Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Name	Tejas Jadhav
UID No.	2022301006
Class	COMPS A (B batch)
Experiment No.	6

Aim: Single source shortest path using Dijkstra's Algorithm

Theory:

- **Greedy Alogrithms**

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

- **Dijkstra's Algorithm**

Dijkstra's algorithm is an efficient algorithm for finding the shortest path in a weighted graph with non-negative edge weights. The algorithm works by maintaining a set of nodes whose shortest distance from the source node is already known.

Initially, this set only contains the source node, and the distance to all other nodes is set to infinity. At each step, the algorithm selects the node with the smallest distance and adds it to the set of known nodes. Then, it updates the distances of all the neighboring nodes that are not already in the set of known nodes, based on the distance to the newly added node.

Algorithm:

1) Relaxation

Assume, u and v are the vertices, uv is the edge we want to relax, and w contains all the distances calculated so far.

```
Relax(u, v, w):
        if v.d > u.d + w(u,v):
                v.d = u.d + w(u,v)
```

2) Dijkstra's Algorithm

Here G is the graph and s is the source

```
Dijkstra(G, s):
  Initialize-Single-Source(G, s)
  S = empty set
  Q = priority queue containing all vertices in G
  while Q is not empty:
     u = Extract-Min(Q)
     Add u to S
     for each vertex v adjacent to u:
        if v is not in S:
           Relax(u, v, w)
```

3) Bellman Ford Algorithm

```
function bellmanFordAlgorithm(G, s) //G is the graph and s is the source vertex
  for each vertex V in G
    dist[V] <- infinite // dist is distance
     prev[V] <- NULL // prev is previous
  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[], previ[]

Code:

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

int V;

void print_solution(int distance[]) {
    cout << "Vertex\tDistance from source" << endl;
    for (int i = 0; i < V; i++) {
        cout << i << "\t\t" << distance[i] << endl;
    }
}

int* belman(int** graph, int src) {
    int distance[V];

    for (int i = 0; i < V; i++) {
        distance[i] = INT_MAX;
    }

    distance[src] = 0;

    for (int i = 0; i < V - 1; i++) {
        for (int u = 0; u < V; u++) {
            for (int v = 0; v < V; v++) {
                if (distance[u] != INT_MAX && graph[u][v] && distance[u] +
graph[u][v] < distance[v]) {
                    distance[v] = distance[u] + graph[u][v];
                }
            }
        }
    }

    int* distance_from_src = new int[V];
    for (int i = 0; i < V; i++) {
```

```
        distance_from_src[i] = distance[i];
    }

    return distance_from_src;
}

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

    int** graph = new int*[V];
    for (int i = 0; i < V; i++) {
        graph[i] = new int[V];
    }

    cout << "Enter the adjacency matrix: " << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cin >> graph[i][j];
        }
    }

    int src;
    cout << "Enter the source vertex: ";
    cin >> src;

    int* distance_from_src = belman(graph, src);
    print_solution(distance_from_src);

    return 0;
}
```

Output:

```
PS D:\Tejas\clg\daa\Experiment 06\code> g++ .\belman.cpp
PS D:\Tejas\clg\daa\Experiment 06\code> ./a
Enter the number of vertices: 6
Enter the adjacency matrix:
0 2 4 0 0 0
0 0 1 7 0 0
0 0 0 0 3 0
0 0 0 0 0 1
0 0 0 2 0 5
0 0 0 0 0 0
Enter the source vertex: 0
Vertex  Distance from source
0               0
1               2
2               3
3               8
4               6
5               9
```

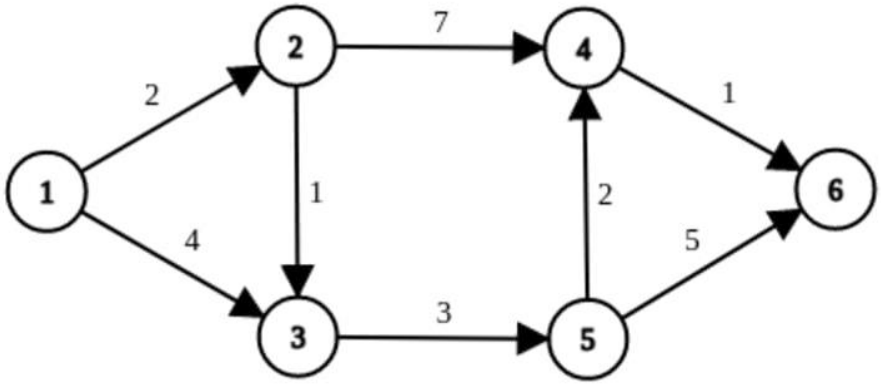
Code:

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

int V;

int min_distance(int distance[], bool visited[]) {
    int min = INT_MAX;
    int min_index = 0;

    for (int i = 0; i < V; i++) {
        if (!visited[i] && distance[i] <= min) {
            min = distance[i];
            min_index = i;
        }
    }
    return min_index;
}

void print_solution(int distance[]) {
```

```
    cout << "Vertex\tDistance from source" << endl;
    for (int i = 0; i < V; i++) {
        cout << i << "\t\t" << distance[i] << endl;
    }
}

int* dijkstras(int **graph, int src) {
    int distance[V];
    bool visited[V];

    for (int i = 0; i < V; i++) {
        // Initialize all distances as infinite and visited as false
        distance[i] = INT_MAX;
        visited[i] = false;
    }

    // Distance of source vertex from itself is always 0
    distance[src] = 0;

    for (int i = 0; i < V - 1; i++) {
        int u = min_distance(distance, visited);

        visited[u] = true;
        for (int v = 0; v < V; v++) {
            // Update distance[v] only if it is not in visited, there is an
edge from u to v, and total weight of path from src to v through u is
smaller than current value of distance[v]
            if (!visited[v] && graph[u][v] && distance[u] != INT_MAX &&
distance[u] + graph[u][v] < distance[v]) {
                distance[v] = distance[u] + graph[u][v];
            }
        }
    }
    int *distance_from_src = new int[V];
    for (int i = 0; i < V; i++) {
        distance_from_src[i] = distance[i];
    }
    return distance_from_src;
}

int main() {
```

```
    cout << "Enter the number of vertices: ";
    cin >> V;

    int **graph = new int*[V];
    for (int i = 0; i < V; i++) {
        graph[i] = new int[V];
    }

    cout << "Enter the adjacency matrix: " << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cin >> graph[i][j];
        }
    }

    int src;
    cout << "Enter the source vertex: ";
    cin >> src;

    int *distance_from_src = dijkstras(graph, src);

    print_solution(distance_from_src);

    return 0;
}
```

Input: Source vertex 1



Output:

```
PS D:\Tejas\clg\daa\Experiment 06\code> ./a
Enter the number of vertices: 6
Enter the adjacency matrix:
0 2 4 0 0 0
0 0 1 7 0 0
0 0 0 0 3 0
0 0 0 0 0 1
0 0 0 2 0 5
0 0 0 0 0 0
Enter the source vertex: 0
Vertex  Distance from source
0               0
1               2
2               3
3               8
4               6
5               9
```

Observation:

- Dijkstra's algorithm is generally faster than the Bellman-Ford algorithm for sparse graphs with non-negative edge weights.
- The Bellman-Ford algorithm can handle graphs with negative edge weights, while Dijkstra's algorithm cannot.

- Both algorithms have a time complexity of O(|E|*|V|), where |E| is the number of edges and |V| is the number of vertices in the graph, but in practice, Dijkstra's algorithm often performs better on average.

CONCLUSION:

In conclusion, the choice between bellman-ford algorithm and dijkstra's algorithm depends on the specific characteristics of the graph and the type of problem being solved. Dijkstra's algorithm is generally faster for sparse graphs with non-negative edge weights, while the bellman-ford algorithm is more suitable for graphs with negative edge weights.