Name : Sonal Dilip Gholap

UID : 2021600020

Branch : CSE-AIML

Batch : B

Experiment : 6

Aim : Experiment on Graph Algorithms

- **Objectives :**

1. To study Graph Algorithms by implementing Dijkstra’s and Bellman Ford Algorithms.
2. To derive and analyze the time complexity of Dijkstra’s and Bellman Ford Algorithms.

- **Theory :**

1. The Dijkstra algorithm is a graph traversal algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights, producing a shortest path tree.
2. It is a greedy algorithm, meaning that it always chooses the node with the smallest distance from the source at each step.
3. It is commonly used in network routing protocols, GPS systems, and in various other applications that require finding the shortest path in a graph.
4. Its time complexity is $O(|E| + |V|\log|V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph. This makes it efficient for graphs with small edge weights and sparse graphs.
5. The Bellman-Ford algorithm is a shortest path algorithm that works for graphs with negative weight edges.
6. It can handle graphs that Dijkstra's algorithm cannot, such as graphs with negative weight edges.
7. This algorithm can work with both sparse and dense graphs, although it may be less efficient on sparse graphs.

8. Dijkstra's algorithm is a greedy algorithm, while Bellman-Ford is a dynamic programming algorithm.
9. Bellman-Ford algorithm can detect negative weight cycles in a graph, while Dijkstra's algorithm cannot.

- **<u>Dijkstra’s Algorithm</u>**

<u>Pseudocode for Dijkstra’s Algorithm</u>

```
Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

<u>Derivation of Time Complexity</u>

```
for each vertex V in G
    dist[V] = ∞
    prev[V] = -1
    add V to Q

dist[source] = 0

while Q is not empty
    u = vertex in Q with min dist[u]
    remove u from Q

    for each neighbor v of u still in Q
        alt = dist[u] + graph.edges(u,v)
        if alt < dist[v]
            dist[v] = alt
            prev[v] = u
            Q.push(dist[v], v)
```

Priority Queue takes $(\log V)$ time to invusert and extract a vertex.

Inner loop has time complexity $O(E)$

Total time spent of Priority Queue = $(E+V)\log V$

Initializing distance & prev arrays takes $O(V)$ time.

$\therefore$ Total time $= O((E+V)\log V)$

Code :

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

#define INF INT_MAX

typedef pair<int, int> pii;

void dijkstra(vector<vector<pii>>& graph, int source, vector<int>& dist, vector<int>& prev) {

    int n = graph.size();

    dist.resize(n, INF);  // Initialize all distances to infinity
    prev.resize(n, -1);  // Initialize all predecessors to -1

    priority_queue<pii, vector<pii>, greater<pii>> pq;  // Create a min-heap priority queue

    pq.push({0, source});
    dist[source] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;  // Get the vertex with the smallest distance
        pq.pop();

        for (auto& neighbor : graph[u]) {  // For each neighbor v of u still in the queue
            int v = neighbor.first;
            int w = neighbor.second;
            int alt = dist[u] + w;  // Calculate the new distance

            if (alt < dist[v]) {  // If the new distance is smaller than the current distance
                dist[v] = alt;
                prev[v] = u;
                pq.push({dist[v], v});
            }
        }
    }
}

int main() {
    int n, m;
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;
    vector<vector<pii>> graph(n);

    cout << "Enter the edges and their weights (u v w):" << endl;
```

```
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].push_back({v, w});
        graph[v].push_back({u, w}); // undirected graph
    }

    int source = 0;
    vector<int> dist, prev;
    dijkstra(graph, source, dist, prev);

    cout << "Vertex \t Distance from Source\n";
    for (int i = 0; i < n; i++) {
        cout << i << "\t\t" << dist[i] << "\n";
    }

    return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-dzwv5h20.3g1' '--stdout=Microsoft-MIEngine-Out-j4u3rcyb.zgc' '--stderr=Microsoft-MIEngine-Err
or-disj3as5.byx' '--pid=Microsoft-MIEngine-Pid-neetf3ri.vbg' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter the number of vertices: 6
Enter the number of edges: 8
Enter the edges and their weights (u v w):
0 1 4
0 2 4
1 2 2
2 3 3
2 5 6
2 4 1
3 5 2
4 5 3
Vertex   Distance from Source
0               0
1               4
2               4
3               7
4               5
5               8
PS C:\C++ Learning Course>
```

- **Bellman Ford Algorithm**

Pseudocode for Bellman Ford Algorithm

```
bellmanFordAlgorithm(G, s)

  for each vertex V in G
    dist[V] <- infinite
    prev[V] <- NULL

  dist[s] <- 0

  for each vertex V in G
    for each edge (u,v) in G
      temporaryDist <- dist[u] + edgeweight(u, v)
      if temporaryDist < dist[v]
        dist[v] <- temporaryDist
        prev[v] <- u

  for each edge (U,V) in G
    If dist[U] + edgeweight(U, V) < dist[V]
      Error: Negative Cycle Exists

  return dist[], prev[]
```

Derivation of Time Complexity

Bellman Ford

The outer loop of the algorithm iterates $|V|-1$ times
The inner loop iterates $|E|$ times

	Time
for each V in G	V
for each E in G	$V \times E$
temp = dist[u] + edgeW(u,v)	$V \times E$
if temp < dist[v]	$V \times E$
dist[v] = temp	$V \times E$
prev[v] = u	$V \times E$
for each E in G	E
if dist[u] + edgeW(u,v) < dist[v]	
ERROR	

Total time $= V + E + 5(V \times E)$

$= O(V \times E)$

Code :

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

#define MAX 100001
#define INF INT_MAX
```

```
struct Edge {
    int source, destination, weight;
};

void bellmanFordAlgorithm(vector<Edge> edges, int vertices, int source) {

    int prev[MAX];
    int dist[MAX];

    for (int i = 1; i <= vertices; i++) {
        dist[i] = INF;
        prev[i] = -1;
    }

    dist[source] = 0;  // set the distance of the source vertex to 0

    for (int i = 1; i <= vertices - 1; i++) {
        for (Edge e : edges) {
            int u = e.source, v = e.destination, w = e.weight;
            if (dist[u] != INF && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                prev[v] = u;
            }
        }
    }

// iterate through all the edges again to check for negative cycles
    for (Edge e : edges) {
        int u = e.source, v = e.destination, w = e.weight;
        if (dist[u] != INF && dist[u] + w < dist[v]) {
            cout << "Error: Negative Cycle Exists\n";
            return;
        }
    }

    cout << "Vertex    Distance from source vertex\n";
    for (int i = 0; i < vertices; i++) {
        cout << i << "\t\t" << dist[i] << "\n";
    }
}

int main() {
    int vertices, edges;
    cout << "Enter the number of vertices and edges: ";
    cin >> vertices >> edges;

    vector<Edge> graph;

    cout << "Enter the edges and their weights (u v w):" << endl;
    for (int i = 0; i < edges; i++) {
```

```cpp
        int u, v, w;
        cin >> u >> v >> w;
        graph.push_back({u, v, w});
    }

    cout << "Enter source: ";
    int source;
    cin >> source;

    bellmanFordAlgorithm(graph, vertices, source);

    return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptools-1.15.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-2yume5dw.bjb' '--stdout=Microsoft-MIEngine-Out-bfn3lju0.vgp' '--stderr=Microsoft-MIEngine-Error-1ekae0tv.ndb' '--pid=Microsoft-MIEngine-Pid-rdj5tpdu.nwm' '--dbgExe=C:\Program Files (x86)\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter the number of vertices and edges: 5 7
Enter the edges and their weights (u v w):
0 1 200
0 2 -20
0 3 100
1 4 70
2 3 50
3 4 10
4 2 40
Enter source: 0
Vertex    Distance from source vertex
0               0
1               200
2               -20
3               30
4               40
PS C:\C++ Learning Course>
```

Conclusion :

1. Time complexity of Dijkstra Algorithm – **O((E+V)*logV)**
2. Time complexity of Bellman Ford Algorithm– **O(E*V)**
3. Dijkstra's algorithm is faster than Bellman-Ford algorithm when the graph has no negative weight edges, but it cannot handle negative weight edges unlike the Bellman Ford algorithm.