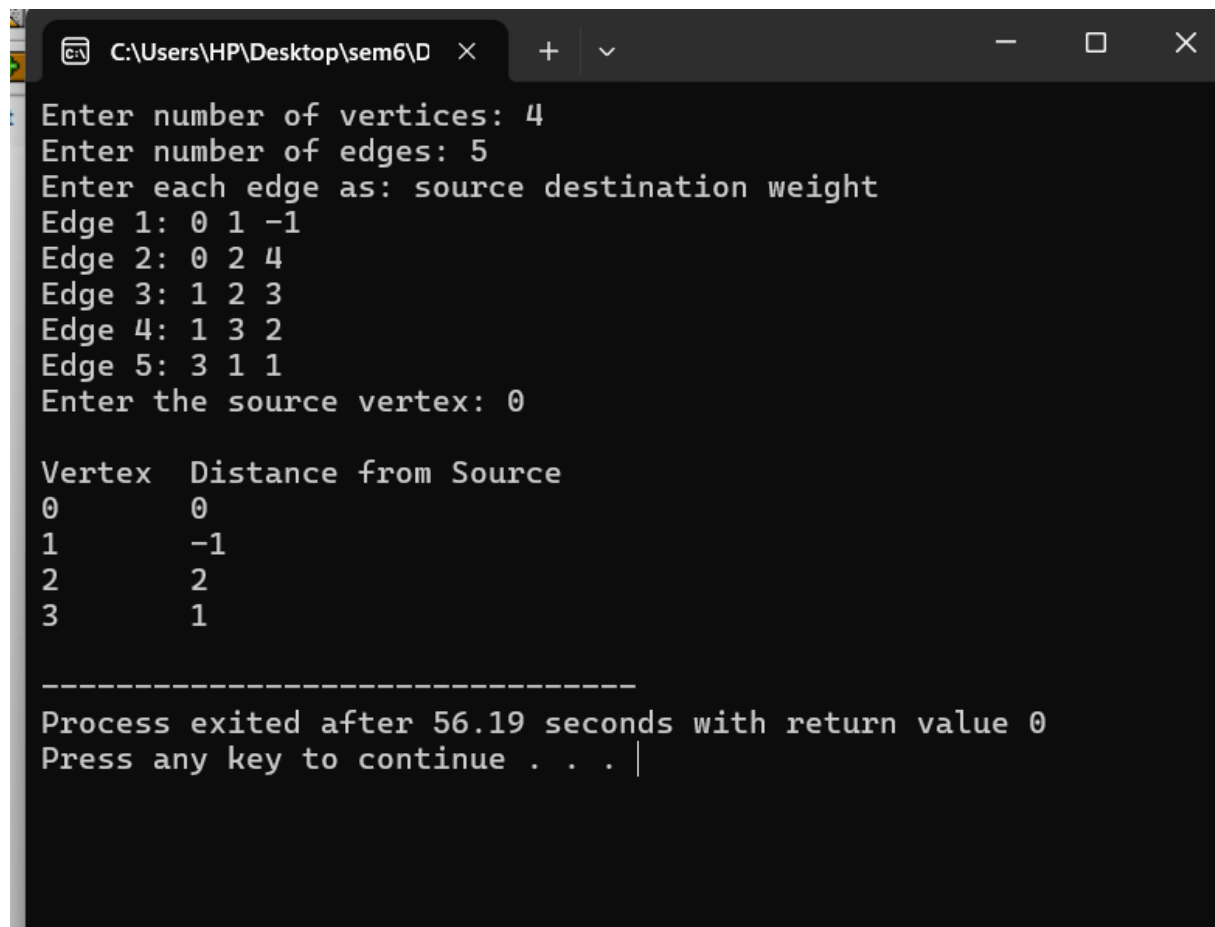
    // Call Bellman-Ford algorithm
    BellmanFord(graph, src);

    // Free allocated memory
    free(graph->edge);
    free(graph);

    return 0;
}

```

#OUTPUT:



```

C:\Users\HP\Desktop\sem6\D >
Enter number of vertices: 4
Enter number of edges: 5
Enter each edge as: source destination weight
Edge 1: 0 1 -1
Edge 2: 0 2 4
Edge 3: 1 2 3
Edge 4: 1 3 2
Edge 5: 3 1 1
Enter the source vertex: 0

Vertex  Distance from Source
0          0
1         -1
2          2
3          1

-----
Process exited after 56.19 seconds with return value 0
Press any key to continue . . . |

```

QUESTIONS:

- 1. What is the time complexity of the Bellman-Ford algorithm?**
 - Explain how the time complexity is derived and why it is important to understand for large graphs.
- 2. How does the Bellman-Ford algorithm handle negative weight edges?**
 - Discuss the behavior of the algorithm when negative weight edges are present, and how it detects negative weight cycles.
- 3. What is the difference between the Bellman-Ford algorithm and Dijkstra's algorithm?**
 - Compare the two algorithms in terms of their functionality, time complexity, and handling of negative weights.
- 4. How many times does the Bellman-Ford algorithm relax the edges?**
 - Explain the number of times the edges are relaxed and the reasoning behind the number of iterations.
- 5. Can the Bellman-Ford algorithm detect a negative weight cycle in the graph?**
 - Describe how the Bellman-Ford algorithm detects negative weight cycles and the significance of this feature.

Conclusion:

The Bellman-Ford algorithm is a powerful tool for finding the shortest paths in a graph, particularly when the graph contains edges with negative weights. It is versatile in detecting negative weight cycles, which distinguishes it from other shortest path algorithms like Dijkstra's. Although the time complexity of Bellman-Ford is $O(V * E)$, making it less efficient for large graphs, its ability to handle negative weights and cycle detection makes it indispensable in various scenarios. The practical demonstrates how the algorithm works through iterative relaxation of edges and ensures that the shortest path is determined, even in the presence of negative weight cycles, while highlighting the algorithm's strengths and limitations in real-world applications.

Sign of course Teacher

Mr. Vinit Kakde