INDEX

Experiment No: 01

Experiment Name: Implementation of Breadth First Search (BFS).

Theory: Breadth-First Search (BFS) is a graph traversal algorithm that explores vertices layer by layer. It starts with a source vertex and visits all its adjacent vertices before moving to the next level. BFS is implemented using a queue and is commonly used in shortest path problems, network flow, and AI applications.

Algorithm:

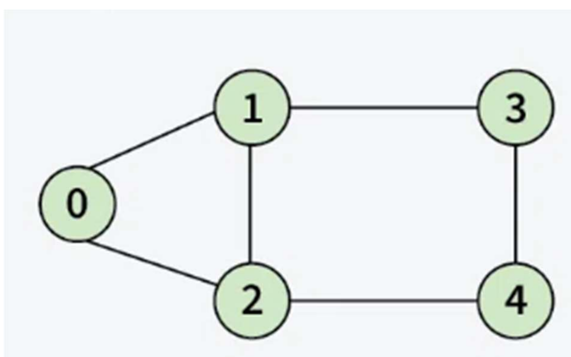
1. Initialize an empty queue and mark the source node as visited.
2. Enqueue the source node.
3. While the queue is not empty: a. Dequeue a node and process it. b. Enqueue all unvisited adjacent nodes and mark them as visited.
4. Repeat until the queue is empty.

Code:

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 100

void enqueue(int queue[], int *rear, int vertex) {
    queue[*rear] = vertex;
    (*rear)++;
}

int dequeue(int queue[], int *front) {
    int vertex = queue[*front];
    (*front)++;
    return vertex;
}

void bfs(int graph[MAX_VERTICES][MAX_VERTICES], int startVertex, int numVertices) {
    int queue[MAX_VERTICES];
    bool visited[MAX_VERTICES] = {false};
    int front = 0, rear = 0;
    visited[startVertex] = true;
    enqueue(queue, &rear, startVertex);

    while (front < rear) {
        int currentVertex = dequeue(queue, &front);
        printf("%d ", currentVertex);

        for (int i = 0; i < numVertices; i++) {
            if (graph[currentVertex][i] == 1 && !visited[i]) {
```

```c
                visited[i] = true;
                enqueue(queue, &rear, i);
            }
        }
    }
}

int main() {
    int numVertices;
    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);

    int graph[MAX_VERTICES][MAX_VERTICES];
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    int startVertex;
    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);

    printf("BFS:\t");
    bfs(graph, startVertex, numVertices);
    return 0;
}
```

Graph Picture:



Input:

```
Enter the number of vertices: 5
Enter the adjacency matrix:
0 1 1 0 0
1 0 1 1 0
1 1 0 0 1
0 1 0 0 1
0 0 1 1 0
Enter the starting vertex: 0
```

Output:

```
BFS:    0 1 2 3 4
PS D:\3rd Semester\DSA\DSA Codes>
```

Experiment No: 02

Experiment Name: Implementation of Depth First Search (DFS).

Theory: Depth-First Search (DFS) is a graph traversal algorithm that explores as deep as possible along a branch before backtracking. It uses recursion or an explicit stack and is commonly used in cycle detection, topological sorting, and pathfinding.

Algorithm:

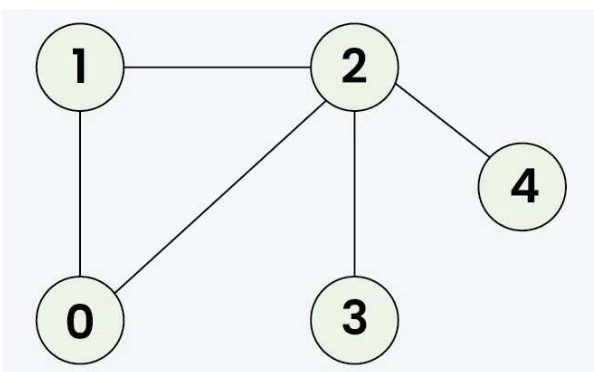
1. Initialize an empty stack and mark the source node as visited.
2. Push the source node onto the stack.
3. While the stack is not empty: a. Pop a node from the stack and process it. b. Push all unvisited adjacent nodes onto the stack and mark them as visited.
4. Repeat until the stack is empty.

Code:

```
#include <stdio.h>

void DFS(int v, int n, int G[n][n], int visited[n]);

int main() {
    int n;

    printf("Enter number of vertices: ");
    scanf("%d", &n);
    int G[n][n], visited[n];

    printf("\nEnter adjacency matrix of the graph:\n");
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            //printf("G[%d %d] : ",i,j);
            scanf("%d", &G[i][j]);
        }
    }


    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            printf("%d ",G[i][j]);
        }
        printf("\n");
    }

```

```c

    // Initialize visited array to 0 (unvisited)
    for(int i = 0; i < n; i++) {
        visited[i] = 0;
    }

    printf("Enter the starting node : ");
    int start;
    scanf("%d",&start);
    printf("DFS Traversal:\t");
    DFS(start, n, G, visited);

    return 0;
}

void DFS(int start, int n, int G[n][n], int visited[n]) {
    printf("%d ", start);
    visited[start] = 1;

    for(int j = 0; j < n; j++) {
        if(!visited[j] && G[start][j] == 1) {
            DFS(j, n, G, visited);
        }
    }
}
```

Graph:



Input:

```
Enter number of vertices: 5

Enter adjacency matrix of the graph:
0 1 1 0 0
1 0 1 0 0
0 1 0 1 1
0 0 1 0 0
0 0 1 0 0
Enter the starting node : 1
```

Output:

```
DFS Traversal:  1 0 2 3 4
PS D:\4th Semester\Algorithms 1\Codes>
```

Experiment No: 03

Experiment Name: Implementation of Strongly Connected Component (SCC).

Theory: Strongly Connected Components (SCCs) are subgraphs where every vertex is reachable from every other vertex in a directed graph. Identifying SCCs helps in network analysis, deadlock detection, and circuit design. Kosaraju’s and Tarjan’s algorithms are commonly used for this purpose.

Algorithm (Kosaraju’s Algorithm):

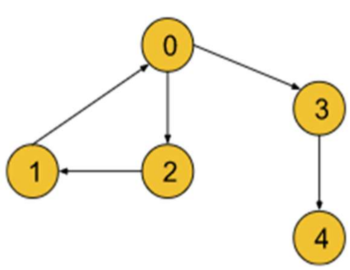
1. Perform DFS on the original graph and store the finishing order.
2. Reverse the graph (transpose it).
3. Perform DFS on the transposed graph in the order obtained in step 1.
4. Each DFS traversal results in a strongly connected component.

Code:

```c
#include <stdio.h>
int top = -1;
void dfs1(int n, int graph[n][n], int visited[n], int stack[n], int v) {
    visited[v] = 1;
    for (int i = 0; i < n; i++) {
        if (!visited[i] && graph[v][i] == 1) {
            dfs1(n, graph, visited, stack, i);
        }
    }
    stack[++top] = v;  // Store the vertex in finish order after exploring it
}

void dfs2(int n, int transpose[n][n], int visited[n], int v) {
    visited[v] = 1;
    printf("%d ", v);
    for (int i = 0; i < n; i++) {
        if (!visited[i] && transpose[v][i] == 1) {
            dfs2(n, transpose, visited, i);
        }
    }
}

void findSCC(int n, int graph[n][n]) {
    int visited[n];
    int stack[n];
    int transpose[n][n];
    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }
    for (int i = 0; i < n; i++) {
```

```
        if (visited[i]==0) {
            dfs1(n, graph, visited, stack, i);
        }
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            transpose[i][j] = graph[j][i];
        }
    }
    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }

    while(top!=-1){
        int v = stack[top--];
        if (visited[v]==0) {
            printf("\nSCC: ");
            dfs2(n, transpose, visited, v);
        }
    }
}
int main() {
    int n;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    int graph[n][n];
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    findSCC(n, graph);
    return 0;}
```

Graph:



Input:

```
PS C:\Users\new> cd "d:\4th Semester\Algor
ected } ; if ($?) { .\Strongly_Connected }
Enter the number of vertices: 5
Enter the adjacency matrix:
0 0 1 1 0
1 0 0 0 0
0 1 0 0 0
0 0 0 0 1
0 0 0 0 0
```

Output:

```
SCC: 0 1 2
SCC: 3
SCC: 4
PS D:\4th Semester\Algorithms 1\Codes>
```

Experiment No: 04

Experiment Name: Implementation of Finding Articulation Points**.**

Theory: Articulation points are critical nodes in an undirected graph whose removal increases the number of connected components. They play an important role in network reliability analysis and vulnerability detection.

Algorithm:

1. Perform DFS and assign discovery and low values to nodes.
2. If the root has more than one child, it is an articulation point.
3. For other nodes, check if any adjacent node has a low value greater than or equal to the discovery value.
4. If yes, mark it as an articulation point.

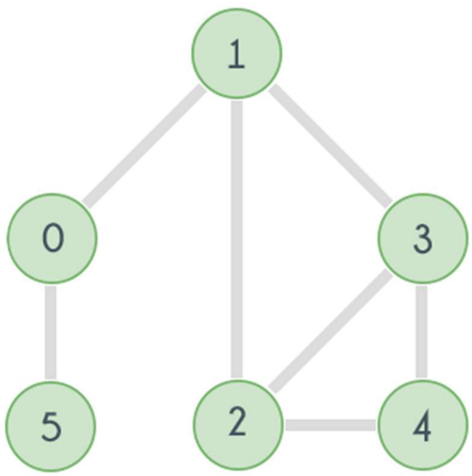
Code:

```
#include <stdio.h>
#define MAX 100
int visited[MAX];
int parent[MAX];
int low[MAX];
int disc[MAX];
int time = 0;
int ap[MAX];
int adj[MAX][MAX];

int min(int a, int b) {
    return a < b ? a : b;
}

void DFS(int V, int u) {
    visited[u] = 1;
    disc[u] = low[u] = ++time;

    int children = 0;

    for (int v = 0; v < V; v++) {
        if (adj[u][v] == 1 && visited[v]==0) {
            children++;
            parent[v] = u;
            DFS(V, v);

            low[u] = min(low[u], low[v]);

            // Articulation point conditions:
            if (parent[u] == -1 && children > 1) {
```

```
                ap[u] = 1;
            } else if (parent[u] != -1 && low[v] >= disc[u]) {
                ap[u] = 1;
            }
        } else if (adj[u][v] == 1 && v != parent[u]) {
            low[u] = min(low[u], disc[v]);
        }
    }
}

int find_articulation_points(int V) {
    for (int i = 0; i < V; i++) {
        parent[i] = -1;
        visited[i] = 0;
        ap[i] = 0;
    }

    for (int i = 0; i < V; i++) {
        if (visited[i]==0) {
            DFS(V, i);
        }
    }

    int count = 0;
    for (int i = 0; i < V; i++) {
        if (ap[i]==1) {
            count++;
        }
    }

    return count;
}
int main() {
    int V, E;
    printf("Enter the number of Vertices and Edges : ");
    // Read the number of vertices and edges
    scanf("%d %d", &V, &E);

    // Initialize the adjacency matrix
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            adj[i][j] = 0;
        }
    }

    printf("Enter the Adjacent pairs : \n");
    for (int i = 0; i < E; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        adj[u][v] = adj[v][u] = 1;
    }

    int count = find_articulation_points(V);
    printf("Number of articulation points: %d\n", count);
```

```c
    printf("Articulation points: ");
    for (int i = 0; i < V; i++) {
        if (ap[i]==1) {
            printf("%d ", i);
        }
    }
    printf("\n");
    return 0;
}.
```

Graph:



Input:

```
cd "d:\4th Semester\Algorithms 1\Codes\" ; if ($?) { g
 { .\tempCodeRunnerFile }
Enter the number of Vertices and Edges : 6 7
Enter the Adjacent pairs :
0 1
0 5
1 3
1 2
2 4
2 3
3 4
```

Output:

```
Number of articulation points: 2
Articulation points: 0 1
PS D:\4th Semester\Algorithms 1\Codes>
```

Experiment No: 05

Experiment Name: Implementation of Dijkstra's Algorithm.

Theory: Dijkstra's algorithm finds the shortest path from a single source vertex to all other vertices in a weighted graph. It uses a greedy approach and a priority queue, making routing and navigation applications efficient.

Algorithm**:**

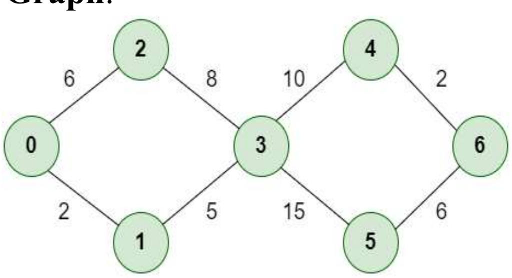
1. Initialize distances to all nodes as infinity except the source.
2. Use a priority queue to select the node with the smallest distance.
3. Update distances of adjacent nodes if a shorter path is found.
4. Repeat until all nodes are processed.

Code:

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

int min(int n, int dist[n], int visited[n]) {
    int min = INT_MAX, minIndex;
    for (int i = 0; i < n; i++) {
        if (!visited[i] && dist[i] <= min) {
            min = dist[i];
            minIndex = i;
        }
    }
    return minIndex;
}

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

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

        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] != INT_MAX && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
```

```c
            }
        }
    }
    printf("Vertex \t Distance from Source\n");
    int sum = 0;
    for (int i = 0; i < n; i++) {
        if (dist[i] == INT_MAX) {
            printf("%d \t\t INF\n", i);
        } else {
            printf("%d \t\t %d\n", i, dist[i]);
            sum += dist[i];
        }
    }
    printf("\nTotal path cost: %d\n", sum);
}
int main() {
    int n;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    int graph[n][n];
    printf("Enter the adjacency matrix (use 0 for no edge):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
            if (i != j && graph[i][j] == 0) {
                graph[i][j] = INT_MAX;
            }
        }
    }
    int src;
    printf("Enter the source vertex (0 to %d): ", n - 1);
    scanf("%d", &src);
    if (src < 0 || src >= n) {
        printf("Invalid source vertex. Please restart the program.\n");
        return 1;
    }
    dijkstra(n, graph, src);
    return 0;
}
```

Graph:



Input:

```
PS C:\Users\new> cd "d:\4th Semester\Algorithms 1
) { .\Dijsktra_2 }
Enter the number of vertices: 7
Enter the adjacency matrix (use 0 for no edge):
 0  2  6  0  0  0  0
 2  0  0  5  0  0  0
 6  0  0  8  0  0  0
 0  5  8  0  10 15 0
 0  0  0 10  0  0  2
 0  0  0 15  0  0  6
 0  0  0  0  2  6  0
Enter the source vertex (0 to 6): 0
```

Output:

```
Vertex   Distance from Source
0               0
1               2
2               6
3               7
4               17
5               22
6               19

Total path cost: 73
PS D:\4th Semester\Algorithms 1\Codes>
```

Experiment No: 06

Experiment Name: Implementation of Prim’s Algorithm (Greedy Method).

Theory: Prim’s algorithm constructs a Minimum Spanning Tree (MST) by greedily selecting the smallest edge while ensuring all nodes are connected. It is widely used in network design and clustering problems.

Algorithm:

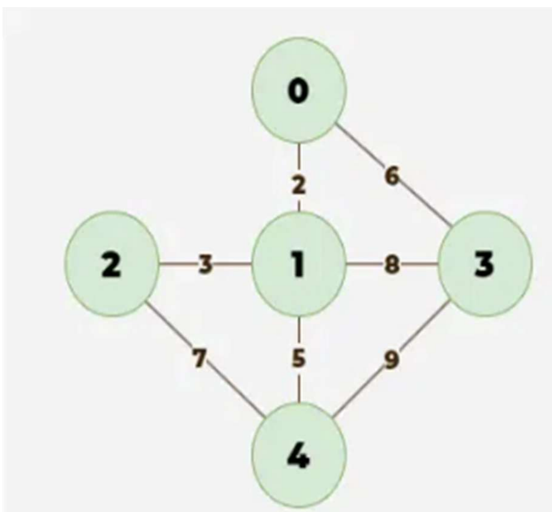
1. Start from an arbitrary node and initialize an empty MST.
2. Use a priority queue to repeatedly select the minimum weight edge.
3. Add the selected edge to the MST.
4. Repeat until all nodes are included.

Code:

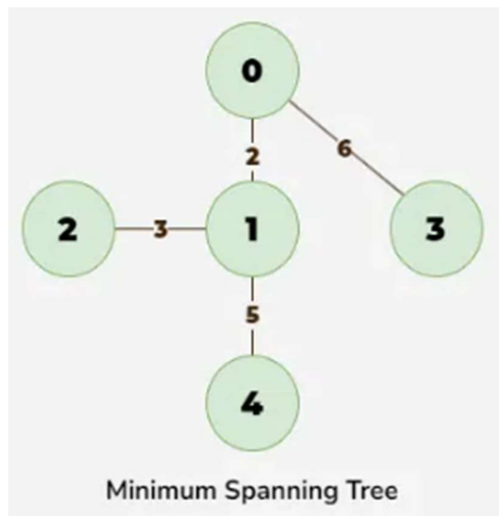
```c
#include<stdio.h>
#include<limits.h>
void main(){
    int V;
    printf("Enter Number of vertex : ");
    scanf("%d",&V);
    int G[V][V];
    printf("Enter the Adjacency Matrix : \n");
    for(int i = 0 ; i < V ; i++){
        for(int j = 0 ; j < V ; j++){
            scanf("%d",&G[i][j]);
        }
    }
    int selected[V];
    for(int i = 0 ; i < V ; i ++){
        selected[i] = 0;
    }
    selected[0] = 1;
    int E = 0;
    int x,y;
    int totalCost = 0;
    printf("Edge  : Weight \n");
    while(E < V - 1 ){
    int min = INT_MAX;
    x = 0;
    y = 0;
    for(int i = 0; i < V ; i++){
        if(selected[i]){
            for(int j = 0 ; j < V ; j++){
                if(!selected[j] && G[i][j]){
                    if(G[i][j] < min){
                        min = G[i][j];
                        x = i;
```

```
                y = j;
              }

            }
          }
        }
      }
      selected[y] = 1;
      printf("%d - %d : %d \n", x , y , G[x][y]);
      totalCost += G[x][y];
      E++;
    }
    printf("Total cost : %d", totalCost);
}
```

Graph:



MST:



Input:

```
.c -o Prims_algorithm } ; if
Enter Number of vertex : 5
Enter the Adjacency Matrix :
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
```

Output:

```
Edge   : Weight
0 - 1 : 2
1 - 2 : 3
1 - 4 : 5
0 - 3 : 6
Total cost : 16
```

Experiment No: 07

Experiment Name: Implementation of 0/1 Knapsack Problem.

Theory: The 0/1 Knapsack problem is an optimization problem where a set of items with weights and values must be selected to maximize total value without exceeding a weight limit. Dynamic programming is used to solve this efficiently.

Algorithm:

Create a DP table where each cell represents the maximum value at a given weight capacity.

Iterate through items:

a. If the item fits in the current capacity, choose the maximum of including or excluding it.

b. Update the DP table accordingly.

The final cell contains the optimal solution**.**

Code:

```
#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int W, int n, int weight[], int value[]) {
    int dp[n + 1][W + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (weight[i - 1] <= w)
                dp[i][w] = max(value[i - 1] + dp[i - 1][w - weight[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    printf("\nDP Matrix:\n");
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            printf("%4d", dp[i][w]);  /
        }
        printf("\n");
    }
```

```c
    return dp[n][W];
}

int main() {
    int W, n;
    printf("Enter the knapsack size/capacity: ");
    scanf("%d", &W);

    printf("Enter the number of items: ");
    scanf("%d", &n);

    int weight[n], value[n];

    printf("Enter the weights: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &weight[i]);
    }

    printf("Enter the values: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &value[i]);
    }
    int max_value = knapsack(W, n, weight, value);
    printf("\nMaximum value in Knapsack = %d\n", max_value);
    return 0;
}

```

Input:

```
PS D:\4th Semester\Algorithms 1\Codes>
c -o Knapsack } ; if ($?) { .\Knapsack
Enter the knapsack size/capacity: 5
Enter the number of items: 4
Enter the weights: 3 2 5 4
Enter the values: 4 3 6 5
```

Output:

```
DP Matrix:
   0    0    0    0    0    0
   0    0    0    4    4    4
   0    0    3    4    4    7
   0    0    3    4    4    7
   0    0    3    4    5    7

Maximum value in Knapsack = 7
PS D:\4th Semester\Algorithms 1\Codes>
```

Experiment No:08

Experiment Name: Implementation of Kruskal's Algorithm.

Theory: Kruskal's algorithm constructs a Minimum Spanning Tree (MST) by sorting all edges and adding the smallest ones while avoiding cycles. It efficiently finds MSTs using the Union-Find data structure.

Algorithm:

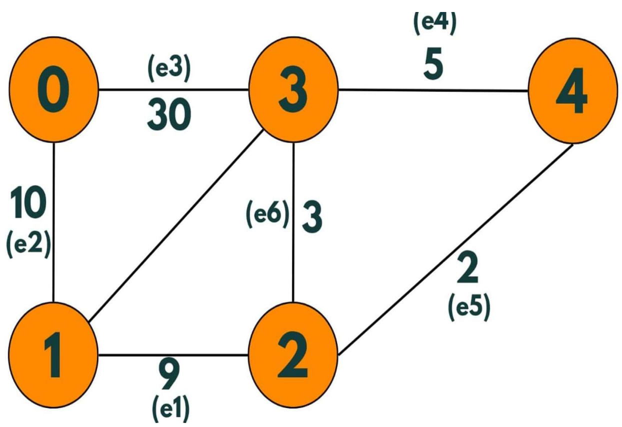
1. Sort all edges in ascending order of weight.
2. Pick the smallest edge and check if it forms a cycle using Union-Find.
3. If no cycle is formed, include the edge in the MST.
4. Repeat until the MST contains (V-1) edges.

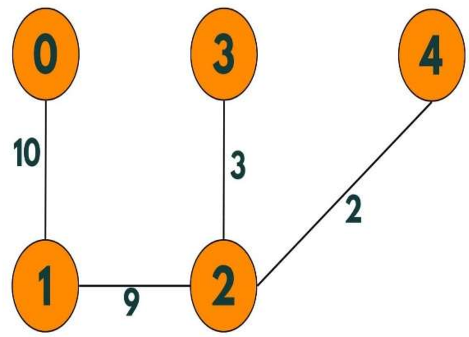
Code:

```
#include <stdio.h>

int find(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return parent[i] = find(parent, parent[i]);  // Path compression
}

void unionSets(int parent[], int rank[], int x, int y) {
    int rootX = find(parent, x);
    int rootY = find(parent, y);

    if (rank[rootX] > rank[rootY])
        parent[rootY] = rootX;
    else if (rank[rootX] < rank[rootY])
        parent[rootX] = rootY;
    else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
}

void sortEdges(int edges[][3], int E) {
    for (int i = 0; i < E - 1; i++) {
        for (int j = 0; j < E - i - 1; j++) {
            if (edges[j][2] > edges[j + 1][2]) {
                // Swap edges[j] and edges[j + 1]
                int temp1 = edges[j][0], temp2 = edges[j][1], temp3 = edges[j][2];
                edges[j][0] = edges[j + 1][0];
                edges[j][1] = edges[j + 1][1];
                edges[j][2] = edges[j + 1][2];
                edges[j + 1][0] = temp1;
                edges[j + 1][1] = temp2;
```

```
                edges[j + 1][2] = temp3;
            }
        }
    }
}

void kruskal(int edges[][3], int V, int E) {
    int parent[V];  // Parent array for Union-Find
    int rank[V];    // Rank array for Union by Rank

    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
    sortEdges(edges, E);

    printf("Edges in the Minimum Spanning Tree:\n");
    int mstWeight = 0, edgeCount = 0;

    for (int i = 0; i < E && edgeCount < V - 1; i++) {
        int src = edges[i][0], dest = edges[i][1], weight = edges[i][2];

        if (find(parent, src) != find(parent, dest)) {
            printf("%d -- %d  | Weight: %d\n", src, dest, weight);
            mstWeight += weight;
            unionSets(parent, rank, src, dest);
            edgeCount++;
        }
    }

    if (edgeCount == V - 1) {
        printf("Total Weight of MST: %d\n", mstWeight);
    } else {
        printf("MST cannot be formed. Not enough edges.\n");
    }
}

int main() {
    int V, E;
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);

    int edges[E][3];

    printf("Enter edges (source destination weight):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &edges[i][0], &edges[i][1], &edges[i][2]);
    }
    kruskal(edges, V, E);
    return 0;
}
```

Graph: **MST**:





Input:

```
rushkal } ; if ($?) { .\krushkal }
Enter number of vertices: 5
Enter number of edges: 6
Enter edges (source destination weight):
1 2 9
1 0 10
0 3 30
3 4 5
2 4 2
2 3 3
```

Output:

```
Edges in the Minimum Spanning Tree:
2 -- 4  | Weight: 2
2 -- 3  | Weight: 3
1 -- 2  | Weight: 9
1 -- 0  | Weight: 10
Total Weight of MST: 24
PS D:\4th Semester\Algorithms 1\Codes>
```