Experiment No.	6
Aim	Implement Graph Algorithms
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

Theory:

Bellman-Ford

Bellman-Ford algorithm is a graph algorithm used to find the shortest path from a source vertex to all other vertices in a weighted, directed graph. It can also detect negative weight cycles in the graph. The algorithm was introduced by Richard Bellman and Lester Ford Jr. in 1958.

The Bellman-Ford algorithm maintains a set of distances from the source vertex to all other vertices in the graph. It iteratively relaxes edges in the graph, updating the distances until the optimal distances are found. The algorithm works by considering all possible paths from the source vertex to other vertices, and it gradually refines the estimates of the shortest path distances by relaxing the edges.

The main idea behind the Bellman-Ford algorithm is the principle of relaxation, which states that if there is a shorter path from the source vertex to a vertex than the current estimate, the algorithm updates the estimate with the shorter distance. The algorithm repeats this process for all vertices in the graph, iterating over the edges multiple times until convergence is reached.

One key feature of the Bellman-Ford algorithm is its ability to detect negative weight cycles in the graph. If there is a negative weight cycle in the graph, the algorithm can detect it by observing that the distance to some vertices may keep decreasing in every iteration, indicating that there is no shortest path since the distance can be made arbitrarily small by going around the negative weight cycle multiple times.

The time complexity of the Bellman-Ford algorithm is O(V * E), where V is the number of vertices and E is the number of edges in the graph. The algorithm is widely used in various applications, such as routing protocols, network analysis, and transportation systems, where finding the shortest path in a weighted graph is a fundamental problem.

Dijkstra

Dijkstra's algorithm is a widely used algorithm for finding the shortest path between two nodes in a weighted graph. It was developed by Dutch computer scientist Edsger W. Dijkstra in 1956 and is commonly used in various applications such as routing and navigation systems, network analysis, and transportation planning.

Dijkstra's algorithm works by exploring the graph from a starting node to find the shortest path to all other nodes in the graph. It maintains a set of "unvisited" nodes and repeatedly selects the node with the smallest known distance from the starting node as the current node to visit. It then updates the distances of its neighboring nodes and continues the process until all nodes have been visited or the destination node has been reached.

The algorithm uses a priority queue or a min-heap data structure to efficiently select the node with the smallest known distance. It also uses an array or a dictionary to keep track of the shortest distances from the starting node to each visited node, and another array or dictionary to store the previous node that leads to the shortest path.

Dijkstra's algorithm has a time complexity of O(|V|^2) in its basic form, where |V| is the number of nodes in the graph. However, with the use of priority queues, the time complexity can be reduced to O(|V| + |E| log |V|), where |E| is the number of edges in the graph. Dijkstra's algorithm guarantees to find the shortest path in a graph with non-negative edge weights, but it may not work correctly for graphs with negative edge weights or cycles. In such cases, other algorithms like Bellman-Ford or negative weight cycle detection algorithms may be more appropriate

Algorithm:

Bellman-Ford

Step1: Initialize the distances: Set the distance from the source vertex to itself as 0, and set the distance to all other vertices as infinity. You can also set a predecessor array to keep track of the shortest path.

Step 2: Relax edges repeatedly: Repeat the following process |V|-1 times, where |V| is the number of vertices in the graph:

- For each edge (u, v) in the graph, where u is the source vertex and v is the destination vertex, relax the edge by updating the distance to v if a shorter path is found. The relaxation step can be done by comparing the distance from the source vertex to u plus the weight of the edge (u, v) with the current distance to v. If the distance to v can be improved, update it and set the predecessor of v as u.

Step 3: Check for negative cycles: After |V|-1 iterations, check for the presence of negative cycles in the graph. To do this, repeat the relaxation step once more. If any distance can still be improved, it means there is a negative cycle in the graph, and the algorithm cannot find a shortest path. In this case, you can halt the algorithm and report the presence of a negative cycle.

Step 4: Extract shortest paths: If there are no negative cycles, you can extract the shortest paths from the source vertex to all other vertices using the predecessor array. Starting from the destination vertex, follow the predecessor links backward until you reach the source vertex, and construct the shortest path accordingly.

Dijkstra

Step 1: Initialize the algorithm

- Create a set of unvisited nodes and mark them as "unvisited".
- Set the distance of the starting node to 0, and the distance of all other nodes to infinity.
- Set the starting node as the current node.

Step 2: Visit the neighbors

- For the current node, calculate the distance to its neighbors through the edges.

- If the calculated distance to a neighbor is less than the current known distance, update the distance and set the current node as the previous node of the neighbor.
- Mark the current node as "visited".

Step 3: Select the next node

- Select the unvisited node with the smallest known distance as the next current node.
- If there are no unvisited nodes left, or if the destination node has been visited, the algorithm terminates.

Step 4: Repeat the process

- Repeat Steps 2 and 3 until the destination node is visited or there are no more unvisited nodes.

Step 5: Trace the path

- Once the destination node is visited, trace back the shortest path from the destination node to the starting node using the previous node information that was stored during the algorithm.
- This will give the shortest path from the starting node to the destination node.

Program:

Bellman-Ford

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

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E;
    struct Edge* edges;
};
```

```c
// Function to create a new graph
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (struct Edge*) malloc(E * sizeof(struct Edge));
    return graph;
}

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

// Bellman-Ford algorithm function
void bellmanFord(struct Graph* graph, int src) {
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Initialize distances from source to all vertices as infinity
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
    }

    // Distance from source to itself is 0
    dist[src] = 0;

    // Relax all edges V-1 times
    for (int i = 0; i < V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edges[j].src;
            int v = graph->edges[j].dest;
            int weight = graph->edges[j].weight;
            if (dist[u] != INT_MAX && 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->edges[i].src;
        int v = graph->edges[i].dest;
        int weight = graph->edges[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative-weight cycle\n");
            return;
        }
    }

    printDistances(dist, V);
}

int main(){
    int V,E,src,i;
    printf("Enter Number of Vertices: ");
    scanf("%d",&V);
    printf("Enter Number of Edges: ");
    scanf("%d",&E);
    struct Graph* graph = createGraph(V, E);
    printf("\nEnter Source, Destination and Weight for each edge:-\n");
    for (i=0;i<E;i++){
        printf("\nEdge %d\n",(i+1));
        printf("Source: ");
        scanf("%d",&graph->edges[i].src);
        printf("Destination: ");
        scanf("%d",&graph->edges[i].dest);
        printf("Weight: ");
        scanf("%d",&graph->edges[i].weight);
    }
    printf("\n\nEnter Source Vertex: ");
    scanf("%d",&src);
    printf("Bellman-Ford Algorithm:\n");
    printf("Source vertex: %d\n", src);
    bellmanFord(graph, src);
    return 0;
}
```

Dijkstra

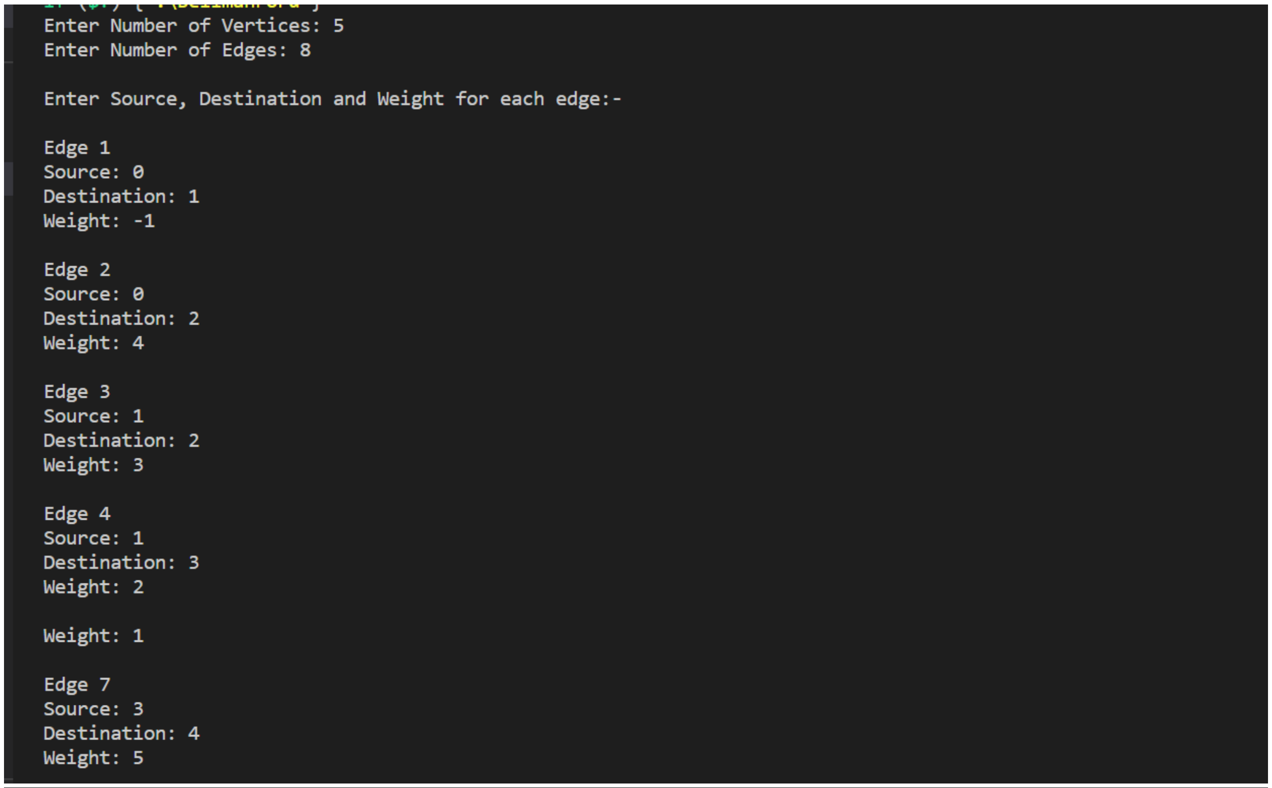
```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

int V;
```

```
// Function to find the vertex with the minimum distance value
int minDistance(int dist[], bool sptSet[]){
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++){
        if (sptSet[v] == false && dist[v] <= min){
            min = dist[v], min_index = v;
        }
    }

    return min_index;
}

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

// Function that implements Dijkstra's algorithm
void dijkstra(int graph[V][V], int src){
    int dist[V];     // Array to store the shortest distance from src to each
vertex
    bool sptSet[V]; // Array to keep track of vertices included in shortest path
tree

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++){
        dist[i] = INT_MAX, sptSet[i] = false;
    }
    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++){
        // Pick the minimum distance vertex from the set of vertices not yet
processed
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist[] value of the adjacent vertices of the picked vertex
```

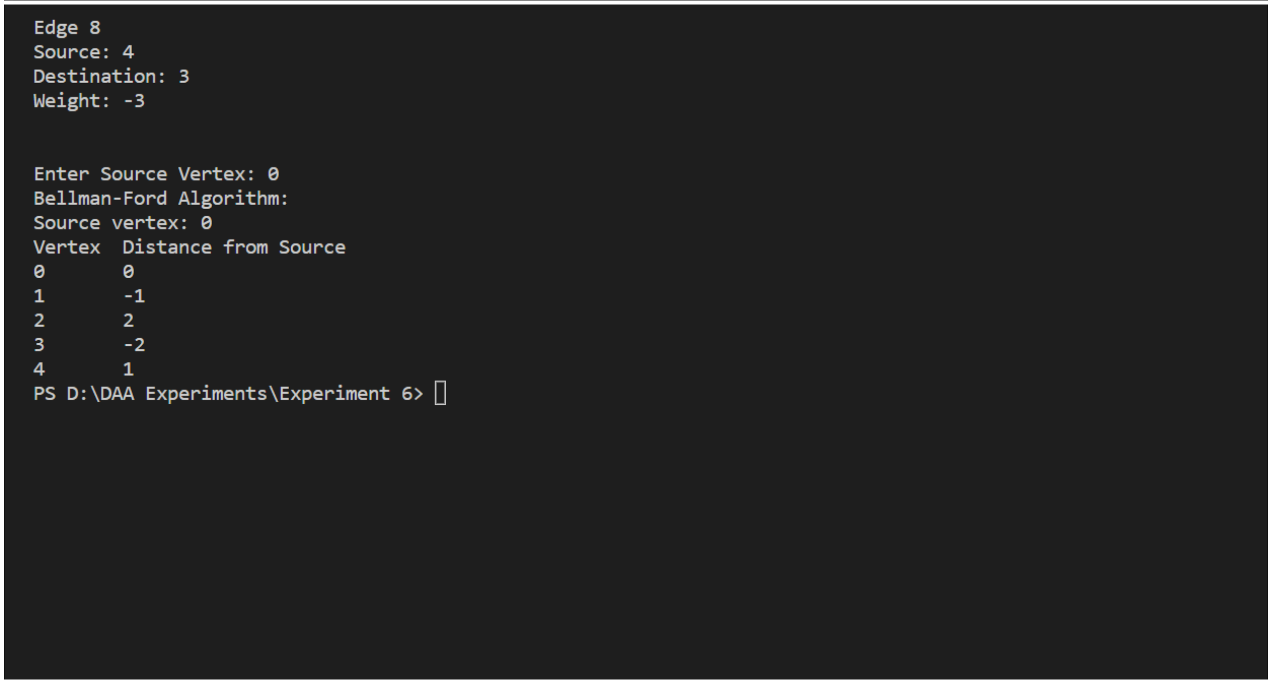
```c
        for (int v = 0; v < V; v++){
            // Update dist[v] only if it's not in sptSet, there is an edge from u
to v,
// and total weight of path from src to v through u is smaller than current value
of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]){
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Print the constructed distance array
    printSolution(dist, V);
}

int main(){
    printf("Enter Number of Vertices: ");
    scanf("%d",&V);
    int graph[V][V],src,i,j;
    printf("\nEnter the adjacency matrix of the graph:-\n\n");
    for(i=0;i<V;i++){
        for(j=0;j<V;j++){
            scanf("%d",&graph[i][j]);
        }
    }
    printf("\n\nEnter Source Vertex: ");
    scanf("%d",&src);
    printf("\nShortest path distances from source vertex %d:\n", src);
    dijkstra(graph, src);
    return 0;
}
```

Results:

Bellman-Ford

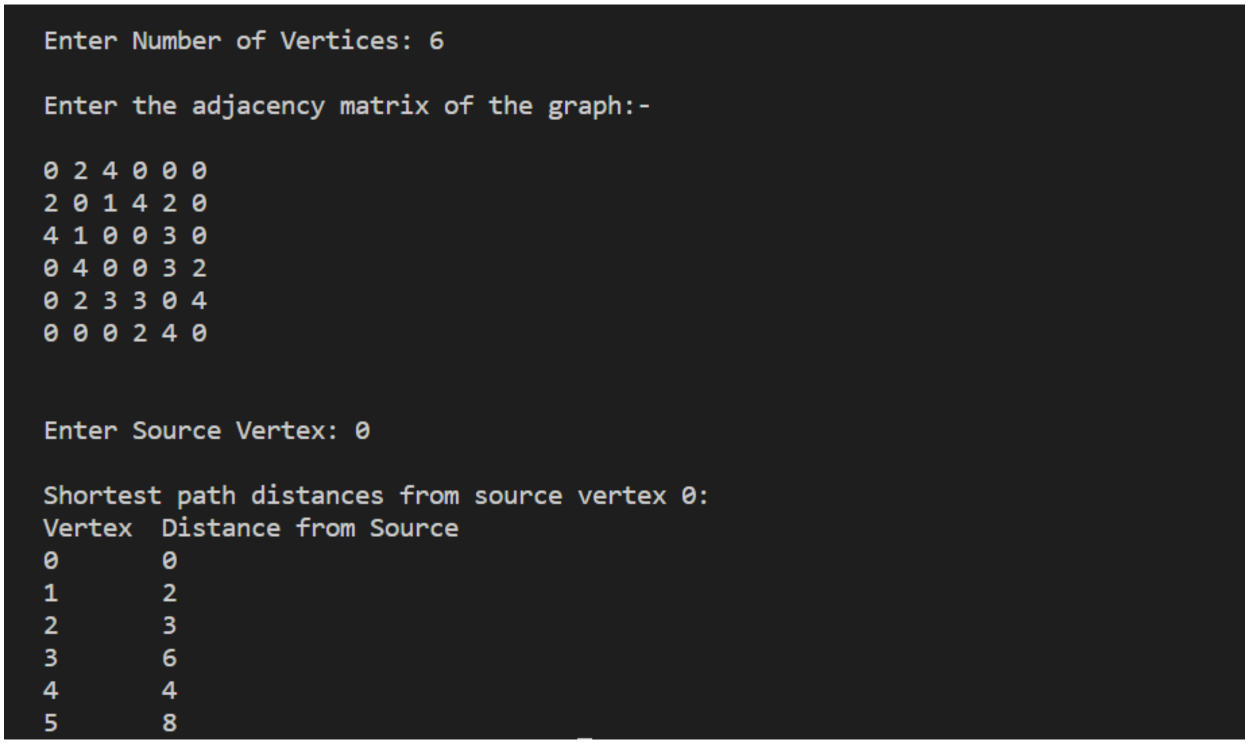


```
Enter Number of Vertices: 5
Enter Number of Edges: 8

Enter Source, Destination and Weight for each edge:-

Edge 1
Source: 0
Destination: 1
Weight: -1

Edge 2
Source: 0
Destination: 2
Weight: 4

Edge 3
Source: 1
Destination: 2
Weight: 3

Edge 4
Source: 1
Destination: 3
Weight: 2

Weight: 1

Edge 7
Source: 3
Destination: 4
Weight: 5
```



```
Edge 8
Source: 4
Destination: 3
Weight: -3


Enter Source Vertex: 0
Bellman-Ford Algorithm:
Source vertex: 0
Vertex  Distance from Source
0       0
1       -1
2       2
3       -2
4       1
PS D:\DAA Experiments\Experiment 6> 
```

Dijkstra



```
Enter Number of Vertices: 6

Enter the adjacency matrix of the graph:-

0 2 4 0 0 0
2 0 1 4 2 0
4 1 0 0 3 0
0 4 0 0 3 2
0 2 3 3 0 4
0 0 0 2 4 0


Enter Source Vertex: 0

Shortest path distances from source vertex 0:
Vertex  Distance from Source
0       0
1       2
2       3
3       6
4       4
5       8
```

Conclusion:

The experiments on Dijkstra's algorithm and Bellman-Ford algorithm provide insights into their performance and characteristics in solving the shortest path problem in weighted graphs. Dijkstra's algorithm is efficient, suitable for graphs with non-negative weights, and has a time complexity of O(V^2) or O(V + E log V). It is ideal for small, sparse graphs. Bellman-Ford algorithm can handle negative weights and detect negative weight cycles, but has a time complexity of O(V * E), making it less efficient. It is suitable for scenarios where negative weights or cycles may exist. The choice between the two algorithms depends on the specific requirements of the problem, such as the nature of the graph, presence of negative weights or cycles, and desired efficiency. Further experimentation and analysis can provide deeper insights into their performance in real-world scenarios.