**Name:** Mustafa Baran Ercan **Section**: CMPE\_224 –SEC01  
**ID:** 28810555206 **Assignment**: / HW-2

**PA REPORT – 2**

**Problem Statement and Code Design**

For this part of the assignment, I designed two solutions for two different questions given to us. The first solution includes the Digraph algorithm, and the second solution includes the Edge Weighted Graph implementation. The first question is simply about a project where we are designing a special ride network for the Uber company. The tricky part of the first solution is to check whether the given Digraph can be kept in a tree structure.

The second solution involves farmlands, which need to be harvested by a company with a good trip path that has starting and ending indices. The program may consist of more than just one farmland (the test case) and different sizes of these farmlands. We should first implement this test case in a grid, then instantiate an edge weighted graph from that grid. Finally, we must turn that graph into an MST (Minimum Spanning Tree) and print the weight of this path.

A picture containing text, screenshot, font, line

Description automatically generatedSolution 1: Solution 2:

A picture containing text, screenshot, font, line

Description automatically generated

**Implementation and Functionality**

The first solution consists of Main, Digraph, District, Bag classes. In the main class, the program gets input from the user and creates the necessary Digraph and District objects. In the main class, the program also ensures whether the given inputs are valid. The District objects have very similar usage to the Node objects in Java. The District objects hold both the name of the entity and its id number. The Digraph class uses District objects as its vertices. Digraph class represents the directed graph data structure. It contains methods to add edges between vertices, iterate over vertices pointing from a given vertex, and check if the graph can be represented as a tree. The Bag object is a collection that is used to store a group of items, very similar to lists.

The second solution consists of Main, Edge, MST, EdgeWeightedGraph and Bag classes. In the main class, the program gets input from the user, initializes the necessary graph, edge, and MST objects. In the main class, the program also ensures whether the given inputs are valid. Edge objects represent the connections between vertices. They have a weight value, which is calculated by the XOR method. An edge weighted graph has edges instead of vertices. EWG is also used to find the MST, which spans all vertices but with the path with the minimum weight. The Bag object is a collection that is used to store a group of items, very similar to lists.

The program comprises four primary sub-modules for each solution, as demonstrated in the  
previous section. Details about each sub-module are defined in the next section:

**Solution 1:**

1. **Get Input:** This sub-module takes user input for the number of taxi pickups (V), the number of taxi rides (E), and input for the taxi rides, which are "from" and "to". This information is used to instantiate the Digraph and District objects.
2. **District Objects:** District objects contain information such as name and id which are used to create and connect vertices in the directed graph.
3. **Instantiate Digraph:** This sub-module is the instantiation of the Directed Graph implementation that models the taxi ride network. It contains information about the vertices and edges in the graph and provides methods to add edges, iterate over vertices, and check if the graph can be represented as a tree or has a cycle.
4. **Connection-Tree:** After creating all taxi rides, we print the adjacency list for each district (vertex) in the graph. Then we check whether the graph can be represented as a tree. In Java, you can determine if a directed graph can be represented as a tree by checking if it is a connected acyclic graph.
5. **Print the Result:** This sub-module is the part where the program prints out the desired outputs on the screen.

**Solution 2:**

1. **Get Input:** This sub-module takes user input for the number of test cases, dimensions and values of each grid, and the starting and ending points for each test case. This information is used to instantiate the Edge, EWG and MST objects.
2. **Create Grid:** Firstly, the program transfers the input information to a 2D array. The reason we do this is to use this grid to find the edges and calculate the XOR weights of these edges.
3. **Instantiate Edge-Weighted Graph:** This sub-module is the instantiation of the Edge-Weighted Graph implementation. It contains edges in between two vertices and a weight value for each edge.
4. **MST**: MST is used to find the most efficient way to connect all the vertices in a graph. In our implementation, the aim is to find the weight of the path that starts and ends at the vertices we want.
5. **Print the Result:** This sub-module is the part where the program prints out the desired outputs on the screen.

**Testing and Graphs**

There are three different test cases for each of the two questions on the VPL. Both of my solutions for this homework successfully passed all six test cases. In addition, I tried to foresee the possible problems my code could face and solve them. I sought to avoid potential exceptions my code could throw and tried to solve them beforehand.

            Even though my second implementation for the solution of the second question calculates and returns the expected answer, it has a minor issue that is in fact not affecting the answer of my program. Throughout the question, we are expected to calculate the weight of the MST, whose starting and ending points are pre-defined. My code was successfully implemented up until there, including the starting index. Yet, my method returns the correct output without specifying the ending index. This was caused by the general structure of the MST algorithm and the formula we use

() to calculate the weight of the MST. Since we can only visit any edge a maximum of two times, the result will not be affected because of the ceiling operation we do in the formula above.

**Final Assessments**

While creating my program, I was able to successfully implement the directed graph, edge-weighted graph, and their methods into my code. It was useful for me to understand the mentality of how directed graphs and edge-weighted graphs work. For me, the most challenging part of this assignment was trying to figure out a way to arrange the method that returns the weight of the MST with the desired starting and ending indices. Overall, the program demonstrates a strong understanding of the concepts behind these data structures and how they can be used to model and solve real-world problems.