PROJECT 2

Graph Algorithms and Related Data Structures

College of Computing and Informatics, University of North Carolina at Charlotte.

Algorithms and Data Structures

Dr. Dewan Tanvir Ahmed

ITCS 6114/8114 – SPRING 2024

Submitted By:

Nishant Acharekar – 801363902

Project 2: Single-Source Shortest Path Algorithm

The Single-Source Shortest Path (SSSP) algorithm is a fundamental idea in graph theory and computer science that finds the most efficient paths from one starting vertex to all other vertices in a weighted graph. In the setting of a weighted directed graph, this algorithm searches the path with the lowest total weight between any two vertices. The SSSP issue seeks to determine the shortest path from a selected source vertex to all remaining vertices in the graph.

Consider a connected weighted graph, denoted as G. The "length" of a path, represented by P, is determined by adding up the weights of its constituent edges. If the path P is composed of edges e0, e1,..., ek-1, then its length, indicated as w(P), is calculated as:

$$w(P) = \sum_{i=0}^{k-1} w(e_i)$$

The "distance" from a vertex v to a starting vertex s corresponds to the shortest path's length between s and v. This distance is symbolized as **d (s, v).** If there's no existing path between s and v, the distance is infinity:

$$\mathbf{d\ (s,\ v) = +\infty\ if\ no\ path\ exists}$$

Dijkstra's Algorithm:

- This algorithm calculates the shortest distances from a specified starting vertex, denoted as **s**, to all other vertices in the graph.
- Dijkstra's algorithm operates on a greedy approach, presuming that all edges within graph **G** have non-negative weights.
- Initially, we establish a "cloud" or collection of vertices, starting with the vertex **s** and progressively encompassing all vertices.
- Each vertex **v** is assigned a label, **d(v)**, indicating its distance from **s** within the subgraph that includes the current cloud and its neighboring vertices.
- At every iteration: We incorporate into the cloud the vertex **u** that lies outside the current cloud but possesses the smallest distance label, **d(u).** Subsequently, we adjust the labels of vertices adjacent to **u** accordingly.

Pseudo code:

```
DIJKSTRA(G, w, s)
   INITIALIZE_SOURCE(G, s)
   S = EmptySet
   Q = Queue containing all vertices in G

   while Q is not empty:
      u = ExtractMin(Q based on distance)
      ADD_TO_SET(S, u)
```

```
        for each neighbor v of u:
            RELAX_EDGE(u, v, w)

INITIALIZE_SOURCE(G, s)
    for each vertex v in G:
        v.distance = Infinity
        v.predecessor = Null
    sourceVertex.distance = 0

RELAX_EDGE(u, v, w)
    if v.distance > (u.distance + weight of edge (u,v)):
        v.distance = u.distance + weight of edge (u,v)
        v.predecessor = u
```

Edge Relaxation:

Edge Relaxation is an important topic in graph theory, particularly when calculating the shortest paths within a graph. This notion is fundamental to algorithms like Dijkstra's and Bellman-Ford. Edge relaxation is fundamentally about revising the current estimate of the shortest path to a vertex by changing the weights of the edges that lead to it.

Let us decode the symbols utilized in this concept.

- d[u]: The shortest known distance from the starting vertex, denoted as 'S', to vertex 'u'.
- d[v]: The shortest known distance from the starting vertex, 'S', to vertex 'v'.
- c(u, v): The weight or cost of the edge connecting vertex 'u' to vertex 'v'.

The formula d[v] = min{d[v], d[u] + c(u, v)} captures the core of edge relaxation. It determines whether the current shortest path to vertex 'v' (d[v]) is longer than the path through vertex 'u' plus the edge weight from 'u' to 'v'. If it is, the shortest path to 'v' is changed to reflect the new shorter path.

In a more algorithmic representation, the relaxation process can be defined as:

```
if (d[u] + c(u, v) < d[v]) then

{

    d[v] = d[u] + c(u, v)

}
```

Analysis of Runtime Complexity:

Let's look at the time complexity of using a binary min-heap as the priority queue in Dijkstra's algorithm, where |E| represents the number of edges and |V| represents the number of vertices in the graph.

1. **Inserting vertices into the heap:**
    a. Adding all |V| vertices to the binary min-heap takes linear time, represented as O(|V|). This includes the heapify procedure discussed in our lectures.

2. **Update Priority Values in the Priority Queue:**
    a. To change a vertex's priority, it is deleted and reinserted into the binary min-heap. Using a hash map for instantaneous indexing, this procedure takes O(log |V|) time for each update.
    b. Considering each vertex's neighbors and assuming a maximum of |E| edges, the aggregate time for all these updates is O(|E| log |V|).

3. **Deriving updated priority values:**
    a. Leveraging constant-time access to edge weights and vertex priorities ensures that updating priorities takes O(|E|) time and is executed O(|E|) times in total.

4. **Remove Vertices from the Priority Queue:**
    a. Each vertex is extracted from the priority queue once, for a total of |V| times. Given that each removal takes O(log |V|), the total time for all removals is O(|V| log |V|).

5. **Verifying the Priority Queue's emptiness:**
    a. The priority queue's emptiness is checked O(|V|) times before each vertex is removed. This check operates in real-time.

6. **Traversal across Vertex Neighbors:**
    a. The degree of a vertex limits the amount of time available for traversing its neighbors. Throughout the algorithm's execution, the whole traversal of all vertex neighbors takes O(|E|) time.

Overall Time Complexity: Dijkstra's method with a binary min-heap as the priority queue has an overall time complexity of O((|E| + |V|) log |V|).

Used Data Structures and Their Effect on Runtime:

1. Adjacency List (List<List<Node>> ipGraph):

a. The input graph is represented as an adjacency list. Every vertex corresponds to a list of nodes, with each node representing a nearby vertex and its edge weight.
b. Using an adjacency list makes it easier to find a vertex's neighbors, making it a better option for an adjacency matrix.

2. PriorityQueue<Node> priorityQueue:

a. A binary heap-based priority queue contains nodes and their associated costs.
b. This priority queue makes it easier to identify the node with the lowest cost throughout each algorithm iteration.
c. Given the binary heap's nature, retrieving the minimum element takes O(log N) time. Thus, the priority queue operations do not outweigh the algorithm's overall time complexity.

3. Arrays (int[] overallCost, int[] currentParent):

a. We use arrays to store the cost and parent information for each vertex as the algorithm progresses.
b. The 'overallCost' array records the minimal cost of accessing each vertex starting from the source.
c. Meanwhile, the 'currentParent' array tracks each vertex's predecessor along the shortest path tree.
d. Direct array access ensures a consistent O(1) time complexity for updating and retrieving vertex-specific information.

4. Set<Integer> C:

a. A set manages vertices that have been integrated into the finalized shortest path tree, i.e. those with determined shortest paths.
b. This collection streamlines operations by avoiding redundant computations for processed vertices.
c. The inherent set operations have an average O(1) time complexity, demonstrating their efficiency.

Dijkstra's approach has an overall runtime complexity of O((V + E) * log V), which includes a binary heap for the priority queue and an adjacency list. V represents the vertex count, while E represents the edge count. The chosen data structures considerably improve the algorithm's efficiency.

Implementation:

Proj2AlgoDs.java

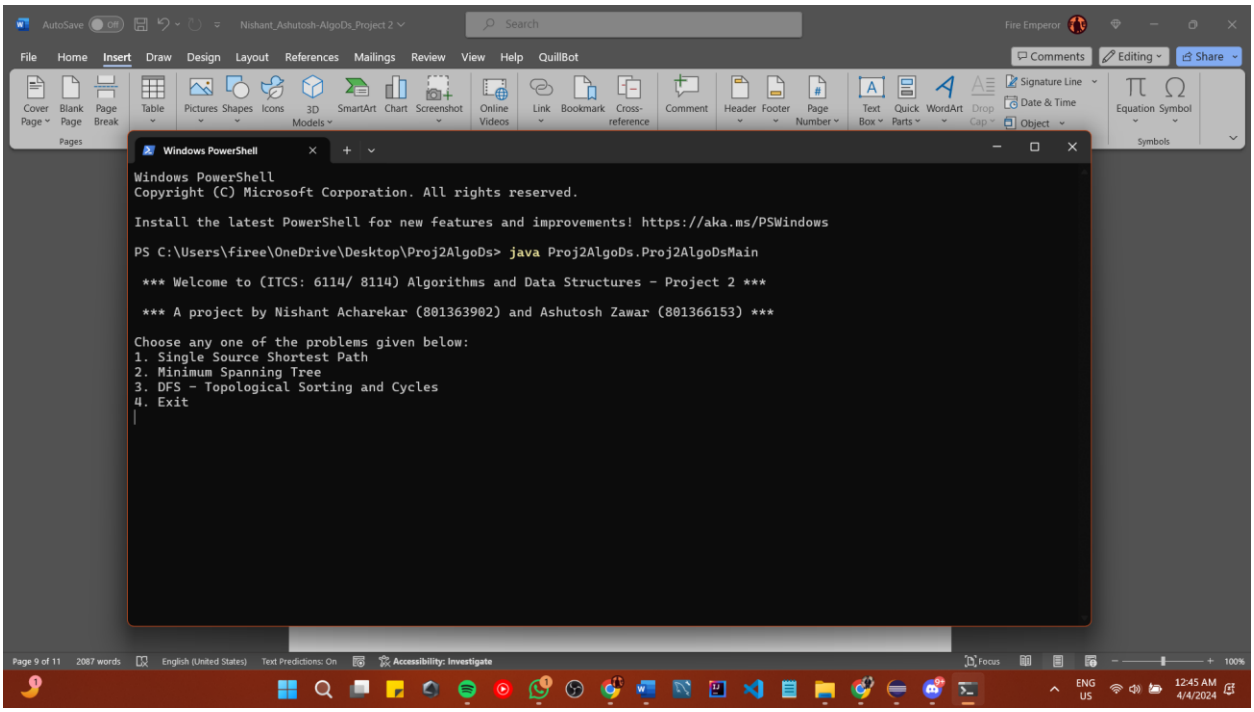
```
//Nishant Acharekar - 801363902 & Ashutosh Zawar (801366153)
package Proj2AlgoDs;
import java.util.Scanner;
public class Proj2AlgoDsMain {
    public static void main(String[] args) throws Exception{
        // Create objects for the three algorithms.
        SingleSourceShortestPathAlgorithm problem1 = new
SingleSourceShortestPathAlgorithm();
        MinimumSpanningTreeAlgorithm problem2 = new
MinimumSpanningTreeAlgorithm();
        DFSAlgorithm problem3 = new DFSAlgorithm();

     // A scanner object for user input.
        Scanner scanner = new Scanner(System.in);
        int myChoice;
        int ipFileChoice;
        String ipFilePath;
        boolean stop = false;

        // Display welcome message
        System.out.println("\n *** Welcome to (ITCS: 6114/ 8114) Algorithms and
Data Structures - Project 2 ***");
        System.out.println("\n *** A project by Nishant Acharekar (801363902) and
Ashutosh Zawar (801366153) ***");
        do {
            // Display menu options
            System.out.println();
            System.out.println("Choose any one of the problems given below:");
            System.out.println("1. Single Source Shortest Path");
            System.out.println("2. Minimum Spanning Tree");
            System.out.println("3. DFS - Topological Sorting and Cycles");
            System.out.println("4. Exit");

            // Get user choice
            myChoice = scanner.nextInt();
            switch (myChoice) {
                case 1:
                    // Single Source Shortest Path problem
                    System.out.println("Select any one input file from the
following:");
                    System.out.println("1. Input-1 (Undirected Graph)");
```

```
                    System.out.println("2. Input-2 (Undirected Graph)");
                    System.out.println("3. Input-3 (Directed Graph)");
                    System.out.println("4. Input-4 (Directed Graph)");

                    // Get input file choice
                    ipFileChoice = scanner.nextInt();
                    // Validate input file choice
                    if(ipFileChoice < 1 || ipFileChoice > 4) {
                        System.out.println("Invalid option. Please enter valid
option !");
                        break;
                    }

                    // Get input file path based on choice
                    ipFilePath = getIpFilePath(ipFileChoice, 1);

                    // Read input file, execute algorithm and display output
                    problem1.readIpFile(ipFilePath);
                    problem1.executeMainAlgorithm();
                    problem1.displayOp();
                    break;

                case 2:
                    // Minimum Spanning Tree problem
                    System.out.println("Select any one input file from the
following:");
                    System.out.println("1. Input-1 (Undirected Graph)");
                    System.out.println("2. Input-2 (Undirected Graph)");
                    System.out.println("3. Input-3 (Undirected Graph)");
                    System.out.println("4. Input-4 (Undirected Graph)");
                    ipFileChoice = scanner.nextInt();
                    if(ipFileChoice < 1 || ipFileChoice > 4) {
                        System.out.println("Invalid option. Please enter valid
option !");
                        break;
                    }
                    ipFilePath = getIpFilePath(ipFileChoice, 2);
                    problem2.readIpFile(ipFilePath);
                    problem2.executeMainAlgorithm();
                    problem2.displayOp();
                    break;

                case 3:
                    // DFS - Topological Sorting and Cycles problem
```

```
                    System.out.println("Select any one input file from the
following:");
                    System.out.println("1. Input-1 (Directed Graph (Acyclic)");
                    System.out.println("2. Input-2 (Directed Graph (Cyclic)");
                    System.out.println("3. Input-3 (Directed Graph (Acyclic)");
                    System.out.println("4. Input-4 (Directed Graph (Cyclic)");
                    ipFileChoice = scanner.nextInt();
                    if(ipFileChoice < 1 || ipFileChoice > 4) {
                        System.out.println("Invalid option. Please enter valid
option !");
                        break;
                    }
                    ipFilePath = getIpFilePath(ipFileChoice, 3);
                    problem3.readIpFile(ipFilePath);
                    problem3.executeMainAlgorithm();
//                    problem3.displayOp();
                    problem3.reset();
                    break;

                case 4:
                    // Exit the program
                    stop = true;
                    break;

                default:
                    // Invalid choice
                    System.out.println("Invalid option. Please enter valid option
!");
                    break;
            }
        } while (!stop);
    }

 // Method for determining input file path based on user selection and problem
kind
    public static String getIpFilePath(int myChoice, int problem) {
        String path = "";

        // Adjust choice based on problem type
        if(problem == 2)
            myChoice += 4;
        else if (problem == 3)
            myChoice += 8;

        // Assign file path based on adjusted choice
```

```
        switch (myChoice) {
            case 1:
                path = "./InputFiles/input1.txt";
                break;
            case 2:
                path = "./InputFiles/input2.txt";
                break;
            case 3:
                path = "./InputFiles/input3.txt";
                break;
            case 4:
                path = "./InputFiles/input4.txt";
                break;
            case 5:
                path = "./InputFiles/input5.txt";
                break;
            case 6:
                path = "./InputFiles/input6.txt";
                break;
            case 7:
                path = "./InputFiles/input7.txt";
                break;
            case 8:
                path = "./InputFiles/input8.txt";
                break;
            case 9:
                path = "./InputFiles/input9.txt";
                break;
            case 10:
                path = "./InputFiles/input10.txt";
                break;
            case 11:
                path = "./InputFiles/input11.txt";
                break;
            case 12:
                path = "./InputFiles/input12.txt";
                break;
            default:
                break;
        }
        return path;
    }
}
```

Output of Main Screen:



SingleSourceShortestPathAlgorithm.java

```
//Nishant Acharekar - 801363902 & Ashutosh Zawar (801366153)

package Proj2AlgoDs;
import java.io.File;
import java.io.FileReader;
import java.util.*;
public class SingleSourceShortestPathAlgorithm {

    // FileReader and Scanner objects for reading input file
    private FileReader readFile;
    private Scanner scanFile;

    // String to represent vertices using letters
    private final String VERTEX_INDEX_TRACKING = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Data structures for graph representation and algorithm variables
    private List<List<Node>> ipGraph; // Adjacency list to represent the graph
    private int sourceVertex, noOfVertices, noOfEdges; // Number of vertices,
edges, and source vertex
    private int[] overallCost; // Array to store overall cost to reach each
vertex from source
```

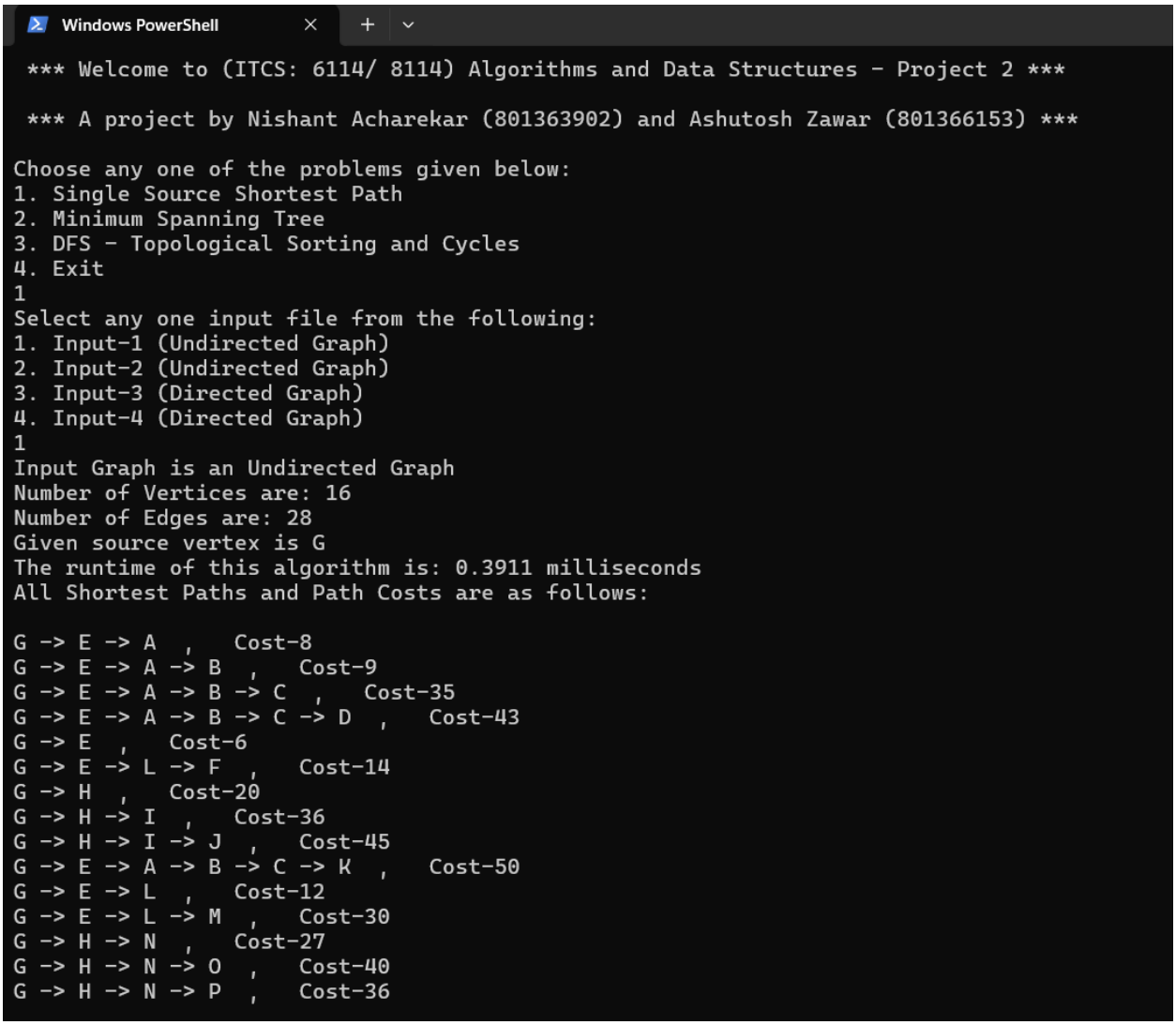
```
    private int[] currentParent; // Array to store parent vertex for each vertex
in shortest path
    private String currentData; // String to hold current line of data from input
file
    private boolean isDirectedGraph;  // Flag to indicate if the graph is
directed or undirected

    // Method to read input file and populate graph data structures
    public void readIpFile(String file) throws Exception {
    File inputFile = new File(file);
    readFile = new FileReader(inputFile);
    scanFile = new Scanner(readFile);
    currentData = scanFile.nextLine();

    // Read first line to determine graph type, number of vertices, and edges
    if(currentData != null) {
        String[] t = currentData.split(" ");
        if(t[2].equals("U")) {
            isDirectedGraph = false;
            System.out.println("Input Graph is an Undirected Graph");
        }
        else {
            isDirectedGraph = true;
            System.out.println("Input Graph is a Directed Graph");
        }

        // Extract number of vertices and edges
        noOfVertices = Integer.parseInt(t[0]);
        noOfEdges = Integer.parseInt(t[1]);

        System.out.println("Number of Vertices are: " + noOfVertices);
        System.out.println("Number of Edges are: " + noOfEdges);

        // Initialize adjacency list for graph representation
        ipGraph = new ArrayList<List<Node>>();
        for (int i = 0; i < noOfVertices; i++)
            ipGraph.add(new ArrayList<Node>());

        // Read edge data and populate adjacency list
        for (int i = 0; i < noOfEdges; i++) {
            currentData = scanFile.nextLine();
            t = currentData.split(" ");
            int x = VERTEX_INDEX_TRACKING.indexOf(t[0].charAt(0));
            int y = VERTEX_INDEX_TRACKING.indexOf(t[1].charAt(0));
            ipGraph.get(x).add(new Node(y, Integer.parseInt(t[2])));
```

```
            // Add reverse edge for undirected graph
            if(!isDirectedGraph)
                ipGraph.get(y).add(new Node(x, Integer.parseInt(t[2])));
        }
        // Read source vertex if provided, otherwise set default source vertex
        if(scanFile.hasNext()) {
            currentData = scanFile.nextLine();
            sourceVertex = VERTEX_INDEX_TRACKING.indexOf(currentData.charAt(0));
            System.out.println("Given source vertex is " +
currentData.charAt(0));
        }
        else {
            sourceVertex = 0;
            System.out.println("Source vertex is not provided. Let the source
vertex be: " + VERTEX_INDEX_TRACKING.charAt(sourceVertex));
        }
    }
}
    // Method to execute the main Dijkstra's Algorithm
    public void executeMainAlgorithm() {
    if(ipGraph == null) {
        System.out.println("Graph is not initialized. Please read the input file
first.");
        return;
    }

    // Get the start time
    long startTime = System.nanoTime();
    // Initialization of arrays
    currentParent = new int[noOfVertices];
    overallCost = new int[noOfVertices];

    // Setting initial values for overallCost and currentParent arrays
    for (int i = 0; i < noOfVertices; i++)
        overallCost[i] = Integer.MAX_VALUE;
    overallCost[sourceVertex] = 0;
    currentParent[sourceVertex] = -1;

    // Set to keep track of vertices included in the shortest path tree
    Set<Integer> cloud = new HashSet<Integer>();

    // Priority queue to prioritize vertices based on their cost
    PriorityQueue<Node> priorityQueue = new PriorityQueue<Node>(noOfVertices, new
Node());
```

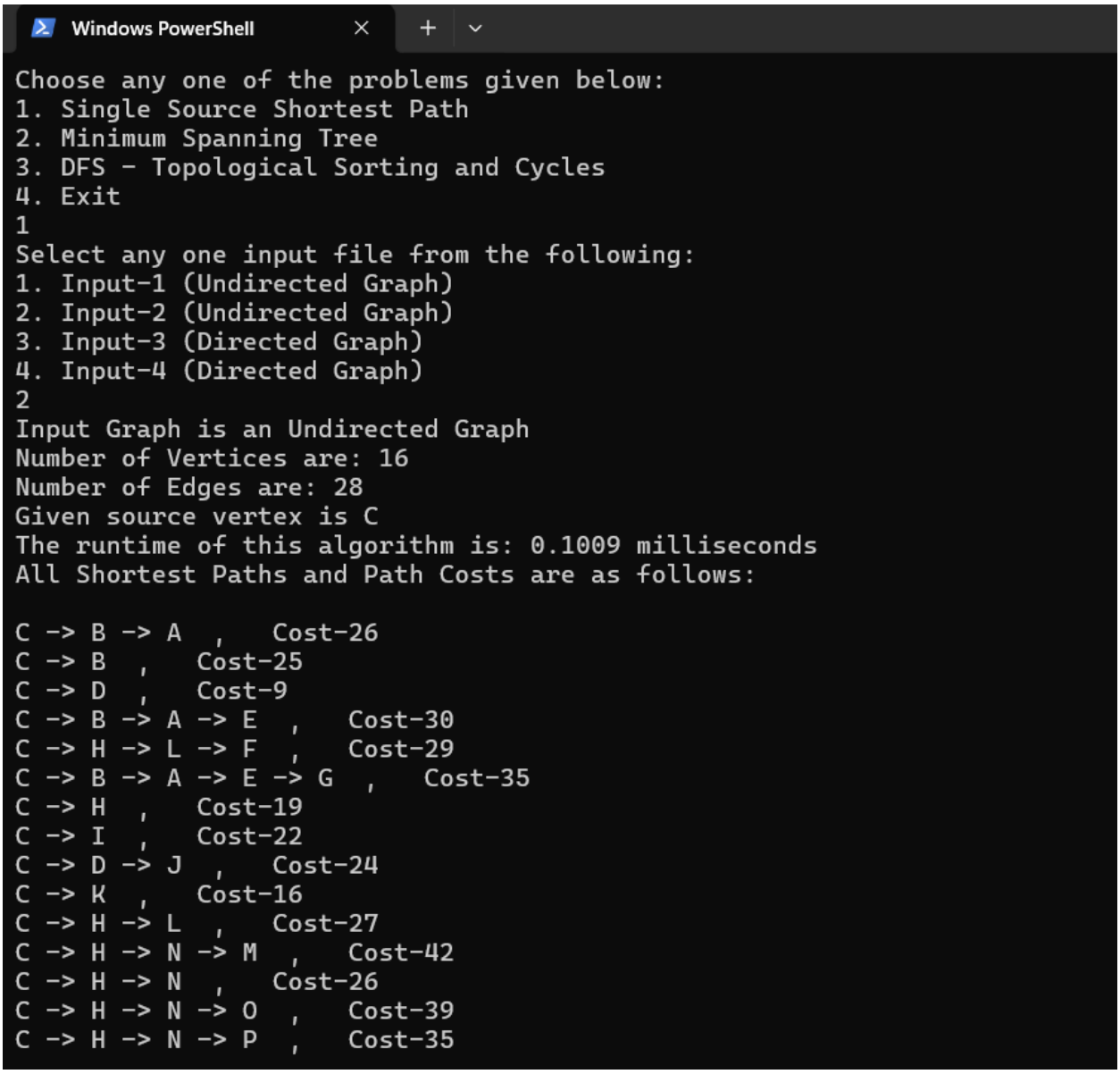
```
    // Adding source vertex to priority queue
    priorityQueue.add(new Node(sourceVertex, 0));

    // Dijkstra's Algorithm main loop
    while (!priorityQueue.isEmpty()) {
        int a = priorityQueue.remove().currentNode;
        cloud.add(a);
        for (Node neighbor : ipGraph.get(a)) {
            int b = neighbor.currentNode;
            if (!cloud.contains(b)) {
                int c = neighbor.currentWeight;

                // Update overall cost and parent if a shorter path is found
                if (overallCost[a] + c < overallCost[b]) {
                    overallCost[b] = overallCost[a] + c;
                    currentParent[b] = a;
                    priorityQueue.add(new Node(b, overallCost[b]));
                }
            }
        }
    }
    // Get the end time
    long endTime = System.nanoTime();
    // Calculate and print the runtime
    double runtimeInMillis = (endTime - startTime) / 1_000_000.0; // Convert to
milliseconds
    System.out.println("The runtime of this algorithm is: " + runtimeInMillis + "
milliseconds");
}
    // Method to display shortest paths and path costs
    public void displayOp() {
        System.out.println("All Shortest Paths and Path Costs are as follows: ");
        for (int i = 0; i < noOfVertices; i++) {
            if(i != sourceVertex) {
                int vertex = i;
                String temp = "";

                // Constructing shortest path string
                while (currentParent[vertex] != -1) {
                    temp = " -> " + VERTEX_INDEX_TRACKING.charAt(vertex) + temp;
                    vertex = currentParent[vertex];
                }

                // Displaying shortest path and path cost
```

```
                System.out.print("\n" +
VERTEX_INDEX_TRACKING.charAt(sourceVertex) + temp + "  ,   Cost-");
                System.out.print(overallCost[i]);
            }
        }
        System.out.println();
    }
}
//Node class to represent vertices and their weights
class Node implements Comparator<Node> {
    int currentNode, currentWeight;

    // Default constructor
    Node() {}
    Node(int currentNode, int currentWeight) {
        this.currentNode = currentNode;
        this.currentWeight = currentWeight;
    }

    // Compare method to compare nodes based on their weights
    public int compare(Node n1, Node n2) {
        if (n1.currentWeight < n2.currentWeight)
            return -1;
        if (n1.currentWeight > n2.currentWeight)
            return 1;
        return 0;
    }
}
```

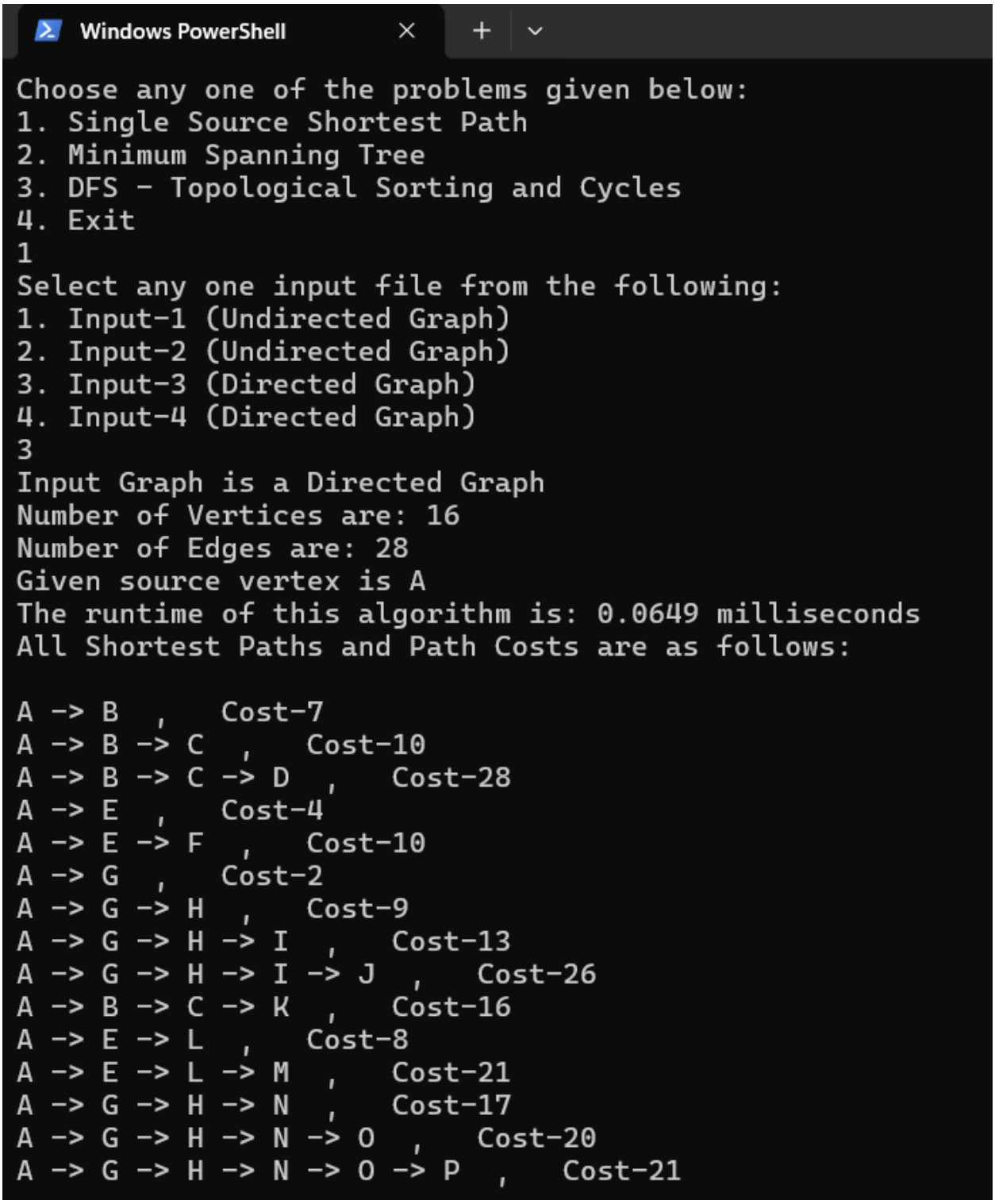
Output: Input Graph 1:



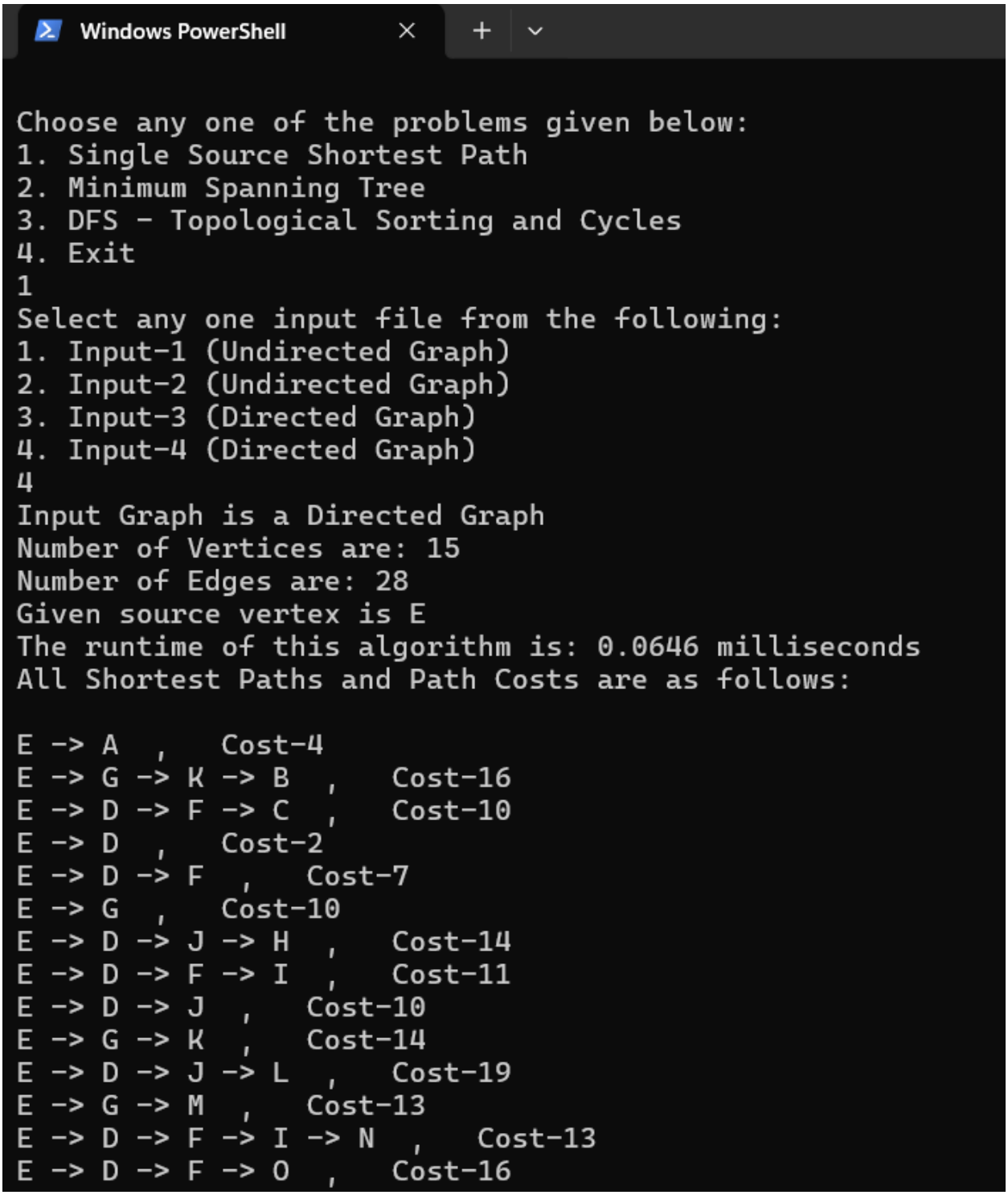
Input Graph 2:



Input Graph 3:



Input Graph 4:



Project 2: Minimum Spanning Tree Algorithm

Minimum Spanning Tree (MST):

For a connected, weighted, undirected graph, a Minimum Spanning Tree (MST) is a tree in which the total of the edge weights is as small as possible. In essence, it's a tree that makes sure the overall weight of all of its edges is as low as possible and connects all of its vertices without creating any cycles.

Kruskal's Algorithm:

The greedy method of Kruskal seeks to find an MST for a given weighted, undirected graph. When the algorithm first starts, every node forms a different tree. After that, it methodically chooses the least expensive edge among the possibilities and adds it to the growing tree forest. This process keeps going until the forest unites into a single tree that symbolizes the MST.

Essentially, Kruskal's algorithm examines the edges of the graph and sorts them according to weight. It always selects the edge with the least amount of weight and incorporates it into the expanding tree. This technique of iteration continues until every vertex is contained within the MST.

Pseudo code:

```
KRUSKAL(G, w)
    A = EmptySet
    // Create a disjoint set for each vertex in the graph
    for each vertex v in G.V
        CREATE_SET(v)
    // Sort the graph's edges by weight in ascending order
    SORT edges of G.E based on weight w
    // Traverse edges in ascending order of weight
    for each edge (u, v) in G.E
        if SET_FIND(u) is not the same as SET_FIND(v)
            A = A combined with {(u, v)}
            UNION SET(u, v)
    return A
```

Analysis of Runtime Complexity:

Given a graph G with 'n' vertices and 'm' edges, the time complexity breakdown for Kruskal's algorithm is as follows:

- CREATE_SET (v) is only executed once for each vertex and takes O(n) time to complete.
- The algorithm's edge sorting phase uses a sorting approach, which results in a temporal complexity of O(m log m).
- The SET_FIND () procedure, which is critical for identifying set membership, runs in O(m) time.
- The UNION SET (u, v) function is repeated n times because a spanning tree with 'n' vertices has 'n-1' edges. Thus, every UNION SET (u, v) action takes O(n) time.

To summarize, Kruskal's approach has an overall runtime complexity of O(m logm).

Used Data Structures and Their Effect on Runtime:

1. Edge Class (Edge[])

a. The Edge class represents individual edges in the graph, with properties such as currentSource, currentDestination, and currentWeight.
b. A collection of Edge instances, represented as an array (edges), contains all graph edges.
c. Leveraging the Comparable interface within the Edge class enables edge comparison based on weight, which is critical for Kruskal's algorithm's sorting step.

2. SubSet Class (SubSet[]):

a. The SubSet class defines subsets required for the union-find (or disjoint-set) data structure, which includes attributes like parent and rank.
b. The subSets array contains instances of the SubSet class, which manages the disjoint sets corresponding to vertices.
c. The parent field identifies the set's representative element within each subset, whereas rank optimizes the union operation by keeping track of tree height.

3. Arrays and Sorting (Arrays.sort()):

a. The Arrays.sort() function makes the critical edge sorting portion of Kruskal's algorithm easier, grouping edges according to weight.
b. This sorting procedure has a temporal complexity of O(E log E), where E is the edge count.

4. **Union-Find Operations (union(), find()):**
    a. The union-find data structure is implemented using dedicated unionMST() and findMST() methods.
    b. The unionMST() procedure combines sets, whereas findMST() identifies the set representative for any given element.

Implementation:

```
//Nishant Acharekar - 801363902 & Ashutosh Zawar (801366153)

package Proj2AlgoDs;

import java.io.File;
import java.io.FileReader;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

public class MinimumSpanningTreeAlgorithm {
    // File reading components
    private Scanner scanFile;
    private FileReader readFile;
    private String currentData;

    // Graph properties
    private int noOfVertices, noOfEdges, currentIndex;
    private Edge[] currentAnswer, currentEdges;

    // Constant for vertex indexing
    private final String VERTEX_INDEX_TRACKING = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Edge class to represent graph edges
    static class Edge implements Comparable<Edge> {
        int currentSource, currentDestination, currentWeight;
        public int compareTo(Edge edge) {
            return this.currentWeight - edge.currentWeight;
        }
    }

    // Subset class for union-find operations
    static class SubSet {
        int currentParent, currentRank;
    }

    // Main algorithm execution
```

```
    public void executeMainAlgorithm() {

        // Start measuring the runtime
        long startTime = System.nanoTime();

        // Initialize answer array
        currentAnswer = new Edge[noOfVertices];
        currentIndex = 0;

        for (int j = 0; j < noOfVertices; j++)
            currentAnswer[j] = new Edge();
        // Sort edges based on weight
        Arrays.sort(currentEdges);

        // Initialize subsets for union-find
        SubSet[] subSets = new SubSet[noOfVertices];
        for (int j = 0; j < noOfVertices; j++)
            subSets[j] = new SubSet();

        // Initialize subsets with parent and rank
        for(int i = 0; i < noOfVertices; i++) {
            subSets[i].currentParent = i;
            subSets[i].currentRank = 0;
        }

        // Kruskal's algorithm to find MST
        int j = 0;
        while (currentIndex < noOfVertices - 1) {
            Edge nextEdge;
            nextEdge = currentEdges[j++];
            int x = findMST(subSets, nextEdge.currentSource);
            int y = findMST(subSets, nextEdge.currentDestination);
            if (x != y) {
                currentAnswer[currentIndex++] = nextEdge;
                unionMST(subSets, x, y);
            }
        }
        // End measuring the runtime
        long endTime = System.nanoTime();

        // Calculate and print the runtime
        double runtimeInMillis = (endTime - startTime) / 1_000_000.0; // Convert
to milliseconds
        System.out.println("The runtime of this algorithm is: " + runtimeInMillis
+ " milliseconds");
```

```
}

// Read input file to populate graph data
public void readIpFile(String file) throws Exception {
    File inputFile = new File(file);
    readFile = new FileReader(inputFile);
    scanFile = new Scanner(readFile);

    // Read number of vertices and edges
    noOfVertices = scanFile.nextInt();
    noOfEdges = scanFile.nextInt();
    scanFile.next();
    System.out.println("Number of Vertices are: " + noOfVertices);
    System.out.println("Number of Edges are: " + noOfEdges);

 // Initialize currentEdges array
    currentEdges = new Edge[noOfEdges];
    for (int i = 0; i < noOfEdges; i++)
        currentEdges[i] = new Edge();

 // Populate currentEdges with edge data
    for (int i = 0; i < noOfEdges; i++) {
        int src = VERTEX_INDEX_TRACKING.indexOf(scanFile.next().charAt(0));
        int dest = VERTEX_INDEX_TRACKING.indexOf(scanFile.next().charAt(0));
        int currentWeight = scanFile.nextInt();
        currentEdges[i].currentSource = src;
        currentEdges[i].currentDestination = dest;
        currentEdges[i].currentWeight = currentWeight;
    }
}

// Union operation in union-find
public void unionMST(SubSet[] subSets, int x, int y) {
    int X = findMST(subSets, x);
    int Y = findMST(subSets, y);
    if(subSets[X].currentRank > subSets[Y].currentRank)
        subSets[Y].currentParent = X;
    else if (subSets[X].currentRank < subSets[Y].currentRank)
        subSets[X].currentParent = Y;
    else {
        subSets[Y].currentParent = X;
        subSets[X].currentRank++;
    }
}
```

```
    // Find operation in union-find with path compression
    public int findMST(SubSet[] subSets, int i) {
        if(subSets[i].currentParent != i)
            subSets[i].currentParent = findMST(subSets,
subSets[i].currentParent);
        return subSets[i].currentParent;
    }

    // Displaying the MST edges and total cost
    public void displayOp() {
        System.out.println("Minimum Spanning Tree- ");
        int totalCost = 0;
        for(int i = 0; i < currentIndex; i++) {
            System.out.println(VERTEX_INDEX_TRACKING.charAt(currentAnswer[i].curr
entSource) + " -> " +
VERTEX_INDEX_TRACKING.charAt(currentAnswer[i].currentDestination) + "  ,   Cost-
" + currentAnswer[i].currentWeight);
            totalCost += currentAnswer[i].currentWeight;
        }
        System.out.println("Total Cost of the Minimum Spanning Tree is: " +
totalCost);
    }
}
```

Output: Input Graph 1:

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
2
Select any one input file from the following:
1. Input-1 (Undirected Graph)
2. Input-2 (Undirected Graph)
3. Input-3 (Undirected Graph)
4. Input-4 (Undirected Graph)
1
Number of Vertices are: 16
Number of Edges are: 25
The runtime of this algorithm is: 1.3773 milliseconds
Minimum Spanning Tree-
A -> B  ,   Cost- 1
N -> O  ,   Cost- 1
O -> P  ,   Cost- 1
C -> E  ,   Cost- 6
F -> G  ,   Cost- 6
H -> I  ,   Cost- 7
D -> G  ,   Cost- 8
H -> J  ,   Cost- 9
I -> L  ,   Cost- 9
J -> K  ,   Cost- 9
E -> I  ,   Cost- 16
F -> H  ,   Cost- 16
L -> M  ,   Cost- 19
A -> D  ,   Cost- 25
A -> P  ,   Cost- 25
Total Cost of the Minimum Spanning Tree is: 158
```

Input Graph 2:

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
2
Select any one input file from the following:
1. Input-1 (Undirected Graph)
2. Input-2 (Undirected Graph)
3. Input-3 (Undirected Graph)
4. Input-4 (Undirected Graph)
2
Number of Vertices are: 15
Number of Edges are: 26
The runtime of this algorithm is: 0.0357 milliseconds
Minimum Spanning Tree-
A -> B  ,   Cost- 5
M -> N  ,   Cost- 5
C -> L  ,   Cost- 8
E -> J  ,   Cost- 8
J -> K  ,   Cost- 9
B -> C  ,   Cost- 12
A -> E  ,   Cost- 16
G -> H  ,   Cost- 16
F -> L  ,   Cost- 18
H -> I  ,   Cost- 23
I -> L  ,   Cost- 35
H -> M  ,   Cost- 37
B -> D  ,   Cost- 44
A -> O  ,   Cost- 69
Total Cost of the Minimum Spanning Tree is: 305
```

Input Graph 3:

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
2
Select any one input file from the following:
1. Input-1 (Undirected Graph)
2. Input-2 (Undirected Graph)
3. Input-3 (Undirected Graph)
4. Input-4 (Undirected Graph)
3
Number of Vertices are: 17
Number of Edges are: 27
The runtime of this algorithm is: 0.0354 milliseconds
Minimum Spanning Tree-
B -> D  ,   Cost- 3
H -> I  ,   Cost- 3
J -> N  ,   Cost- 3
B -> C  ,   Cost- 4
E -> F  ,   Cost- 4
I -> J  ,   Cost- 4
K -> P  ,   Cost- 4
L -> O  ,   Cost- 4
A -> B  ,   Cost- 5
E -> H  ,   Cost- 5
K -> L  ,   Cost- 5
M -> O  ,   Cost- 5
C -> E  ,   Cost- 7
K -> N  ,   Cost- 7
F -> G  ,   Cost- 9
M -> Q  ,   Cost- 9
Total Cost of the Minimum Spanning Tree is: 81
```

Input 4:

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
2
Select any one input file from the following:
1. Input-1 (Undirected Graph)
2. Input-2 (Undirected Graph)
3. Input-3 (Undirected Graph)
4. Input-4 (Undirected Graph)
4
Number of Vertices are: 16
Number of Edges are: 26
The runtime of this algorithm is: 0.0331 milliseconds
Minimum Spanning Tree-
B -> D  ,   Cost- 3
H -> I  ,   Cost- 3
J -> N  ,   Cost- 3
B -> C  ,   Cost- 4
E -> F  ,   Cost- 4
I -> J  ,   Cost- 4
K -> P  ,   Cost- 4
L -> O  ,   Cost- 4
A -> B  ,   Cost- 5
E -> H  ,   Cost- 5
K -> L  ,   Cost- 5
M -> O  ,   Cost- 5
C -> E  ,   Cost- 7
K -> N  ,   Cost- 7
F -> G  ,   Cost- 9
Total Cost of the Minimum Spanning Tree is: 72
```

Project 2: DFS - Topological Sorting and Cycles

Depth-First Search (DFS):

An essential algorithm for navigating or searching tree or graph data structures is called Depth-First Search (DFS). When traversing a graph, DFS begins from a predetermined source node and proceeds as far down each branch as it can go before turning around.

Topological Sorting:

In directed acyclic graphs (DAGs), where nodes (or vertices) are ordered in a way that respects the direction of the edges, topological sorting is an essential task. In other words, node u comes before node v in the topological ordering for each directed edge (u, v). For activities like dependency resolution and work scheduling, this ordering is useful.

DFS for Topological Sorting:

On DAGs, DFS provides an effective way to carry out topological sorting. The algorithm first does a DFS traversal, after which nodes are added to a stack in the order in which they finish processing. The graph's topological sorting is obtained by reversing the order of this stack.

Cycles in Graphs:

In a graph, a cycle is a closed path made up of vertices and edges where a vertex can be reached directly or indirectly from itself. A graph needs to be acyclic in order to be topologically sorted. When a cycle is present, topological sorting is not feasible.

Pseudo code:

DFS(G, v)

```
reset all data structures
for each vertex u in G.V
   if visited[u] == 0
      DFS-Visit(u)
```

DFS-Visit(u)

```
mark u as visited
push u to currentCycleStack
for each neighbor v of u in G
   if visited[v] == 0
      DFS-Visit(v)
   else if v is in currentCycleStack
      record cycle
pop u from currentCycleStack
prepend u to topologicalOrder
```

Analysis of Runtime Complexity:

1. **DFS(G, v):**
    a. The reset method initializes data structures, which requires O(V) time.
    b. The for loop iterates across each vertex, which takes O(V) time.
    c. The DFS-Visit(u) function is called within the loop, and it has its own level of complexity.

2. **DFS-Visit(u):**
    a. Marking a vertex as visited and moving it to the currentCycleStack requires O(1) time.
    b. The for loop iterates across each neighbor of u in O(E) time.
    c. DFS-Visit(v) is called within the loop, and it has its own complexity.
    d. Checking if v is in the current cycle stack takes O(1) time.
    e. It takes O(1) time to remove a vertex from the currentCycleStack and add it to the topologicalOrder.

Based on the foregoing study, the overall time complexity of DFS is O(V + E).

Used Data Structures and Their Effect on Runtime:

1. **Graph Representation (List<List<Node>> ipGraph):**
    a. A collection of adjacency lists for the graph.
    b. Allows for quick access to a vertex's neighbors.
    c. The size of this structure affects memory but has no substantial impact on runtime complexity.

2. **Mapping between Vertices and Characters:**
    a. HashMaps (vertexToCharMapping and charToVertexMapping) enable rapid lookups between characters and vertex indices.
    b. Vertex lookup based on character has O(1) time complexity, and vice versa.

3. **Visited Array (int[] visited):**
    a. An array that keeps track of visited vertices during DFS.
    b. Allows you to determine whether a vertex has been visited in O(1) time.

4. **Current Cycle Stack (Stack<Character> currentCycleStack):**
    a. A stack for keeping track of the current path during DFS.
    b. Enables cycle detection by verifying if a vertex is already on the stack in O(1) time.

5. **Topological Order (List<Character> topologicalOrder) and Cycles (List<List<Character>> cycles):**
    a. Lists for storing the topological order and discovered cycles.
    b. Adding to these lists requires O(1) time.

6. **LinkedList<Integer> path:**
    a. A linked list for keeping track of the path during DFS.
    b. It is used, but has no substantial impact on runtime complexity.

In summary, the data structures used facilitate efficient implementation of DFS with a runtime complexity of O(V + E).

Implementation:

```
//Nishant Acharekar - 801363902 & Ashutosh Zawar (801366153)
package Proj2AlgoDs;
import java.io.File;
import java.io.FileReader;
import java.util.*;
public class DFSAlgorithm {

    // File reading components listed below
    private FileReader readFile;
    private Scanner scanFile;

    // Constants
    private final String VERTEX_INDEX_TRACKING = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Below is the Graph representation
    private List<List<Node>> ipGraph;
    private int noOfVertices, noOfEdges;
    private boolean isDirectedGraph;

    // Mapping between vertices and characters
    private Map<Integer, Character> vertexToCharMapping = new HashMap<>();
    private Map<Character, Integer> charToVertexMapping = new HashMap<>();

    // Data for topological ordering and cycle detection
    private List<Character> topologicalOrder = new ArrayList<>();
    private List<List<Character>> cycles = new ArrayList<>();
    private Stack<Character> currentCycleStack = new Stack<>();

    // Array to track visitedVerticesTracking vertices during DFS
    private int[] visited;

    // Reset all data structures
    public void reset() {
        visited = new int[noOfVertices];
        topologicalOrder.clear();
        cycles.clear();
        currentCycleStack.clear();
    }
    // Execute the DFS algorithm

    public void executeMainAlgorithm() {
    visited = new int[noOfVertices];
    boolean[] inCycle = new boolean[noOfVertices];
```

```
    LinkedList<Integer> path = new LinkedList<>();
    // Start measuring the runtime
    long startTime = System.nanoTime();
       for (int i = 0; i < noOfVertices; i++) {
           if (visited[i] == 0) {
                dfs1(i, visited, new LinkedList<>());
            }
        }

         // End measuring the runtime
         long endTime = System.nanoTime();

    //  //Displaying the results
    //     if (cycles.isEmpty()) {
    //         System.out.println("Topological Sorting Sequence (Directed Acyclic
Graph):");
    //         for (int i = topologicalOrder.size() - 1; i >= 0; i--) {
    //             System.out.print(topologicalOrder.get(i) + " ");
    //         }
    //         System.out.println();
    //     } else {
    //         System.out.println("The directed graph contains cycles (Directed
Cyclic Graph):");
    //         for (List<Character> cycle : cycles) {
    //             // Add the starting vertex at the end of the cycle
    //             List<Character> extendedCycle = new ArrayList<>(cycle);
    //              extendedCycle.add(cycle.get(1));
    //             System.out.println("Cycle: " + cycle.toString() + ", Length: "
+ cycle.size());
    //         }
    //     }
    // Displaying the results
if (cycles.isEmpty()) {
    System.out.println("Topological Sorting Sequence (Directed Acyclic Graph):");
    for (int i = topologicalOrder.size() - 1; i >= 0; i--) {
        System.out.print(topologicalOrder.get(i) + " ");
    }
    System.out.println();
} else {
    System.out.println("The directed graph contains cycles (Directed Cyclic
Graph):");
    for (List<Character> cycle : cycles) {
        // Add the starting vertex at the end of the cycle
        List<Character> extendedCycle = new ArrayList<>(cycle);
        extendedCycle.add(cycle.get(0));  // Add the starting vertex
```

```
        System.out.println("Cycle: " + extendedCycle.toString() + ", Length: " +
extendedCycle.size());
    }
}
        // Calculate and print the runtime
        double runtimeInMillis = (endTime - startTime) / 1_000_000.0; // Convert
to milliseconds
        System.out.println("The runtime of this algorithm is: " + runtimeInMillis
+ " milliseconds");
    }

    // Depth First Search traversal
    LinkedList<Integer> path = new LinkedList<Integer>();
    private boolean dfs1(int vertex, int[] inCycle, LinkedList<Character>
cyclePath) {
        currentCycleStack.push(VERTEX_INDEX_TRACKING.charAt(vertex));
        if(inCycle[vertex] == 1) {
            cycles.add(new LinkedList<>(cyclePath));
            return false;
        }
        if(inCycle[vertex] == 2) {
            return true;
        }
        cyclePath.add(VERTEX_INDEX_TRACKING.charAt(vertex));
        inCycle[vertex] = 1;
        for (Node neighbor : ipGraph.get(vertex)) {
            if (!dfs1(neighbor.currentNode, inCycle, cyclePath)) {
                return false;
            }
        }
        inCycle[vertex] = 2;
        path.add(vertex);
        topologicalOrder.add(VERTEX_INDEX_TRACKING.charAt(vertex));
        return true;
    }

    // Read input graph from a file
    public void readIpFile(String file) throws Exception {
        File inputFile = new File(file);
        readFile = new FileReader(inputFile);
        scanFile = new Scanner(readFile);

        // Read graph type and validate
        String data = scanFile.nextLine();
        String[] t = data.split(" ");
```

```
        if (t.length > 2 && t[2].equals("U")) {
            throw new IllegalArgumentException("Undirected graphs are not
supported for this problem. "
                    + "/n Please Run the program again with a Directed Graph as
Input!!!");
        }

        // Initialize graph properties
        isDirectedGraph = true;
        System.out.println("Input Graph is a Directed Graph");
        noOfVertices = Integer.parseInt(t[0]);
        noOfEdges = Integer.parseInt(t[1]);

        // Check if the graph meets the minimum requirements
        if(noOfVertices < 15 || noOfEdges < 25) {
            throw new IllegalArgumentException("The graph does not meet the
minimum requirements. Please provide a correct Graph!!!");
        }

        System.out.println("Number of Vertices are: " + noOfVertices);
        System.out.println("Number of Edges are: " + noOfEdges);

        // Initialize graph data structures
        ipGraph = new ArrayList<>();
        for (int i = 0; i < noOfVertices; i++) {
            ipGraph.add(new ArrayList<>());
            vertexToCharMapping.put(i, VERTEX_INDEX_TRACKING.charAt(i));
            charToVertexMapping.put(VERTEX_INDEX_TRACKING.charAt(i), i);
        }

        // Populate graph edges
        for (int i = 0; i < noOfEdges; i++) {
            data = scanFile.nextLine();
            t = data.split(" ");
            int x = VERTEX_INDEX_TRACKING.indexOf(t[0].charAt(0));
            int y = VERTEX_INDEX_TRACKING.indexOf(t[1].charAt(0));
            ipGraph.get(x).add(new Node(y, Integer.parseInt(t[2])));
            if (!isDirectedGraph) {
                ipGraph.get(y).add(new Node(x, Integer.parseInt(t[2])));
            }
        }

        // Read source vertex if provided
        if (scanFile.hasNext()) {
            data = scanFile.nextLine();
```

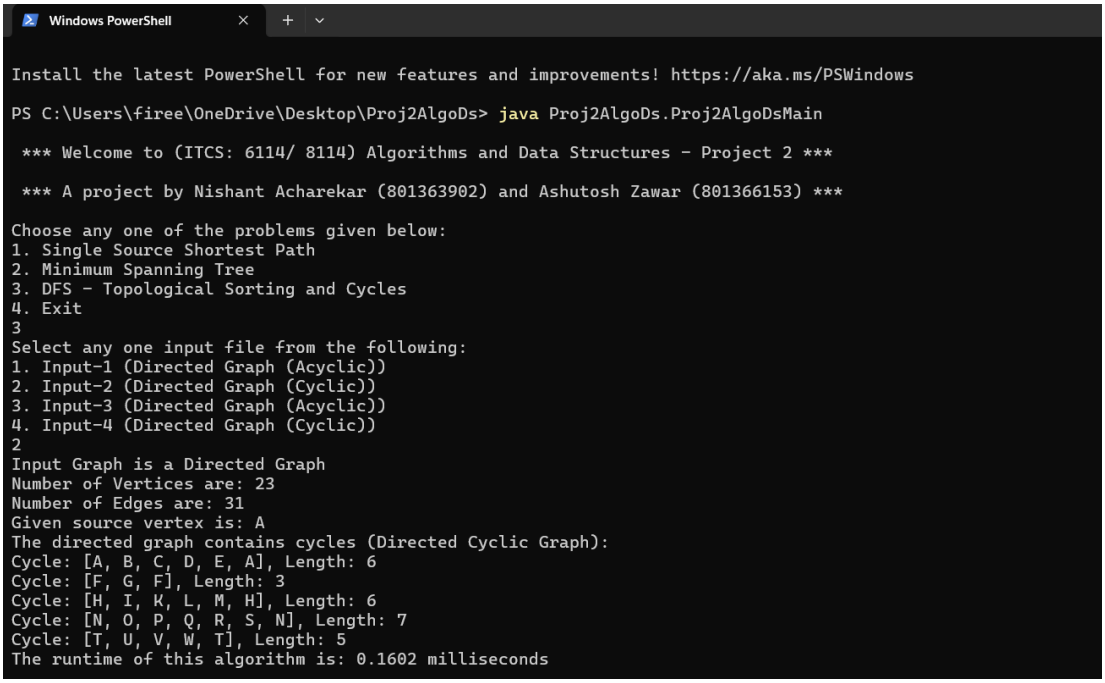
```
            char sourceVertex = data.charAt(0);
            int sourceVertexIndex = charToVertexMapping.get(sourceVertex);
            System.out.println("Given source vertex is: " + sourceVertex);
        } else {
            System.out.println("Source vertex is not provided. Please Provide a
source vertex in Input File!!!");
        }
    }

    // Displaying the output
    public void displayOp() {
        // Output is displayed within the executeMainAlgorithm() method
    }
    // Node class to represent graph vertices
    private static class Node {
        int currentNode, currentWeight;
        Node(int currentNode, int currentWeight) {
            this.currentNode = currentNode;
            this.currentWeight = currentWeight;
        }
    }
}
```

Output:

Input Graph 1: Directed Acyclic Graph

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
3
Select any one input file from the following:
1. Input-1 (Directed Graph (Acyclic))
2. Input-2 (Directed Graph (Cyclic))
3. Input-3 (Directed Graph (Acyclic))
4. Input-4 (Directed Graph (Cyclic))
1
Input Graph is a Directed Graph
Number of Vertices are: 16
Number of Edges are: 25
Given source vertex is: A
Topological Sorting Sequence (Directed Acyclic Graph):
A D C F I L O B G J M E H K P N
The runtime of this algorithm is: 0.1838 milliseconds
```

Input Graph 2: Directed Cyclic Graph



Input Graph 3: Directed Acyclic Graph

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
3
Select any one input file from the following:
1. Input-1 (Directed Graph (Acyclic))
2. Input-2 (Directed Graph (Cyclic))
3. Input-3 (Directed Graph (Acyclic))
4. Input-4 (Directed Graph (Cyclic))
3
Input Graph is a Directed Graph
Number of Vertices are: 16
Number of Edges are: 25
Given source vertex is: A
Topological Sorting Sequence (Directed Acyclic Graph):
A C B D E G K N F H L I M O P J
The runtime of this algorithm is: 0.0454 milliseconds
```

Input Graph 4: Directed Cyclic Graph

```
Choose any one of the problems given below:
1. Single Source Shortest Path
2. Minimum Spanning Tree
3. DFS - Topological Sorting and Cycles
4. Exit
3
Select any one input file from the following:
1. Input-1 (Directed Graph (Acyclic))
2. Input-2 (Directed Graph (Cyclic))
3. Input-3 (Directed Graph (Acyclic))
4. Input-4 (Directed Graph (Cyclic))
4
Input Graph is a Directed Graph
Number of Vertices are: 18
Number of Edges are: 26
Given source vertex is: A
The directed graph contains cycles (Directed Cyclic Graph):
Cycle: [A, B, C, A], Length: 4
Cycle: [D, E, F, D], Length: 4
Cycle: [G, H, I, J, G], Length: 5
Cycle: [K, L, M, N, O, P, Q, R, K], Length: 9
The runtime of this algorithm is: 0.0266 milliseconds
```