Khulna University of Engineering & Technology

KUET

SESSIONAL REPORT

Course No. : CSE 2202

Department Of Computer Science and Engineering

Experiment No. : 02

Name of the Experiment : Implementation and Analysis of Bellman-Ford shortest path algorithm

Remarks

Name :Dadhichi Sarker Shayon

Roll No. : 2207118

Group No. : B2

Date of Performance : 13/10/2025

Year : 2nd

Date of Submission :20/10/2025

Term : 2nd

Title:

Implementation and analysis of Bellman-Ford shortest path algorithm.

Objective:

1.Implementing the Bellman-Ford algorithm to find the shortest paths from a given source vertex to all other vertices in a weighted graph.
2.Analyzing the time complexity and performance of the Bellman-Ford algorithm.
3.Analyzing cycle detection using bellman ford algorithm.
4.Analyzing limitations of bellman ford algorithm.

Theory:

The Bellman-Ford algorithm was proposed by Richard Bellman and Lester Ford in 1958. It is a dynamic programming based algorithm used to find the shortest paths from a single source to all vertices in a weighted directed graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle negative edge weights, making it more versatile in certain applications. However, it cannot handle negative weight cycles.But it can detect negative weight cycles.Bellman-Ford works by repeatedly relaxing all edges of the graph $V-1$ times ,where V is the number of vertices.

Working Principle:

Initializing all vertex distances as infinity except for the source vertex, which has distance zero.For each vertex, performing $V - 1$ iterations where all edges (u, v) are relaxed:If dist[u] + weight(u, v) <dist[v], then updating dist[v].Repeating this process $V - 1$ times so that all shortest paths are found.Finally, running one more pass through all edges. If a shorter path is still found, it indicates a negative weight cycle.This ensures that each edge is relaxed multiple times until the shortest distances are found.
Cycle Detection in Bellman-Ford Algorithm:
One of the most important features of the Bellman-Ford algorithm is its ability to detect negative weight cycles in a graph. After performing $V-1$ relaxation steps ,where V is the number of vertices, the algorithm performs one additional iteration over all edges. If any distance can still be reduced during this step, it indicates the presence of a cycle whose total weight is negative. This happens because continuous relaxation implies that there exists a path whose cost can be indefinitely reduced, which is only possible in a negative-weight cycle. Bellman-Ford can handle negative edges and explicitly identify cycles.

Pseudocode:

Pseudo code for Bellman Ford:

```
Bellman_Ford (Graph,Source,Dist[])
Initialize dist[] of size V with infinity
Set dist[source] = 0
For i = 1 to V - 1:
    For each edge (u, v, w):
        If dist[u] + w < dist[v]:
            dist[v] = dist[u] + w
For each edge (u, v, w):
    If dist[u] + w < dist[v]:
        Print "Graph contains a negative weight cycle"
        Exit
Return dist[]
```

Pseudo code for negative cycle detection using Bellman Ford:

```
 BellmanFordCycleDetect(Graph, source):
    n =number of vertices in Graph
    dist[0 to n-1] =infinity
dist[source] = 0
for i from 1 to n-1:
        for each vertex u in Graph:
            for each edge (u, v, w) outgoing from u:
                if dist[u] !=infinity and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

    for each vertex u in Graph:
        for each edge (u, v, w) outgoing from u:
            if dist[u] !=infinity and dist[u] + w < dist[v]:
                return True
return False
```

Implementation:

Bellman-Ford:

```cpp
#include <bits/stdc++.h>
using namespace std;

void add_edge(int u, int v, int w, vector<vector<pair<int, int>>> &edges)
{
    edges[u].push_back({v, w});
}

void Bellman_Ford(int src, vector<vector<pair<int, int>>> &edges, vector<int> &dist)
{
    int n = edges.size();
    dist[src] = 0;
    for (int i = 0; i < n - 1; i++)
    {
        for (int u = 0; u < n; u++)
        {
            for (auto it : edges[u])
            {
                int v = it.first;
                int w = it.second;
                if (dist[u] != INT_MAX && dist[u] + w < dist[v])
                    dist[v] = dist[u] + w;
            }
        }
    }
}
int main()
{
    int m, n;
    cin >> m >> n;
    vector<vector<pair<int, int>>> edges(m);
    for (int i = 0; i < n; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        add_edge(u, v, w, edges);
    }
    vector<int> dist(m, INT_MAX);
    int src;
  cin >> src;
    Bellman_Ford(src, edges, dist);
    for (int i = 0; i < m; i++)
    {
        if (dist[i] == INT_MAX)
            cout << "Distance of node " << i << " from source " << src << " is INF\n";
        else
            cout << "Distance of node " << i << " from source " << src << " is " << dist[i] <<
"\n";
    }
    return 0;
}
```

<u>Sample Input & Output:</u>

Input:

5 8

0 1 -1

0 2 4

1 2 3

1 3 2

1 4 2

3 2 5

3 1 1

4 3 -3

0

Output:

Distance of node 0 from source 0 is 0

Distance of node 1 from source 0 is -1

Distance of node 2 from source 0 is 2

Distance of node 3 from source 0 is -2

Distance of node 4 from source 0 is 1

Cycle detection using bellman ford:

```
#include <bits/stdc++.h>
using namespace std;

void add_edge(int u, int v, int w, vector<vector<pair<int, int>>> &edges)
{
    edges[u].push_back({v, w});
}

 bool Bellman_Ford(int src, vector<vector<pair<int, int>>> &edges)
{

  int n = edges.size();
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;
    for (int i = 0; i < n - 1; i++)
    {
        for (int u = 0; u < n; u++)
        {
            for (auto it : edges[u])
            {
                int v = it.first;
                int w = it.second;
                if (dist[u] != INT_MAX && dist[u] + w < dist[v])
               dist[v] = dist[u] + w;
            }
        }
    }

 for (int u = 0; u < n; u++)
```

```cpp
    {
        for (auto it : edges[u])
        {
            int v = it.first;
            int w = it.second;
            if (dist[u] != INT_MAX && dist[u] + w < dist[v])
                return true;
        }
    }
    return false;
}
int main()
{
    int m, n;
    cin >> m >> n;
    vector<vector<pair<int, int>>> edges(m);
    for (int i = 0; i < n; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        add_edge(u, v, w, edges);
    }
    int src;
    cin >> src;
    bool b = Bellman_Ford(src, edges);
    if (b)
        cout << "Graph contains negative weight cycle\n";
    else
        cout << "Graph doesn't contain negative weight cycle\n";
    return 0;
}
```

Sample Input & Output:

Input:

```
3 3
0 1 1
1 2 -2
2 0 -2
0
```

Output:

```
Graph contains negative weight cycle.
```

Time Complexity:

The Bellman-Ford algorithm has a time complexity of O(V × E) and space complexity of O(V).

Discussion:

The Bellman-Ford algorithm is a dynamic programming-based shortest path algorithm that can handle negative edge weights, unlike Dijkstra's. The algorithm relaxes all edges repeatedly so that shortest paths are accurately computed even when negative weights exist.
In each iteration, the algorithm checks every edge and updates the shortest distance to the target node if a shorter path is found through the source node. After $V-1$ iterations, all possible paths of at most $V-1$ edges are guaranteed to have been considered. The final iteration helps detect if any edge can still be relaxed indicating a negative cycle.
Although Bellman-Ford is slower compared to Dijkstra's algorithm, it is more powerful due to its ability to handle graphs with negative weights. This makes it highly useful in applications like currency arbitrage detection, network flow optimization, and shortest path problems in graphs with both positive and negative weights.
The main disadvantage is its $O(V \times E)$ time complexity, making it less suitable for very large graphs. However, for moderate-sized graphs or when negative edges are unavoidable, Bellman-Ford remains a robust and reliable choice.

Conclusion:

Bellman-Ford is a fundamental shortest-path algorithm that extends the capabilities of Dijkstra's algorithm by handling negative edge weights. It is conceptually simple and guarantees the correct result even in graphs where Dijkstra fails. Although slower in terms of time complexity, its ability to detect negative weight cycles and handle general graphs makes it invaluable in many real-world applications such as financial modeling and network routing.

References:

1.Lab slides.
2.https://www.geeksforgeeks.org/dsa/bellman-ford-algorithm-dp-23/