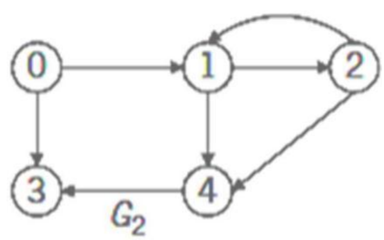
**(MCAL11) Advanced Data Structures LabTitle – Graphs**

- **Definition of Graph –**
  - A graph G is a pair, G=(V, E), where V is a finite nonempty set, called the set of **vertices** of G and E  V X V. That is, the elements of E are pairs of elements of V. E is called the set of **edges** of G. G is called trivial if it has only one vertex.

OR

  - A GRAPH is a collection of nodes called VERTICES, and a collection of line segments connecting vertices, called EDGES.

- Let V(G) denote the set of vertices, and E(G) denote the set of edges of a graph G.




Vertices V(G2) = {0, 1, 2, 3, 4}
Edges E(G2) = {(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)}

**Cost Adjacency Matrix –**

- Let G be a graph with n vertices, where n > 0.
- Let V(G) = {v1, v2, ..., vn}.
- The Cost adjacency matrix w is a two-dimensional nXn matrix such that the (i, j)th entry of w is weight 'wt' of edge from vi to vj; otherwise, the (i, j)th entry is 0 or infinity. That is,

$$w(i,j) = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the directed edge}(i,j) & \text{if } i \neq j \text{ and } \langle i,j \rangle \in E \\ \infty & \text{if } i \neq j \text{ and } \langle i,j \rangle \notin E \end{cases}$$

**Operations on a Graph –**

- The operations commonly performed on a graph are as follows:
  - Create the graph. That is, store the graph in computer memory using a particular graph representation.
  - Traverse the graph (print the graph).
  - Find the shortest path.
  - Find the MST

**Algorithm for Graph – Create graph and BFS and DFS Traversers:**

**1. Initialize Graph**

1. Input the number of vertices, numVertices.
2. Create a 2D array, adjacencyMatrix, of size numVertices x numVertices to represent the edges.

3. Initialize all entries of the matrix to 0.

**2. Add Edges**

4. Input the number of edges, numEdges.
5. For each edge:
   - Input the source vertex (src) and destination vertex (dest).
   - Set adjacencyMatrix[src][dest] = 1 and adjacencyMatrix[dest][src] = 1 (for undirected graph).

**3. Display Adjacency Matrix**

6. Print the adjacency matrix by iterating through the rows and columns of adjacencyMatrix.

**4. Breadth-First Search (BFS)**

7. Input the starting vertex for BFS, startVertex.
8. Initialize a visited array of size numVertices with all values as false.
9. Initialize a queue to manage BFS traversal:
   - Mark startVertex as visited (visited[startVertex] = true) and enqueue it.
10. While the **queue** is not empty:
    - Dequeue a vertex (currentVertex) and print it.
    - For all vertices connected to currentVertex (check adjacencyMatrix[currentVertex][i]):
      - If the vertex is not visited, mark it as visited and enqueue it.

**5. Depth-First Search (DFS)**

11. Input the starting vertex for DFS, startVertex.
12. Initialize a visited array of size numVertices with all values as false.
13. Initialize a stack to manage DFS traversal:
    - Push startVertex onto the stack and mark it as visited.
14. While the **stack** is not empty:
    - Pop a vertex (currentVertex) from the stack and print it.
    - For all vertices connected to currentVertex (check adjacencyMatrix[currentVertex][i] in reverse order to mimic recursion):
      - If the vertex is not visited, mark it as visited and push it onto the stack.

**6. Main Method**

15. Create a graph with numVertices using the Graph class.
16. Add edges as described in step 5.
17. Display the adjacency matrix.
18. Perform BFS traversal starting from the input vertex.
19. Perform DFS traversal starting from the input vertex.

**Minimum Spanning Tree (MST):**

A Minimum Spanning Tree (MST) is a subset of edges in a connected, undirected graph that connects all vertices with the minimum total edge weight and no cycles.

**Properties of MST:**

1. Covers all vertices of the graph.
2. Contains exactly V−1 edges, where V is the number of vertices.
3. The total weight of the edges is minimized.
4. Does not contain cycles.

**Algorithms to Find MST:**

1. **Kruskal's Algorithm** (Edge-based approach)
2. **Prim's Algorithm** (Vertex-based approach)

**Kruskal's Algorithm to Find Minimum Spanning Tree (MST):**

1. **Input Handling**:
   - Read the number of vertices V and edges E from the user.
   - Initialize an array edges of size E to store all the edges of the graph.
2. **Edge Representation**:
   - Define a class Edge to represent an edge with attributes src (source vertex), dest (destination vertex), and weight (weight of the edge).
   - Add edges to the graph by storing them in the edges array.
3. **Sort Edges by Weight**:
   - Use a sorting algorithm (e.g., Bubble Sort) to arrange the edges in ascending order of weight.
4. **Initialize Union-Find Structures**:
   - Create two arrays parent and rank for Union-Find operations:
     - parent[i] stores the parent of vertex i.
     - rank[i] is used to keep track of the rank (depth) of trees in Union-Find.
5. **Union-Find Initialization**:
   - For each vertex i (0 to V-1), set parent[i] = i (each vertex is its own parent initially).
   - Set rank[i] = 0 for all vertices.
6. **MST Construction**:
   - Create an array mst to store the edges included in the MST.
   - Initialize mstIndex = 0 to track the number of edges added to the MST.
   - Initialize mstWeight = 0 to track the total weight of the MST.
7. **Process Sorted Edges**:
   - For each edge in the sorted edges array:
     - Use the findParent method to find the root parents of the source and destination vertices of the edge.
     - If the root parents are different, include the edge in the MST:
       - Add the edge to the mst array.
       - Increment mstIndex and add the edge's weight to mstWeight.
       - Use the union method to merge the sets of the source and destination vertices.
8. **Stop Condition**:

   - Stop processing edges once mstIndex = V-1 (MST is complete).
9. **Output the Result**:
   - Print the edges included in the MST along with their weights.
   - Print the total weight of the MST.
10. **Utility Functions**:
    - **findParent:** Implements path compression to efficiently find the root parent of a vertex in the Union-Find structure.
    - **union:** Merges two disjoint sets in the Union-Find structure based on rank to keep the tree balanced.

**Complexity:**

- Sorting edges: O(E log E)
- Union-Find operations: Approximately O(E · α(V)), where α(V) is the inverse Ackermann function (nearly constant).

**Prim's Algorithm to Find Minimum Spanning Tree (MST):**

**Exercise:**

1. **Write a Java program to construct a graph using adjacency matrix and implement DFS and BFS traversing of it.**

```
package graphpkg;
import java.util.Scanner;

public class Graph {
    private int[][] adjacenyMatrix;
    private int numVertices;
    public Graph(int numVertices) {
        this.numVertices=numVertices;
        adjacenyMatrix=new int[numVertices][numVertices];
    }
    public void addEdge(int start,int end) {
        adjacenyMatrix[start][end]=1;
        adjacenyMatrix[end][start]=1;
    }
    public void bfs(int startVertex) {
        boolean[] visited=new boolean[numVertices];
        int[] queue=new int[numVertices];
        int front=0,rear=0;
        visited[startVertex]=true;
        queue[rear++]=startVertex;
        System.out.println("BFS traversal starting from vertex : "+startVertex+ " :
");
        while(front<rear) {
            int currentVertex=queue[front++];
            System.out.print(currentVertex+" ");
            for(int i=0;i<numVertices;i++) {
                if(adjacenyMatrix[currentVertex][i] == 1 && !visited[i]) {
                    visited[i]=true;
                    queue[rear++]=i;
                }
            }
        }
        System.out.println();
```

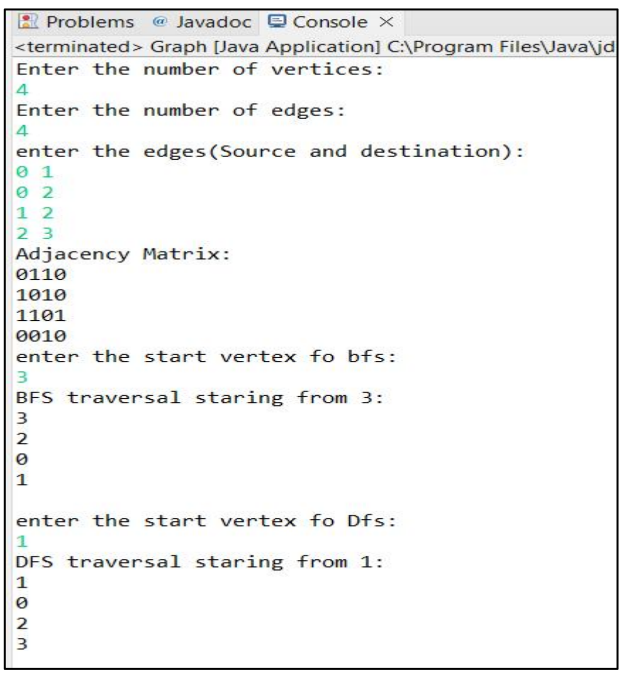
```
	}
	public void dfs(int startVertex) {
		boolean[] visited=new boolean[numVertices];
		int[] stack=new int[numVertices];
		int top=-1;
		stack[++top]=startVertex;
		visited[startVertex]=true;
		System.out.println("DFS traversal starting from vertex "+startVertex+" ");
		while(top>=0) {
			int currentVertex=stack[top--];
			System.out.print(currentVertex+" ");
			for(int i=numVertices-1;i>=0;i--) {
				if(adjacenyMatrix[currentVertex][i]==1 && !visited[i]) {
					visited[i]=true;
					stack[++top]=i;
				}
			}
		}
		System.out.println();
	}
	public void displayAdjacencyMatrix() {
		System.out.println("Adjacency matrix : ");
		for(int i=0;i<numVertices;i++) {
			for(int j=0;j<numVertices;j++) {
				System.out.print(adjacenyMatrix[i][j]+" ");
			}
			System.out.println();
		}
	}

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Scanner sc=new Scanner(System.in);
		System.out.println("Enter the number of vertices : ");
		int numVertices=sc.nextInt();
		Graph gh=new Graph(numVertices);
		System.out.println("Enter number of edges : ");
		int numEdges=sc.nextInt();
```

```
                System.out.println("Enter the edges (Source and destination) : ");
                for(int i=0;i<numEdges;i++) {
                        int src=sc.nextInt();
                        int des=sc.nextInt();
                        gh.addEdge(src, des);
                }
                gh.displayAdjacencyMatrix();
                System.out.println("Enter the start vertex for BFS : ");
                int bfsStart=sc.nextInt();
                gh.bfs(bfsStart);
                System.out.println("Enter the start vertex for DFS : ");
                int dfsStart=sc.nextInt();
                gh.bfs(dfsStart);
        }

}
```

**Output**



```
Problems  Javadoc  Console
<terminated> Graph [Java Application] C:\Program Files\Java\jd
Enter the number of vertices:
4
Enter the number of edges:
4
enter the edges(Source and destination):
0 1
0 2
1 2
2 3
Adjacency Matrix:
0110
1010
1101
0010
enter the start vertex fo bfs:
3
BFS traversal staring from 3:
3
2
0
1

enter the start vertex fo Dfs:
1
DFS traversal staring from 1:
1
0
2
3
```

**2. Write a Java program to find the minimum spanning tree of a graph.**

```
package minspanPkg;

import java.util.Scanner;

class Edge{
    int src,dest,weight;
    Edge(int src,int des ,int weight){
            this.src=src;
            this.dest=des;
            this.weight=weight;
    }
}
public class KruskhalAlgorithum {
    private int V;
    private int E;
    private Edge[] edges;
    private int edgeCount=0;
    public KruskhalAlgorithum(int vertices,int edgesCount) {
            this.V=vertices;
            this.E=edgesCount;
            edges=new Edge[edgesCount];

    }
    public void addEdge(int src, int dest,int weight) {
            edges[edgeCount++]=new Edge(src,dest,weight);

    }
    private int findParent(int[] parent,int vertex) {
            if(parent[vertex]!= vertex) {
                    parent[vertex]=findParent(parent, parent[vertex]);
            }
            return parent[vertex];
    }
    public void union(int[] parent,int[] rank,int x,int y) {
            int rootX=findParent(parent, x);
            int rootY=findParent(parent, y);
            if(rootX != rootY) {
                    if(rank[rootX] < rank[rootY]) {
                            parent[rootX]=rootY;

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

```
    }
    private void sortEdges() {
        for(int i=0;i<E-1;i++) {
            for(int j=0;j<E-i-1;j++) {
                if(edges[j].weight>edges[j+1].weight) {
                    Edge temp=edges[j];
                    edges[j]=edges[j+1];
                    edges[j+1]=temp;
                }
            }
        }
    }
    public void kruskalMST() {
        sortEdges();
        int[] parent=new int[V];
        int[] rank=new int[V];
        for(int i=0;i<V;i++) {
            parent[i]=i;
            rank[i]=0;


        }
        Edge[] mst=new Edge[V-1];
        int mstIndex=0;
        int mstWeight=0;
        for(int i=0;i<E;i++) {
            if(mstIndex == V-1) {
                break;
            }
            Edge edge=edges[i];
            int srcParent=findParent(parent, edge.src);
            int destParent=findParent(parent,edge.dest);
            if(srcParent != destParent) {
                mst[mstIndex++]=edge;
                mstWeight+=edge.weight;
                union(parent,rank,srcParent,destParent);
            }
        }
        System.out.println("Edges in the MST : ");
        System.out.println("Src des Weight");
        for(int i=0;i<mstIndex;i++) {
            System.out.println(mst[i].src+"--"+mst[i].dest+"--"+"=="+mst[i].weight);
        }
        System.out.println("Total weight of MST : "+mstWeight);
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the number of vertices : ");
```

```
            int V=sc.nextInt();
            System.out.println("Enter the number of edges : ");
            int E=sc.nextInt();
            KruskhalAlgorithum graph=new KruskhalAlgorithum(V, E);
            System.out.println("Enter the edges in the formate : src dest weight ");
            for(int i=0;i<E;i++) {
                    int src=sc.nextInt();
                    int dest=sc.nextInt();
                    int weight=sc.nextInt();
                    graph.addEdge(src, dest, weight);
            }
            graph.kruskalMST();
            sc.close();
    }

}
```

## Output

```
<terminated> KruskhalAlgorithum [Java Application] C:\Program Files\Java\jdk-2
Enter the number of vertices :
5
Enter the number of edges :
7
Enter the edges in the formate : src dest weight
0 1 2
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9
0 3 6
Edges in the MST :
Src des Weight
0--1--==2
1--2--==3
1--4--==5
0--3--==6
Total weight of MST : 16
```

## Conclusion

A **graph** is a structure of **vertices** (nodes) connected by **edges**, representing relationships like networks or paths. Key operations include creating the graph, representing it via adjacency matrices, and traversing it using **BFS** (Breadth-First Search) for breadthwise exploration and **DFS** (Depth-First Search) for depth-first exploration.

A **Minimum Spanning Tree (MST)** connects all vertices with minimal total edge weight and no cycles.

- **Kruskal's Algorithm** (edge-based) and **Prim's Algorithm** (vertex-based) are efficient methods to find MST.
- These algorithms optimize problems like network design by minimizing connection costs.

Graphs and MSTs are crucial for solving real-world problems efficiently in areas like optimization, navigation, and network analysis.