Assignment No - 9

PROBLEM STATEMENT 1:

"Write a Program to accept a graph from a user and represent it with Adjacency Matrix and perform BFS and DFS traversals on it. "

OBJECTIVES:

To design and implement a program that accepts a graph from the user and represents it using an adjacency matrix. To understand and apply Breadth-First Search (BFS) and Depth-First Search (DFS) traversal techniques on a given graph. To study the difference between level-wise traversal (BFS) and depth-wise traversal (DFS). To gain practical experience in implementing graph traversal algorithms using programming logic and data structures.To analyze real-world applications of BFS and DFS in areas such as networking, pathfinding, and social media graph analysis.

THEORY:

A graph is a non-linear data structure consisting of vertices (nodes) and edges (links) that connect pairs of vertices. Graphs are widely used to represent relationships or connections in real-world systems, such as computer networks, transportation routes, and social networks. One common method to represent a graph in a computer program is through an Adjacency Matrix, which is a two-dimensional array where each cell (i, j) indicates whether an edge exists between vertex i and vertex j. This structure is easy to implement and ideal for dense graphs.

Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental graph traversal algorithms. BFS uses a queue and explores all neighboring nodes before moving to the next level, making it suitable for finding the shortest path in unweighted graphs. In contrast, DFS uses recursion or a stack and explores nodes deeply along one branch before backtracking, which is useful in tasks like detecting cycles or performing topological sorting. Understanding BFS and DFS helps in solving a wide range of computational problems efficiently and forms the foundation of many advanced algorithms in computer science.

ALGORITHM:

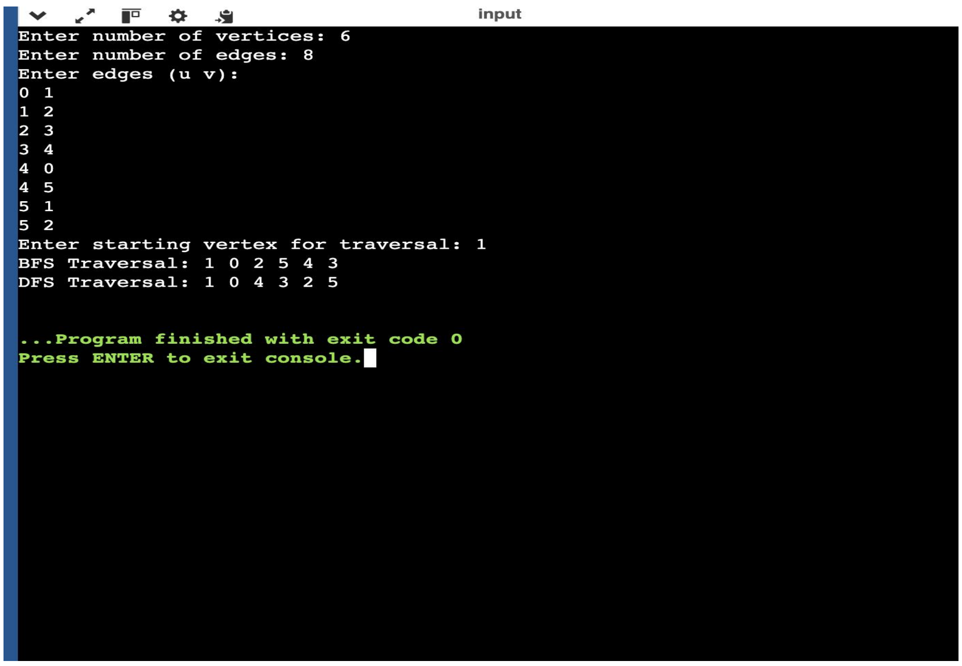
```
procedure create_graph()
   input n        // number of vertices
   create n x n matrix adj initialized to 0
   input e        // number of edges
   for i = 1 to e do
      input u, v
```

```
        adj[u][v] ← 1
        adj[v][u] ← 1      // for undirected graph
    end for
end procedure
procedure BFS(start)
    create an empty queue Q
    create visited[n] initialized to false

    visited[start] ← true
    enqueue(Q, start)

    while Q is not empty do
        vertex ← dequeue(Q)
        print vertex

        for i = 0 to n-1 do
            if adj[vertex][i] = 1 and visited[i] = false then
                visited[i] ← true
                enqueue(Q, i)
            end if
        end for
    end while
end procedure
end bfs
procedure DFS(vertex)
    visited[vertex] ← true
    print vertex

    for i = 0 to n-1 do
        if adj[vertex][i] = 1 and visited[i] = false then
            DFS(i)
        end if
    end for
end procedure
```

CODE:

```
#include <iostream>
#include <queue>
using namespace std;
#define MAX 20

int adj[MAX][MAX];
bool visited[MAX];
int n;

void BFS(int start) {
    queue<int> q;
    bool visitedBFS[MAX] = {false};
    q.push(start);
    visitedBFS[start] = true;

    cout << "BFS Traversal: ";
    while (!q.empty()) {
        int vertex = q.front();
        q.pop();
        cout << vertex << " ";

        for (int i = 0; i < n; i++) {
            if (adj[vertex][i] == 1 && !visitedBFS[i]) {
                q.push(i);
                visitedBFS[i] = true;
            }
        }
    }
    cout << endl;
}

void DFS(int vertex) {
    visited[vertex] = true;
    cout << vertex << " ";

    for (int i = 0; i < n; i++) {
        if (adj[vertex][i] == 1 && !visited[i]) {
            DFS(i);
        }
    }
}

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

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            adj[i][j] = 0;

    cout << "Enter number of edges: ";
    cin >> e;

    cout << "Enter edges (u v):\n";
    for (int i = 0; i < e; i++) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adj[v][u] = 1;
    }
    int start;
    cout << "Enter starting vertex for traversal: ";
    cin >> start;

    BFS(start);

    for (int i = 0; i < n; i++)
        visited[i] = false;

    cout << "DFS Traversal: ";
    DFS(start);
    cout << endl;

    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 6
Enter number of edges: 8
Enter edges (u v):
0 1
1 2
2 3
3 4
4 0
4 5
5 1
5 2
Enter starting vertex for traversal: 1
BFS Traversal: 1 0 2 5 4 3
DFS Traversal: 1 0 4 3 2 5


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

In this program, we successfully represented a graph using an Adjacency Matrix and implemented two fundamental graph traversal algorithms — Breadth First Search (BFS) and Depth First Search (DFS).Through BFS, we explored nodes level by level using a queue, ensuring that all neighboring vertices are visited before moving deeper. In contrast, DFS followed a depth-first approach using recursion, exploring as far as possible along each branch before backtracking.This implementation demonstrates how adjacency matrices provide an efficient way to store and access graph connections, while BFS and DFS help in understanding the structure, connectivity, and traversal order of a graph —

concepts essential in network analysis, pathfinding, and many real-world applications of graph theory.

Assignment No - 9

PROBLEM STATEMENT 2:

“Write a Program to implement Prim’s algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a graph. ”

OBJECTIVES:

To represent a weighted graph using an Adjacency List. To understand and implement Prim’s Algorithm for finding a Minimum Spanning Tree (MST). To learn how to select minimum-weight edges to connect all vertices without forming cycles. To find the minimum cost required to connect all vertices of a connected, undirected, weighted graph

THEORY:

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, undirected, weighted graph that connects all vertices together without any cycles and with the minimum possible total edge weight.

Prim’s Algorithm is a greedy algorithm used to find the MST. It starts from any vertex and repeatedly adds the smallest edge that connects a vertex in the MST to a vertex outside it, until all vertices are included.

Key Points:

It uses a priority queue (min-heap) to efficiently select the smallest edge.

It works well for dense graphs.

Time complexity: O(E log V) using a priority queue.

ALGORITHM:

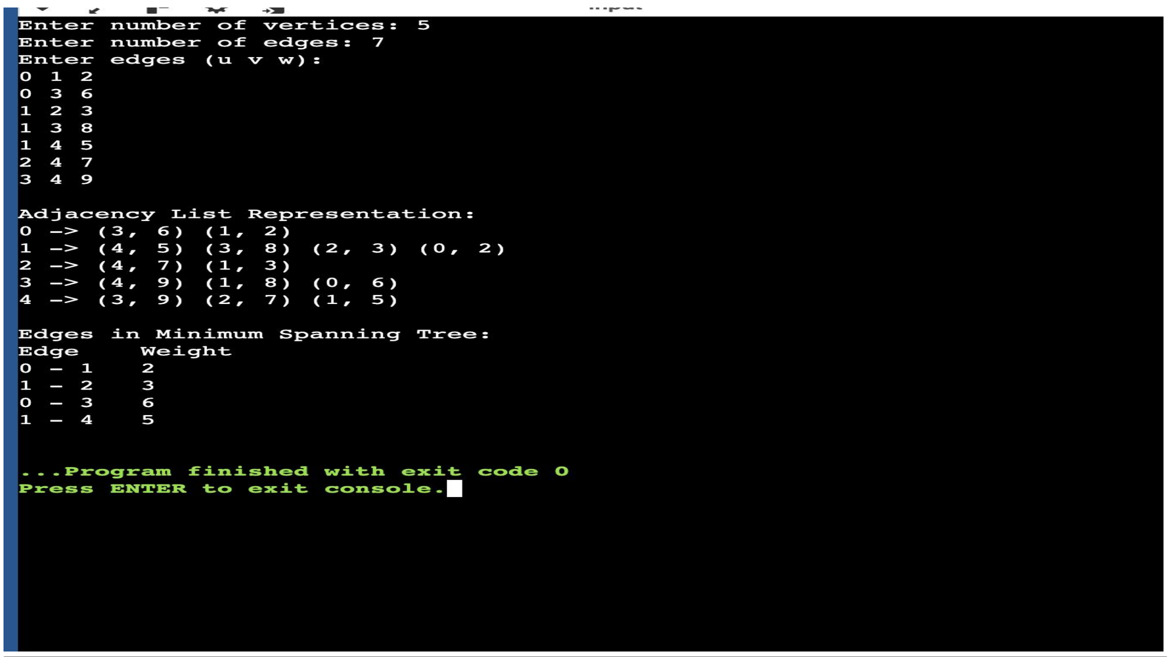
```
procedure prims_algorithm()
   input v
    input e

   create adjacency_list of size v
   for i ← 1 to e do
      input u, w, weight
      add (w, weight) to adjacency_list[u]
      add (u, weight) to adjacency_list[w]
   end for
```

```
    create array key[v] initialized to ∞
    create array parent[v] initialized to -1
    create array inmst[v] initialized to false

    create a min_priority_queue pq
    key[0] ← 0
    insert (0, 0) into pq
    while pq is not empty do
        u ← vertex with minimum key value from pq
        remove u from pq

        if inmst[u] = true then
            continue
        end if

        inmst[u] ← true

        for each (v, weight) in adjacency_list[u] do
            if inmst[v] = false and weight < key[v] then
                key[v] ← weight
                parent[v] ← u
                insert (key[v], v) into pq
            end if
        end for
    end while

    print "edges in minimum spanning tree:"
    total_cost ← 0
    for i ← 1 to v - 1 do
        print parent[i], "-", i, "weight:", key[i]
        total_cost ← total_cost + key[i]
    end for

    print "total cost of mst =", total_cost
end procedure
```

CODE:

```
#include <iostream>
using namespace std;

class Node {
public:
    int vertex;
    int weight;
    Node* next;

    Node(int v, int w) {
        vertex = v;
        weight = w;
        next = NULL;
    }
};

class Graph {
    int n;
    Node* adj[100];

public:
    Graph(int vertices) {
        n = vertices;
        for (int i = 0; i < n; i++) {
            adj[i] = NULL;
        }
    }

    void addEdge(int u, int v, int w) {
        Node* newNode1 = new Node(v, w);
        newNode1->next = adj[u];
        adj[u] = newNode1;

        Node* newNode2 = new Node(u, w);
        newNode2->next = adj[v];
        adj[v] = newNode2;
    }

    void displayGraph() {
        cout << "\nAdjacency List Representation:\n";
        for (int i = 0; i < n; i++) {
            cout << i << " -> ";
            Node* temp = adj[i];
            while (temp != NULL) {
                cout << "(" << temp->vertex << ", " << temp->weight << ") ";
                temp = temp->next;
            }
            cout << endl;
        }
    }

    int minKey(int key[], bool mstSet[]) {
        int minValue = 999999;
        int minIndex = -1;
        for (int i = 0; i < n; i++) {
            if (!mstSet[i] && key[i] < minValue) {
                minValue = key[i];
                minIndex = i;
            }
        }
        return minIndex;
    }

    void primMST() {
        int parent[100];
        int key[100];
        bool mstSet[100];

        for (int i = 0; i < n; i++) {
            key[i] = 999999;
            mstSet[i] = false;
        }

        key[0] = 0;
        parent[0] = -1;

        for (int count = 0; count < n - 1; count++) {
            int u = minKey(key, mstSet);
            mstSet[u] = true;

            Node* temp = adj[u];
            while (temp != NULL) {
                int v = temp->vertex;
                int w = temp->weight;
                if (!mstSet[v] && w < key[v]) {
                    parent[v] = u;
                    key[v] = w;
                }
                temp = temp->next;
            }
        }

        cout << "\nEdges in Minimum Spanning Tree:\n";
        cout << "Edge\tWeight\n";
        for (int i = 1; i < n; i++) {
            cout << parent[i] << " - " << i << "\t" << key[i] << endl;
        }
    }
};

int main() {
    int numVertices, numEdges;
    cout << "Enter number of vertices: ";
    cin >> numVertices;

    Graph g(numVertices);

    cout << "Enter number of edges: ";
    cin >> numEdges;
    cout << "Enter edges (u v w):\n";
    for (int i = 0; i < numEdges; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.addEdge(u, v, w);
    }

    g.displayGraph();
    g.primMST();

    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 5
Enter number of edges: 7
Enter edges (u v w):
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9

Adjacency List Representation:
0 -> (3, 6) (1, 2)
1 -> (4, 5) (3, 8) (2, 3) (0, 2)
2 -> (4, 7) (1, 3)
3 -> (4, 9) (1, 8) (0, 6)
4 -> (3, 9) (2, 7) (1, 5)

Edges in Minimum Spanning Tree:
Edge     Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5

...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

In this program, we implemented prim’s algorithm using an object-oriented approach (class) and adjacency list representation.
The program finds the minimum spanning tree (mst) by iteratively adding the smallest possible edge connecting visited and unvisited vertices.
Without using predefined stl structures, it clearly demonstrates the working of the greedy method and basic pointer-based graph construction.
The output displays all mst edges and the total minimum cost required to connect all vertices of the graph.

Assignment No - 9

PROBLEM STATEMENT 3:

“Write a Program to implement Kruskal’s algorithm to find the minimum spanning tree of a user defined graph. Use Adjacency List to represent a graph. ”

OBJECTIVES:

To represent a weighted graph using an adjacency list. To implement kruskal’s algorithm for finding the minimum spanning tree (mst). To use the disjoint set union–find method for detecting cycles. To understand the greedy approach for selecting edges in increasing order of weight.

THEORY:

Kruskal’s algorithm is a greedy algorithm used to find the minimum spanning tree (mst) of a connected, undirected, weighted graph.

The algorithm sorts all the edges in increasing order of their weights and adds them one by one to the mst, ensuring that no cycle is formed.
To detect cycles, it uses a disjoint set union (dsu) or union–find data structure.

Steps:

Sort all edges in increasing order of their weight. Pick the smallest edge. If it does not form a cycle with the mst formed so far, include it. Repeat until the mst has (v - 1) edges.

Time complexity:

Sorting edges: O(e log e)

Union-find operations: O(e log v)

ALGORITHM:

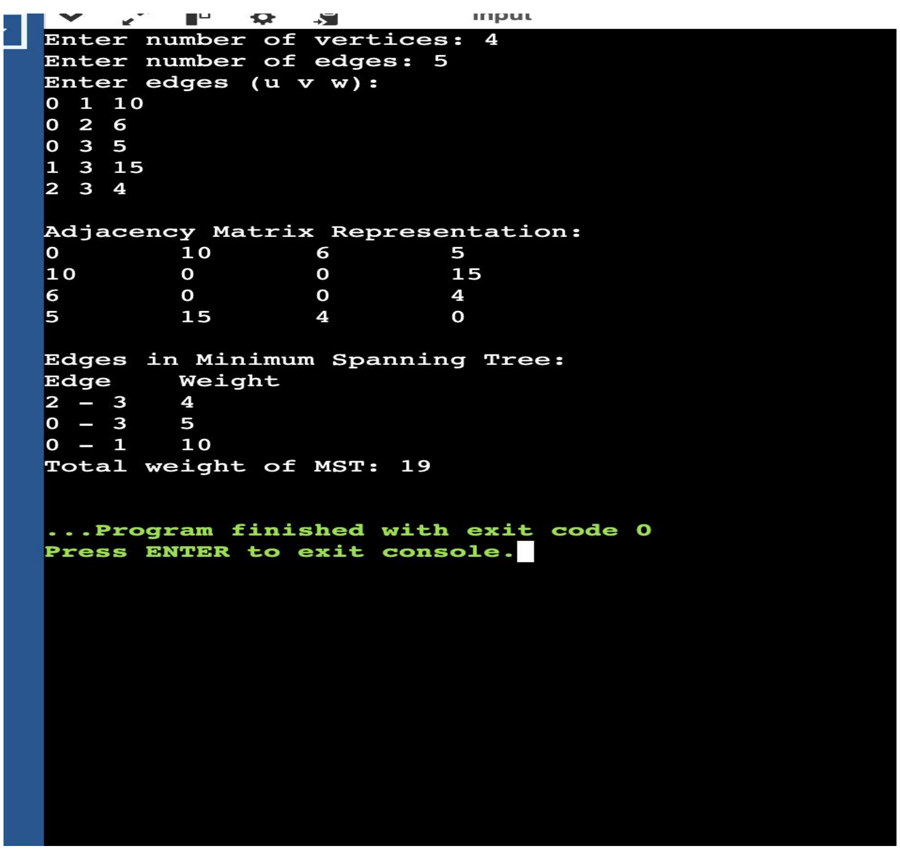
```
procedure kruskal_mst()
   input n            // number of vertices
   input e            // number of edges

   create edge_list[e]
   for i ← 1 to e do
      input u, v, w
      edge_list[i] ← (u, v, w)
   end for
```

```
    sort edge_list by weight ascending

    create parent[n], rank[n]
    for i ← 0 to n-1 do
        parent[i] ← i
        rank[i] ← 0
    end for

    mst_weight ← 0
    mst_edges ← empty

    for each (u, v, w) in edge_list do
        root_u ← find(parent, u)
        root_v ← find(parent, v)

        if root_u ≠ root_v then
            add (u, v, w) to mst_edges
            mst_weight ← mst_weight + w
            union(parent, rank, root_u, root_v)
        end if
    end for

    print "edges in mst:"
    for each edge in mst_edges do
        print u, "-", v, " weight:", w
    print "total cost =", mst_weight
end procedure
```

CODE:

```
#include <iostream>
using namespace std;

class Edge {
public:
    int src, dest, weight;
};

class Graph {
    int n;
    int adj[100][100];
    Edge edges[100];
    int edgeCount;

public:
    Graph(int vertices) {
        n = vertices;
        edgeCount = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                adj[i][j] = 0;
            }
        }
    }

    void addEdge(int u, int v, int w) {
        adj[u][v] = w;
        adj[v][u] = w; // Undirected graph

        edges[edgeCount].src = u;
        edges[edgeCount].dest = v;
        edges[edgeCount].weight = w;
        edgeCount++;
    }

    void displayMatrix() {
        cout << "\nAdjacency Matrix Representation:\n";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << adj[i][j] << "\t";
            }
            cout << endl;
        }
    }

    // Function to find the root of a vertex (for union-find)
    int findParent(int parent[], int i) {
        while (parent[i] != i)
            i = parent[i];
        return i;
    }

    // Function to perform union of two subsets
    void unionSets(int parent[], int x, int y) {
        int xset = findParent(parent, x);
        int yset = findParent(parent, y);
        parent[yset] = xset;
    }

    // Function to sort edges by weight (simple bubble sort)
    void sortEdges() {
        for (int i = 0; i < edgeCount - 1; i++) {
            for (int j = 0; j < edgeCount - i - 1; j++) {
                if (edges[j].weight > edges[j + 1].weight) {
                    Edge temp = edges[j];
                    edges[j] = edges[j + 1];
                    edges[j + 1] = temp;
                }
            }
        }
    }

    void kruskalMST() {
        int parent[100];
        Edge result[100]; // To store MST edges
        int e = 0;        // Count of edges in MST
        int i = 0;        // Index for sorted edges

        // Initialize each vertex as its own parent
        for (int v = 0; v < n; v++)
            parent[v] = v;

        sortEdges();

        cout << "\nEdges in Minimum Spanning Tree:\n";
        cout << "Edge\tWeight\n";

        while (e < n - 1 && i < edgeCount) {
            Edge nextEdge = edges[i++];
            int x = findParent(parent, nextEdge.src);
            int y = findParent(parent, nextEdge.dest);

            // If including this edge doesn’t form a cycle
            if (x != y) {
                result[e++] = nextEdge;
                unionSets(parent, x, y);
            }
        }

        // Print the MST
        int totalWeight = 0;
        for (i = 0; i < e; i++) {
            cout << result[i].src << " - " << result[i].dest << "\t" << result[i].weight << endl;
            totalWeight += result[i].weight;
        }
        cout << "Total weight of MST: " << totalWeight << endl;
    }
};

int main() {
    int numVertices, numEdges;
    cout << "Enter number of vertices: ";
    cin >> numVertices;

    Graph g(numVertices);

    cout << "Enter number of edges: ";
    cin >> numEdges;
    cout << "Enter edges (u v w):\n";
    for (int i = 0; i < numEdges; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.addEdge(u, v, w);
    }

    g.displayMatrix();
    g.kruskalMST();

    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 4
Enter number of edges: 5
Enter edges (u v w):
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4

Adjacency Matrix Representation:
0       10      6       5
10      0       0       15
6       0       0       4
5       15      4       0

Edges in Minimum Spanning Tree:
Edge    Weight
2 - 3   4
0 - 3   5
0 - 1   10
Total weight of MST: 19

...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

In this program, we implemented kruskal’s algorithm using an adjacency list and the union-find approach. The algorithm selects edges in increasing order of weight while ensuring no cycles are formed. It demonstrates the greedy technique, resulting in an mst with the minimum total cost connecting all vertices.

Assignment No - 9

PROBLEM STATEMENT 4:

"Write a program to implement Dijkshtra Algorithm to find shortest distance between two nodes of a user defined graph. Use adjacency list to represent graph. "

OBJECTIVES:

To represent a weighted graph using an adjacency list. To implement dijkstra's algorithm to find the shortest path from a source node to all other nodes. To use basic data structures (arrays, loops) instead of built-in utilities. To understand how the greedy technique minimizes the total path distance in weighted graphs.

THEORY:

Dijkstra's algorithm is a greedy algorithm that finds the shortest distance from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

The algorithm keeps track of the minimum known distance for each vertex and iteratively selects the vertex with the smallest tentative distance that has not been visited yet. It then updates the distance values for its adjacent vertices if shorter paths are found.

Time Complexity:

Using arrays → $O(V^2)$

Using priority queue (not used here) → $O((V + E) \log V)$

ALGORITHM:

```
procedure dijkstra_algorithm()
  input n                // number of vertices
  input e                // number of edges
  create adjacency_list[n]

  for i ← 1 to e do
    input u, v, w
    add (v, w) to adjacency_list[u]
    add (u, w) to adjacency_list[v]
  end for

  input source

  create array dist[n] initialized to ∞
  create array visited[n] initialized to false
```

```
    dist[source] ← 0

    for count ← 0 to n - 1 do
        u ← vertex with minimum dist[u] among unvisited vertices
        visited[u] ← true

        for each (v, w) in adjacency_list[u] do
            if visited[v] = false and dist[u] + w < dist[v] then
                dist[v] ← dist[u] + w
            end if
        end for
    end for

    print dist[] for all vertices
end procedure
```

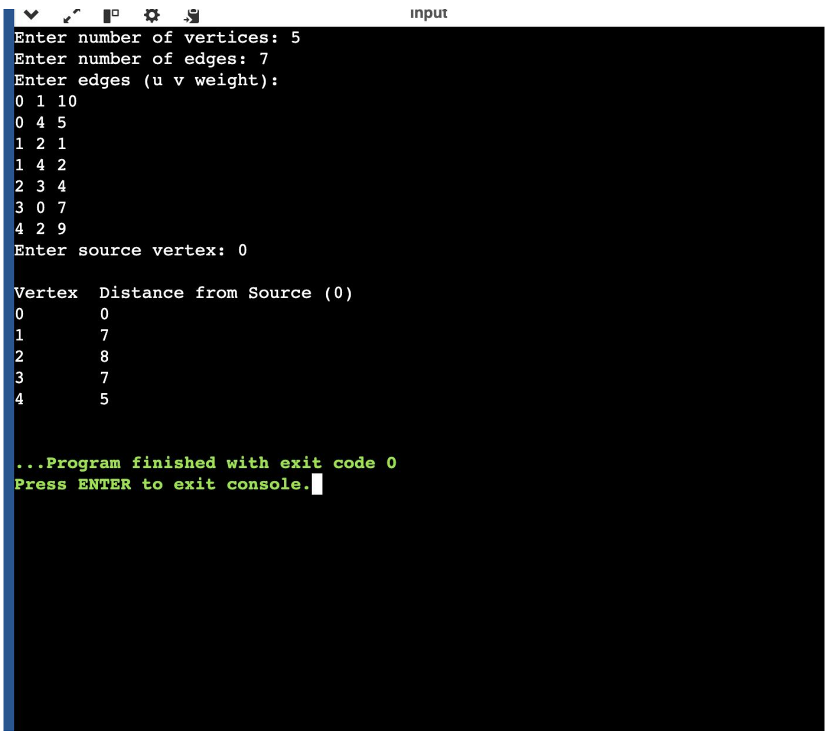
CODE:

```
#include <iostream>
using namespace std;

class Node {
public:
    int vertex;
    int weight;
    Node* next;

    Node(int v, int w) {
        vertex = v;
        weight = w;
        next = NULL;
    }
};

class Graph {
    int n;
    Node* adj[100];

public:
    Graph(int vertices) {
        n = vertices;
        for (int i = 0; i < n; i++) {
            adj[i] = NULL;
        }
    }

    void addEdge(int u, int v, int w) {
        Node* newNode1 = new Node(v, w);
        newNode1->next = adj[u];
        adj[u] = newNode1;
```

```
            Node* newNode2 = new Node(u, w);
            newNode2->next = adj[v];
            adj[v] = newNode2;
        }

        int minDistance(int dist[], bool visited[]) {
            int minVal = 999999, minIndex = -1;
            for (int i = 0; i < n; i++) {
                if (!visited[i] && dist[i] <= minVal) {
                    minVal = dist[i];
                    minIndex = i;
                }
            }
            return minIndex;
        }

        void dijkstra(int src) {
            int dist[100];
            bool visited[100];

            for (int i = 0; i < n; i++) {
                dist[i] = 999999;
                visited[i] = false;
            }

            dist[src] = 0;

            for (int count = 0; count < n - 1; count++) {
                int u = minDistance(dist, visited);
                visited[u] = true;

                Node* temp = adj[u];
                while (temp != NULL) {
                    int v = temp->vertex;
                    int w = temp->weight;
                    if (!visited[v] && dist[u] + w < dist[v]) {
                        dist[v] = dist[u] + w;
                    }
                    temp = temp->next;
                }
            }

            cout << "\nVertex\tDistance from Source (" << src << ")\n";
            for (int i = 0; i < n; i++) {
                cout << i << "\t" << dist[i] << endl;
            }
        }
    };
```

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

    Graph g(vertices);

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Enter edges (u v weight):\n";
    for (int i = 0; i < edges; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.addEdge(u, v, w);
    }

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

    g.dijkstra(src);

    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 5
Enter number of edges: 7
Enter edges (u v weight):
0 1 10
0 4 5
1 2 1
1 4 2
2 3 4
3 0 7
4 2 9
Enter source vertex: 0

Vertex  Distance from Source (0)
0       0
1       7
2       8
3       7
4       5


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

In this program, we implemented Dijkstra’s algorithm using an adjacency list and simple arrays. The algorithm continuously picks the unvisited vertex with the smallest distance and updates the distances of its adjacent vertices. This greedy approach efficiently finds the shortest path from the source node to all other nodes in a weighted graph with non-negative edges.

Assignment No - 9

PROBLEM STATEMENT 5:

"Write a program to accept a graph from user and represnt it with adjacency list and perform and BFS and DFS traversals on it "

OBJECTIVES:

The main objective of this program is to accept a user-defined graph and represent it using an Adjacency List. Then, we perform Breadth First Search (BFS) and Depth First Search (DFS) traversals on the graph to understand how different traversal techniques explore nodes. BFS explores nodes level by level, while DFS explores nodes by going deep into one branch before backtracking.

THEORY:

A graph is a collection of nodes (vertices) and edges that connect pairs of vertices. An Adjacency List is an efficient way to represent a graph, where each vertex has a list of adjacent vertices.

BFS (Breadth First Search):
It uses a queue data structure. BFS starts from a given node and visits all its adjacent nodes before moving to the next level.

DFS (Depth First Search):
It uses either recursion or a stack. DFS starts from a given node and explores as far as possible along each branch before backtracking.

ALGORITHM:

```
start
  input number of vertices n
  input number of edges e
  create adjacency list adj[n]

  for i = 1 to e do
    input u, v
    add v to adj[u]
    add u to adj[v]
  end for

  initialize visited[n] = false

  procedure bfs(start):
    create queue q
    mark visited[start] = true
```

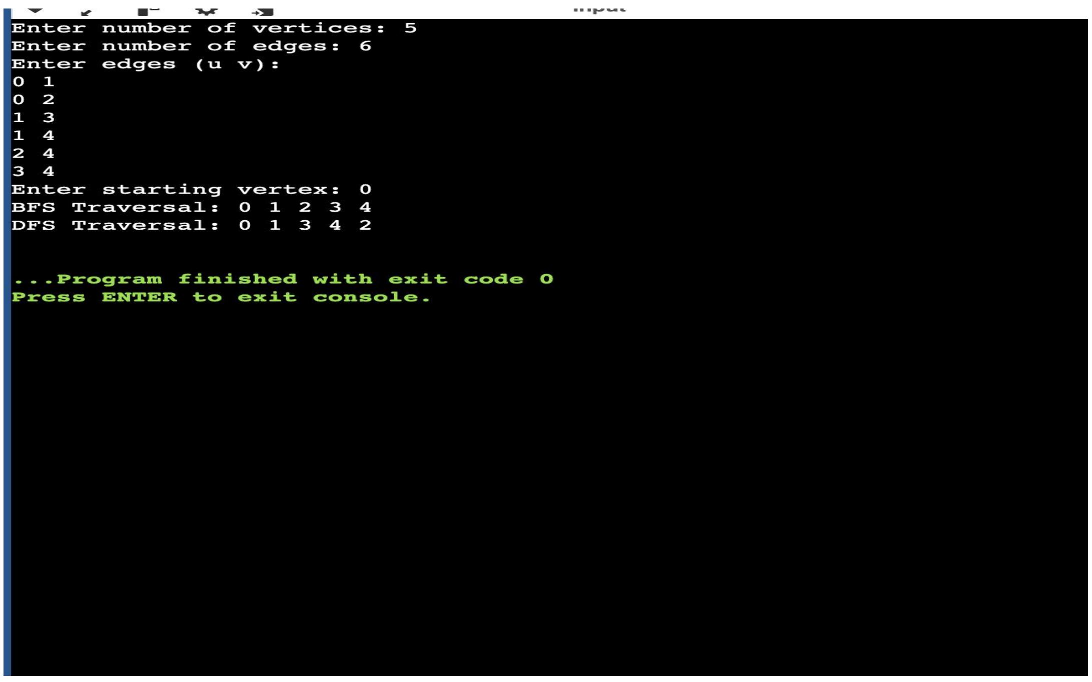
```
        enqueue(start, q)
        while q is not empty do
            node = dequeue(q)
            print node
            for each neighbor in adj[node] do
                if visited[neighbor] = false then
                    mark visited[neighbor] = true
                    enqueue(neighbor, q)
                end if
            end for
        end while
    end procedure

    procedure dfs(node):
        mark visited[node] = true
        print node
        for each neighbor in adj[node] do
            if visited[neighbor] = false then
                call dfs(neighbor)
            end if
        end for
    end procedure

    input start_vertex
    print "bfs traversal:"
    call bfs(start_vertex)

    reset visited[n] = false
    print "dfs traversal:"
    call dfs(start_vertex)
stop
```

CODE:

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;

class Graph {
    int n;
    list<int> *adj;

public:
    Graph(int vertices) {
        n = vertices;
        adj = new list<int>[n];
    }
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    void bfs(int start) {
        bool *visited = new bool[n];
        for (int i = 0; i < n; i++)
            visited[i] = false;

        queue<int> q;
        visited[start] = true;
        q.push(start);

        cout << "BFS Traversal: ";
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cout << node << " ";

            for (int neighbor : adj[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        cout << endl;
    }

    void dfsUtil(int node, bool visited[]) {
        visited[node] = true;
        cout << node << " ";
        for (int neighbor : adj[node]) {
            if (!visited[neighbor])
                dfsUtil(neighbor, visited);
        }
    }

    void dfs(int start) {
        bool *visited = new bool[n];
        for (int i = 0; i < n; i++)
            visited[i] = false;

        cout << "DFS Traversal: ";
        dfsUtil(start, visited);
        cout << endl;
    }
};
```

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

    Graph g(vertices);

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Enter edges (u v):\n";
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    int start;
    cout << "Enter starting vertex: ";
    cin >> start;

    g.bfs(start);
    g.dfs(start);

    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 5
Enter number of edges: 6
Enter edges (u v):
0 1
0 2
1 3
1 4
2 4
3 4
Enter starting vertex: 0
BFS Traversal: 0 1 2 3 4
DFS Traversal: 0 1 3 4 2


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

This program demonstrates how to represent a graph using an Adjacency List and how to traverse it using BFS and DFS.
BFS explores nodes level by level using a queue, while DFS explores depth-wise using recursion.
Through this implementation, we understand graph connectivity, traversal order, and the difference between systematic exploration (BFS) and deep exploration (DFS).