| Experiment **#11** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT] THANOS |

**Experiment Title: To implement programs on problem solving using Graph Algorithms.**

**Aim/Objective:** To understand the concept and implementation of programs on Graph Algorithms-based Problems.

**Description:** The students will understand DFS , BFS, Single Source Shortest Path and All-Pairs Shortest path algorithms. Students will gain experience in implementing these algorithms and applying them to solve real-world problems.

**Pre-Requisites:**

**Knowledge:** Before beginning this lab, students should have a foundational understanding of:

- The concepts of DFS , BFS,Single Source Shortest Path, and All-Pairs Shortest path algorithms.
- Mathematical Background: Logarithmic complexity analysis for tree operations.
- Understanding of height-balanced properties.

**Tools:** Code Blocks/Eclipse IDE.

**Pre-Lab:**

1. Write a program to find the shortest path from a starting point (e.g., a person's location) to the nearest exit in a building represented as a grid using BFS.

- **Procedure/Program:**

```
import java.util.LinkedList;
import java.util.Queue;

class Point {
   int x, y;

   Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}

public class Main{
   static final int MAX = 100;
```

```
public static int bfs(int[][] grid, int n, int m, Point start) {
    int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    boolean[][] visited = new boolean[MAX][MAX];
    Queue<Point> q = new LinkedList<>();
    q.add(start);
    visited[start.x][start.y] = true;
    int distance = 0;

    while (!q.isEmpty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            Point current = q.poll();
            if (grid[current.x][current.y] == 2) {
                return distance;
            }
            for (int[] direction : directions) {
                int newX = current.x + direction[0];
                int newY = current.y + direction[1];
                if (newX >= 0 && newX < n && newY >= 0 && newY < m &&
                    grid[newX][newY] != 1 && !visited[newX][newY]) {
                    visited[newX][newY] = true;
                    q.add(new Point(newX, newY));
                }
            }
        }
        distance++;
    }
    return -1;
}

public static void main(String[] args) {
    int[][] grid = {
        {0, 0, 1, 0, 2},
        {0, 1, 0, 0, 0},
        {0, 0, 0, 1, 0},
        {1, 0, 1, 0, 0},
        {0, 0, 0, 0, 0}
    };
```

```
        Point start = new Point(0, 0);
        int n = 5, m = 5;
        int result = bfs(grid, n, m, start);
        if (result != -1) {
            System.out.println("Shortest path to exit is: " + result);
        } else {
            System.out.println("No exit found.");
        }
    }
}
```

- **Data and Results:**

**Data**

A grid represents a building with obstacles and exits.

**Result**

Shortest path to the nearest exit is determined using BFS.

- **Analysis and Inferences:**

**Analysis**

Breadth-first search explores paths layer-by-layer, ensuring shortest route.

**Inferences**

BFS efficiently finds the shortest exit path in grid-based environments.

**In-Lab:**

1. You are given a connected, undirected graph representing a network of cities. Each edge represents a road between two cities with a given cost. Write a program to find the Minimum Spanning Tree (MST), which connects all cities with the minimum total cost.

- **Procedure/Program**:

```
import java.util.Arrays;

public class Main {

  static final int MAX = 100;

  static final int INF = 99999;

  static int[][] graph = new int[MAX][MAX];

  static int[] parent = new int[MAX];

  static int[] key = new int[MAX];

  static boolean[] visited = new boolean[MAX];

  static int numVertices;


  static void primMST() {

    Arrays.fill(key, INF);

    Arrays.fill(visited, false);

    key[0] = 0;

    parent[0] = -1;
```

```
    for (int count = 0; count < numVertices - 1; count++) {
        int minKey = INF, minIndex = -1;
        for (int v = 0; v < numVertices; v++) {
            if (!visited[v] && key[v] < minKey) {
                minKey = key[v];
                minIndex = v;
            }
        }
        visited[minIndex] = true;

        for (int v = 0; v < numVertices; v++) {
            if (graph[minIndex][v] != 0 && !visited[v] && graph[minIndex][v] < key[v]) {
                parent[v] = minIndex;
                key[v] = graph[minIndex][v];
            }
        }
    }
}

static void printMST() {
    System.out.println("Edge \tWeight");
```

```
    for (int i = 1; i < numVertices; i++) {
        System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
    }
}

public static void main(String[] args) {
    numVertices = 5;

    int[][] exampleGraph = {
        {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}
    };

    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            graph[i][j] = exampleGraph[i][j];
        }
    }
```

```
        primMST();
        printMST();
    }
}
```

- **Data and Results:**

**Data**

Graph with 5 vertices and weighted edges representing city roads.

**Result**

Minimum Spanning Tree (MST) found with the least total cost.

- **Analysis and Inferences:**

**Analysis**

Prim's algorithm selects edges with the smallest weights efficiently.

**Inferences**

MST connects all cities while minimizing the total connection cost.

2. You are given a city map represented as a graph. Write a program to determine if there is a path between two given intersections (nodes). Use Depth-First Search (DFS) or Breadth-First Search (BFS) to solve the problem.

- **Procedure/Program**:

```
public class Main {
   private static final int MAX = 100;
   private int[][] adj = new int[MAX][MAX];
   private int[] visited = new int[MAX];
   private int numNodes;

   public void initGraph(int nodes) {
      this.numNodes = nodes;
      for (int i = 0; i < nodes; i++) {
         visited[i] = 0;
         for (int j = 0; j < nodes; j++) {
            adj[i][j] = 0;
         }
      }
   }

   public void addEdge(int src, int dest) {
      adj[src][dest] = 1;
      adj[dest][src] = 1;
   }

   private void dfs(int node, int target, int[] found) {
      visited[node] = 1;
      if (node == target) {
         found[0] = 1;
         return;
      }
      for (int i = 0; i < numNodes; i++) {
         if (adj[node][i] == 1 && visited[i] == 0) {
            dfs(i, target, found);
         }
```

```
        }
    }

    public boolean isPath(int start, int end) {
        int[] found = {0};
        dfs(start, end, found);
        return found[0] == 1;
    }

    public static void main(String[] args) {
        Main g = new Main();
        g.initGraph(5);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        int start = 0, end = 3;
        if (g.isPath(start, end)) {
            System.out.println("Path exists between " + start + " and " + end);
        } else {
            System.out.println("No path exists between " + start + " and " + end);
        }
    }
}
```

- **Data and Results:**

## Data

Graph with 5 nodes, edges define connectivity between intersections.

## Result

DFS determines if a path exists between two intersections.

- **Analysis and Inferences:**

## Analysis

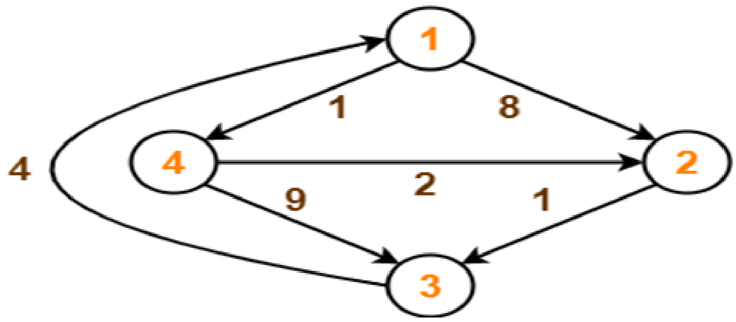
Graph traversal explores connected nodes to check reachability efficiently.

## Inferences

Path existence confirms connectivity; absence indicates graph disconnection.

**Post-Lab:**

Consider the following directed weighted graph and Using Floyd-Warshall Algorithm, find the shortest path distance between every pair of vertices.




- **Procedure/Program**:

## Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

## Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

## Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

- **Data and Results:**

## Data

Graph with weighted edges, vertices, and adjacency matrix representation.

## Result

Final shortest path matrix obtained using Floyd-Warshall algorithm.

- **Analysis and Inferences :**

## Analysis

Each iteration refines shortest paths by considering intermediate vertices.

## Inferences

Algorithm efficiently finds shortest paths between all vertex pairs.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the time complexity of DFS?

   - **O(V + E)**, where V is vertices and E is edges.

2. How can BFS be used to find the shortest path in an unweighted graph?

   - Uses **level-order traversal**, marking shortest distance from the source.

3. How can you implement Dijkstra’s Algorithm using a priority queue?

   - Uses **min-heap** to pick the smallest distance vertex, updating neighbors.

4. What is the Floyd-Warshall Algorithm?

   - **Dynamic Programming** approach for **all-pairs shortest paths** in **$O(V^3)$**.

5. How is Prim's Algorithm different from Kruskal's Algorithm?

- **Prim's** grows MST from a node, **Kruskal's** sorts edges and merges sets.

| | |
|---|---|
| **Evaluator Remark (if Any):** | **Marks Secured ___out of 50** |
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**