Assignment No - 10

PROBLEM STATEMENT 1:

"Write a Program to implement Kruskal's algorithm to find the minimum spanning tree of a user defined graph. Use Adjacency Matrix to represent a graph. "

OBJECTIVES:

The main objective of this program is to implement Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a graph represented using an adjacency matrix.
It aims to demonstrate how edges can be selected in increasing order of their weights without forming cycles.
Another objective is to understand how to use the union–find technique to detect cycles while building the MST.
Finally, it focuses on applying the greedy approach to ensure the resulting spanning tree has the minimum total weight.

THEORY:

Kruskal's algorithm is a greedy algorithm that finds the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph. The MST is a subset of edges that connects all vertices with the minimum total edge weight and no cycles.
The algorithm works by first sorting all edges in non-decreasing order of weight, then repeatedly adding the smallest edge to the MST that does not create a cycle.
Cycle detection is handled using the Union-Find (Disjoint Set Union) structure. Each vertex belongs to a set; when two vertices are connected, their sets are merged. If two vertices belong to the same set, adding that edge would form a cycle and is therefore skipped.
Kruskal's algorithm is especially efficient for sparse graphs, where the number of edges is much smaller than the square of the number of vertices.

ALGORITHM:

```
start
   input number of vertices n
   input adjacency matrix graph[n][n]
   initialize parent[n]
   for i = 0 to n-1
      parent[i] = i

   define function find(i)
      if parent[i] == i
         return i
      return find(parent[i])
   end function
```

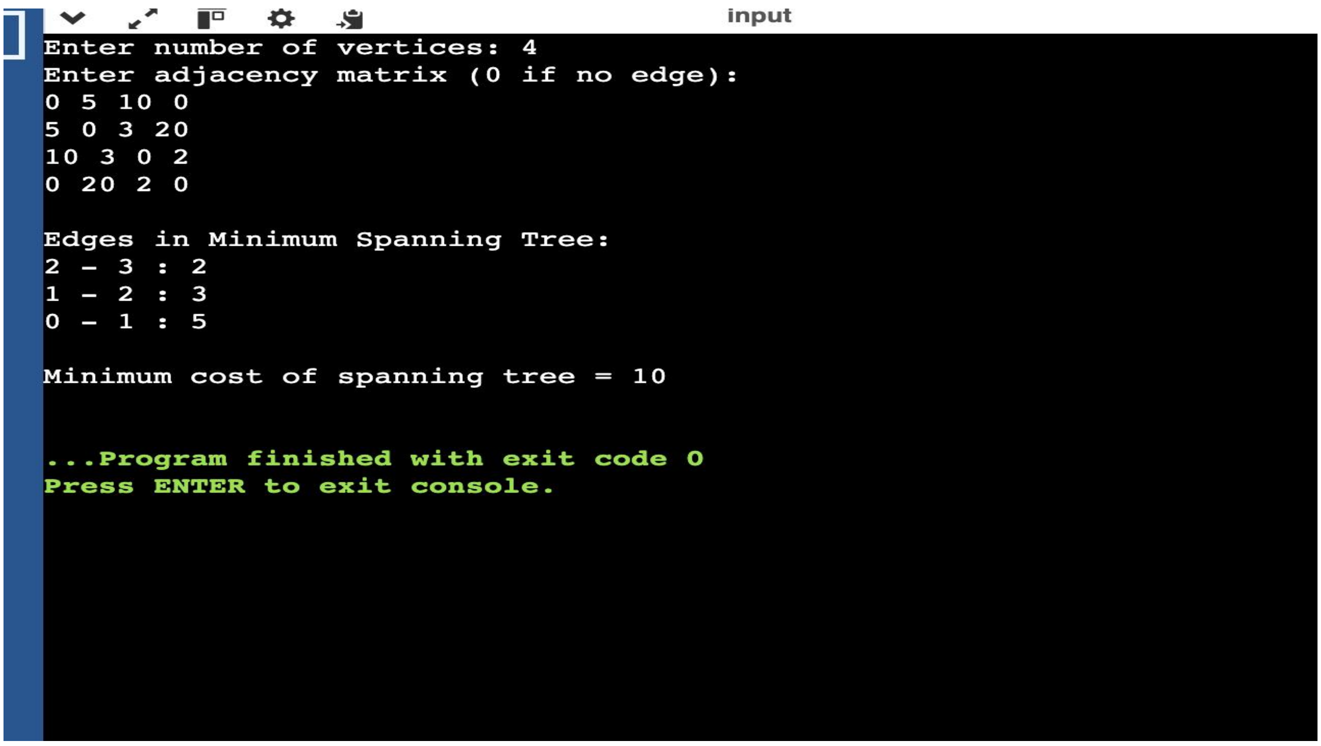
```
    define function union(u, v)
        parent[u] = v
    end function

    edge_count = 0
    mincost = 0

    while edge_count < n - 1
        min = 9999
        a = -1
        b = -1
        for i = 0 to n-1
            for j = 0 to n-1
                if find(i) != find(j) and graph[i][j] < min and graph[i][j] != 0
                    min = graph[i][j]
                    a = i
                    b = j
        u = find(a)
        v = find(b)
        if u != v
            union(u, v)
            print "edge:", a, "-", b, "cost:", min
            edge_count = edge_count + 1
            mincost = mincost + min
        graph[a][b] = graph[b][a] = 9999
    end while

    print "minimum cost of spanning tree:", mincost
stop
```

CODE:

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

class KruskalMST {
    int n;
    int graph[20][20];
    int parent[20];

public:
    void inputGraph() {
        cout << "Enter number of vertices: ";
        cin >> n;
        cout << "Enter adjacency matrix (0 if no edge):\n";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cin >> graph[i][j];
                if (graph[i][j] == 0)
                    graph[i][j] = 9999; // no edge
            }
        }
    }

    int findParent(int i) {
        if (parent[i] == i)
            return i;
        return findParent(parent[i]);
    }

    void unionNodes(int u, int v) {
        parent[u] = v;
    }

    void kruskalAlgorithm() {
        for (int i = 0; i < n; i++)
            parent[i] = i;

        int edgeCount = 0, mincost = 0;

        cout << "\nEdges in Minimum Spanning Tree:\n";
        while (edgeCount < n - 1) {
            int min = 9999, a = -1, b = -1;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (findParent(i) != findParent(j) && graph[i][j] < min) {
                        min = graph[i][j];
                        a = i;
                        b = j;
                    }
                }
            }

            int u = findParent(a);
            int v = findParent(b);

            if (u != v) {
                unionNodes(u, v);
                cout << a << " - " << b << " : " << min << endl;
                edgeCount++;
                mincost += min;
            }

            graph[a][b] = graph[b][a] = 9999;
        }

        cout << "\nMinimum cost of spanning tree = " << mincost << endl;
    }
```

```
};

int main() {
   KruskalMST k;
   k.inputGraph();
   k.kruskalAlgorithm();
   return 0;
}
```

OUTPUT:



```
Enter number of vertices: 4
Enter adjacency matrix (0 if no edge):
0 5 10 0
5 0 3 20
10 3 0 2
0 20 2 0

Edges in Minimum Spanning Tree:
2 - 3 : 2
1 - 2 : 3
0 - 1 : 5

Minimum cost of spanning tree = 10


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

CONCLUSION:

Kruskal's algorithm efficiently constructs a Minimum Spanning Tree by repeatedly adding the smallest available edge that doesn't create a cycle. Using the adjacency matrix representation, it becomes easier to traverse all possible edges. The program demonstrates the greedy nature of Kruskal's approach and successfully finds the MST with minimum total edge weight, ensuring all vertices are connected.

Assignment No - 10

PROBLEM STATEMENT 2:

"Write a Program to implement Dijkstra's algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency Matrix to represent a graph. "

OBJECTIVES:

To represent a weighted graph using an adjacency matrix. To implement Dijkstra's algorithm for finding the shortest path. To calculate the shortest distance from a source node to all other vertices. To understand the concept of greedy algorithms in shortest path finding

THEORY:

Dijkstra's algorithm, developed by Edsger W. Dijkstra in 1956, is one of the most widely used algorithms for finding the shortest path between nodes in a graph with non-negative edge weights.
It is based on a greedy approach, meaning that at each step, the algorithm selects the vertex with the minimum tentative distance that has not yet been processed.
This vertex is then considered "visited", and its neighbors are updated if a shorter path through it is found.

When using an adjacency matrix, the graph is represented as a 2D array where the cell (i, j) holds the weight of the edge between vertex i and vertex j. If no edge exists, the cell holds a very large value (infinity).
This matrix representation allows constant-time edge weight lookups but may consume more memory for sparse graphs.

The algorithm maintains a distance array, where dist[i] stores the current shortest known distance from the source to vertex i. Initially, all distances are infinite except for the source, which is set to 0. The algorithm repeatedly selects the vertex with the smallest distance value and updates the distances of its adjacent vertices.

Dijkstra's algorithm is widely used in network routing protocols, Google Maps, telecommunication systems, and graph-based AI because it efficiently computes the most optimal route or communication path in complex networks.

ALGORITHM:

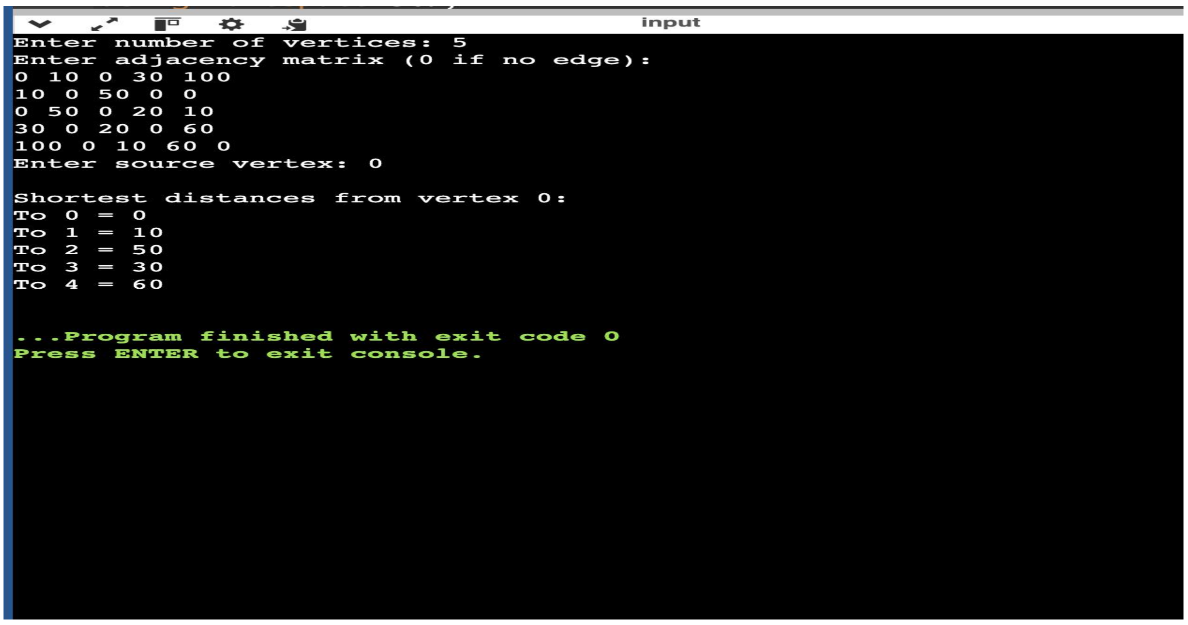
```
start
   input number of vertices n
   input adjacency matrix graph[n][n]
```

```
    input source vertex src
    initialize dist[i] = infinity for all i
    initialize visited[i] = false
    dist[src] = 0
    for count = 0 to n-1
        u = vertex with minimum dist and not visited
        visited[u] = true
        for v = 0 to n-1
            if not visited[v] and graph[u][v] != 0 and dist[u] + graph[u][v] < dist[v]
                dist[v] = dist[u] + graph[u][v]
    print shortest distance from src to all vertices
stop
```

CODE:

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

class Dijkstra {
    int n, graph[20][20];
public:
    void inputGraph() {
        cout << "Enter number of vertices: ";
        cin >> n;
        cout << "Enter adjacency matrix (0 if no edge):\n";
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                cin >> graph[i][j];
    }

    void dijkstra(int src) {
        int dist[20], visited[20];
        for (int i = 0; i < n; i++) {
            dist[i] = 9999;
            visited[i] = 0;
        }
        dist[src] = 0;

        for (int count = 0; count < n - 1; count++) {
            int u = -1, min = 9999;
            for (int i = 0; i < n; i++)
                if (!visited[i] && dist[i] < min) {
                    min = dist[i];
                    u = i;
                }
            visited[u] = 1;
            for (int v = 0; v < n; v++)
                if (!visited[v] && graph[u][v] && dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        cout << "\nShortest distances from vertex " << src << ":\n";
        for (int i = 0; i < n; i++)
            cout << "To " << i << " = " << dist[i] << endl;
    }
};

int main() {
    Dijkstra d;
    d.inputGraph();
    int src;
    cout << "Enter source vertex: ";
    cin >> src;
    d.dijkstra(src);
    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 5
Enter adjacency matrix (0 if no edge):
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
Enter source vertex: 0

Shortest distances from vertex 0:
To 0 = 0
To 1 = 10
To 2 = 50
To 3 = 30
To 4 = 60


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

CONCLUSION:

Dijkstra's algorithm efficiently finds the shortest path from a source to all vertices using a greedy approach. The adjacency matrix allows direct edge access, making the computation straightforward.

Assignment No - 10

PROBLEM STATEMENT 3:

"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 user-defined weighted graph using an adjacency list.
To implement Prim's algorithm for finding the Minimum Spanning Tree (MST).
To learn how to choose minimum edge weights step-by-step from connected vertices.
To understand how greedy algorithms minimize total connection cost.

THEORY:

Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected and undirected graph.
The MST is a subset of edges that connects all vertices with the minimum possible total weight and contains no cycles.

The algorithm starts from any arbitrary vertex and grows the MST one vertex at a time. At each step, it chooses the smallest weight edge that connects a vertex inside the MST to a vertex outside it.
This ensures that at every step, the tree remains connected and minimal in total weight.

In this program, the graph is represented using an adjacency list, where each vertex maintains a linked list of its adjacent vertices and edge weights. This structure is memory-efficient, especially for sparse graphs, where not all pairs of vertices are connected.

Prim's algorithm uses a key array to store the minimum weight edge connecting a vertex to the MST and a boolean array to track which vertices are already included.
The algorithm continues until all vertices are included in the MST.

Prim's algorithm is commonly used in network design problems like computer networks, electrical grid connections, and road network construction, where cost optimization is crucial.

ALGORITHM:

```
#include <iostream>
start
   input number of vertices n
   create adjacency list
```

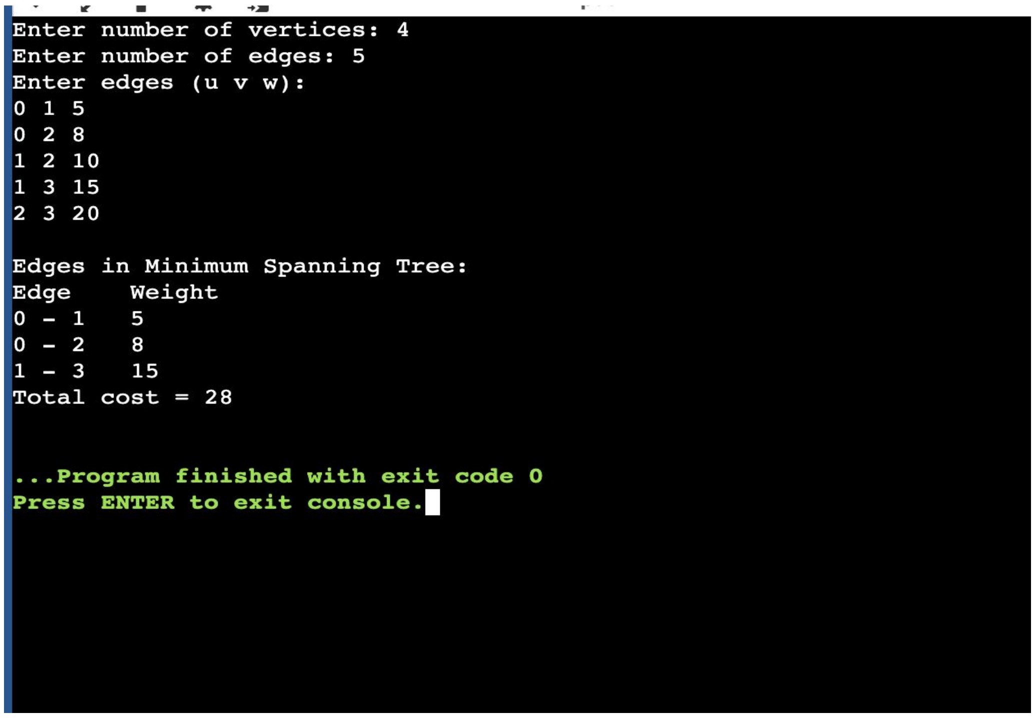
```
    for each edge (u, v, w)
        add (v, w) to list of u
        add (u, w) to list of v
    initialize key[n] = infinity
    initialize mstSet[n] = false
    key[0] = 0
    parent[0] = -1

    repeat n-1 times
        find vertex u with smallest key not in mstSet
        set mstSet[u] = true
        for each adjacent vertex v of u
            if mstSet[v] == false and weight(u,v) < key[v]
                parent[v] = u
                key[v] = weight(u,v)
    print edges and total cost
stop
```

CODE:

```cpp
#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 minKey(int key[], bool mstSet[]) {
        int minValue = 999999, 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";
        int total = 0;
        for (int i = 1; i < n; i++) {
            cout << parent[i] << " - " << i << "\t" << key[i] << endl;
```

```
            total += key[i];
        }
        cout << "Total cost = " << total << 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.primMST();
    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 4
Enter number of edges: 5
Enter edges (u v w):
0 1 5
0 2 8
1 2 10
1 3 15
2 3 20

Edges in Minimum Spanning Tree:
Edge    Weight
0 - 1   5
0 - 2   8
1 - 3   15
Total cost = 28


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

CONCLUSION:

Prim’s algorithm efficiently constructs a Minimum Spanning Tree by expanding from one vertex and always choosing the smallest adjacent edge. The adjacency list representation ensures efficient memory usage and easy traversal for sparse graphs.

Assignment No - 10

PROBLEM STATEMENT 4:

"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 graph using an adjacency list. To implement Kruskal's algorithm for MST creation. To learn edge-based sorting and union-find operations. To compute the minimum cost of connecting all nodes.

THEORY:

Kruskal's algorithm is a greedy algorithm that finds the Minimum Spanning Tree (MST) by sorting all the edges of a graph in increasing order of their weights and then adding them one by one to the MST, provided they do not form a cycle.
Unlike Prim's algorithm, which grows a tree vertex by vertex, Kruskal's algorithm grows the MST edge by edge.

When using an adjacency list, the graph is stored as a list of connected edges for each vertex, along with their weights. This representation is efficient in terms of memory and helps in quickly traversing or collecting all edges.

The algorithm uses the Disjoint Set Union (DSU) or Union-Find data structure to keep track of which vertices belong to which components. Before adding an edge, the algorithm checks if the two endpoints of the edge belong to the same component.
If they do, adding the edge would form a cycle, so it's skipped. If they belong to different components, the edge is added to the MST, and their sets are united.

Kruskal's algorithm is highly efficient for sparse graphs and is commonly used in telecommunications, electrical circuit design, network layout optimization, and cluster analysis.

ALGORITHM:

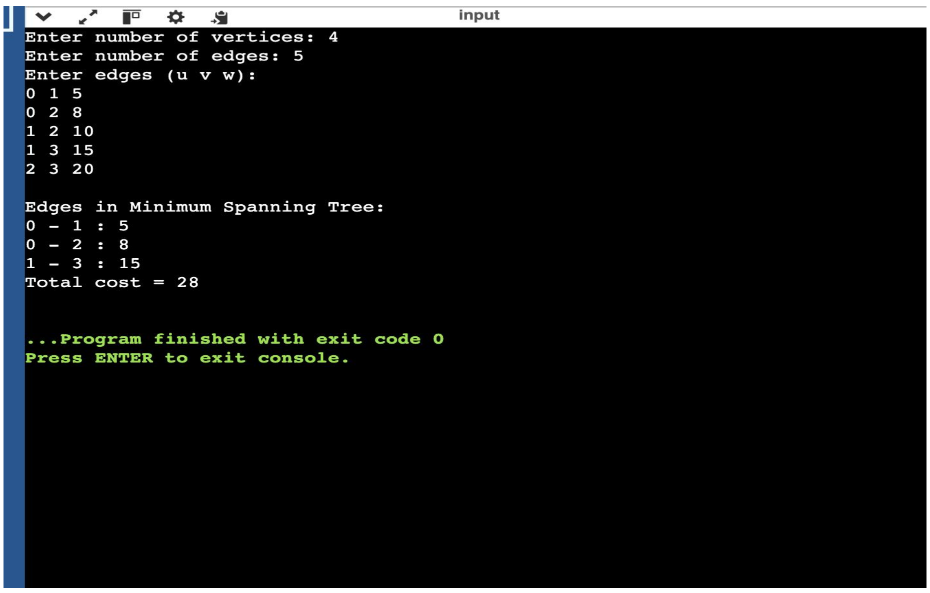
```
#include <iostream>
start
   input vertices and edges
   create edge list with weights
   sort edges by ascending weight
   for each edge (u, v)
      find parent of u and v
```

```
        if parent different
            include edge
            union parent sets
    print mst edges and total cost
stop
```

CODE:

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

class Edge {
public:
    int u, v, w;
};

class KruskalList {
    int n, e;
    Edge edges[100];
    int parent[100];

public:
    void inputGraph() {
        cout << "Enter number of vertices: ";
        cin >> n;
        cout << "Enter number of edges: ";
        cin >> e;
        cout << "Enter edges (u v w):\n";
        for (int i = 0; i < e; i++)
            cin >> edges[i].u >> edges[i].v >> edges[i].w;
    }

    int findParent(int i) {
        if (parent[i] == i)
            return i;
        return findParent(parent[i]);
    }

    void unionNodes(int u, int v) {
        parent[u] = v;
    }

    void sortEdges() {
        for (int i = 0; i < e - 1; i++)
            for (int j = 0; j < e - i - 1; j++)
                if (edges[j].w > edges[j + 1].w)
                    swap(edges[j], edges[j + 1]);
    }
    void kruskalAlgorithm() {
        for (int i = 0; i < n; i++)
            parent[i] = i;

        sortEdges();

        int total = 0, count = 0;

        cout << "\nEdges in Minimum Spanning Tree:\n";
        for (int i = 0; i < e && count < n - 1; i++) {
            int u = findParent(edges[i].u);
            int v = findParent(edges[i].v);
            if (u != v) {
                unionNodes(u, v);
                cout << edges[i].u << " - " << edges[i].v << " : " << edges[i].w << endl;
                total += edges[i].w;
                count++;
            }
        }
        cout << "Total cost = " << total << endl;
    }
};

int main() {
    KruskalList k;
    k.inputGraph();
    k.kruskalAlgorithm();
    return 0;
}
```

OUTPUT:



```
Enter number of vertices: 4
Enter number of edges: 5
Enter edges (u v w):
0 1 5
0 2 8
1 2 10
1 3 15
2 3 20

Edges in Minimum Spanning Tree:
0 - 1 : 5
0 - 2 : 8
1 - 3 : 15
Total cost = 28


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

CONCLUSION:

Kruskal’s algorithm builds a Minimum Spanning Tree by adding the smallest edge that doesn’t form a cycle. It ensures a globally optimal MST and works well with adjacency lists for efficient memory use.

Assignment No - 10

PROBLEM STATEMENT 5:

"Write a Program to implement Dijkstra's algorithm to find shortest distance between two nodes of a user defined graph.Use Adjacency List to represent a graph. "

OBJECTIVES:

The objective of this program is to understand and implement Dijkstra's shortest path algorithm using an adjacency list representation.
It aims to calculate the minimum distance from a source node to all other nodes in a weighted, directed, or undirected graph.
The program also focuses on demonstrating how priority-based edge selection helps in optimizing graph traversal.
Additionally, it aims to strengthen understanding of graph representation, greedy techniques, and data structure implementation in real-world path-finding applications.

THEORY:

Dijkstra's algorithm is a greedy algorithm that finds the shortest path between nodes in a weighted graph, where all edge weights are non-negative.
It works by progressively selecting the vertex with the minimum tentative distance and updating the distances of its neighboring vertices. This process continues until the shortest path to all vertices from the source is found.

The algorithm maintains two sets of vertices:

Visited set – vertices for which the shortest distance is finalized.

Unvisited set – vertices whose distances may still be updated.

At each iteration, the algorithm picks the vertex with the minimum distance value from the unvisited set, checks all its adjacent vertices, and updates their distances if a shorter path is found through the selected vertex.

In this implementation, the graph is represented using an Adjacency List, where each vertex points to a linked list of its connected vertices along with edge weights. This structure is memory-efficient for sparse graphs compared to adjacency matrices.

Dijkstra's algorithm is widely used in GPS navigation systems, network routing protocols (like OSPF), and various AI-based pathfinding problems where the goal is to find the optimal route or minimal traversal cost.

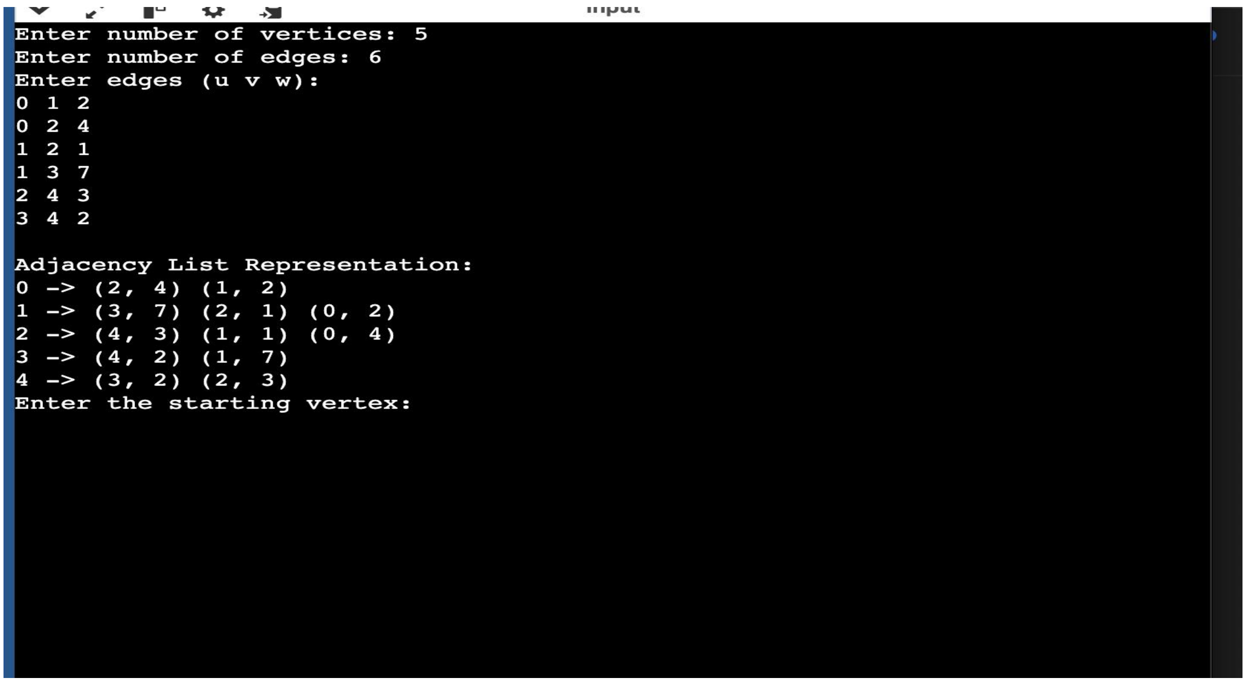
ALGORITHM:

```
#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 i = 0 to n-1
            adj[i] = null

    void add_edge(int u, int v, int w)
        node* newnode = new node(v, w)
        newnode->next = adj[u]
        adj[u] = newnode

        node* newnode2 = new node(u, w)
        newnode2->next = adj[v]
        adj[v] = newnode2

    void display()
        print "adjacency list representation:"
        for i = 0 to n-1
            print i, " -> "
            node* temp = adj[i]
            while temp != null
                print "(" + temp->vertex + ", " + temp->weight + ") "
                temp = temp->next
            print newline

    int min_distance(int dist[], bool visited[])
        int min = 999999, index = -1
        for i = 0 to n-1
            if visited[i] == false and dist[i] < min
                min = dist[i]
            index = i
        return index

    void dijkstra(int start)
        int dist[100]
        bool visited[100]

        for i = 0 to n-1
            dist[i] = 999999
            visited[i] = false

        dist[start] = 0

        for count = 0 to n-1
            u = min_distance(dist, visited)
            visited[u] = true
            node* temp = adj[u]
            while temp != null
                v = temp->vertex
                w = temp->weight
                if visited[v] == false and dist[u] + w < dist[v]
                    dist[v] = dist[u] + w
                temp = temp->next

        print "vertex \t distance from source"
        for i = 0 to n-1
            print i, "\t", dist[i]
end
```

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

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

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

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

        dist[start] = 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 " << start << endl;
        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 w):\n";
    for (int i = 0; i < edges; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.addEdge(u, v, w);
    }

    g.displayGraph();

    int start;
    cout << "Enter the starting vertex: ";
    cin >> start;
    g.dijkstra(start);

    return 0;
}
```

OUTPUT:



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

Adjacency List Representation:
0 -> (2, 4) (1, 2)
1 -> (3, 7) (2, 1) (0, 2)
2 -> (4, 3) (1, 1) (0, 4)
3 -> (4, 2) (1, 7)
4 -> (3, 2) (2, 3)
Enter the starting vertex:
```

CONCLUSION:

Dijkstra’s algorithm efficiently finds the shortest path from a single source to all other vertices in a weighted graph with non-negative weights.
Using an adjacency list makes the implementation memory-efficient and suitable for large, sparse graphs.
This program demonstrates how greedy selection of the next minimum distance vertex leads to the optimal path solution and highlights the algorithm’s real-world importance in navigation, routing, and network optimization.