**Subject: Data Structures and Algorithm Design**

**Assignment 2: Problem Statement 14**

**Group: 166**

The design of the application is split across the required class, its functions and the Main program as explained below. The **error handling** and the **time complexity** is also explained at the bottom.

**High Level Design**

The program follows Dijkstra’s Shortest Path Algorithm to find the shortest route from house to both the pharmacies. The choice of Dijkstra’s algorithm is based on the fact that the weights are positive and graph is undirected. The graph itself is created as a nested list following the adjacency matrix format to store weights between two vertices.

**Class PharmacyRun:** This is the master class that contains all the functions called in the main program

**Function \_\_init\_\_**

This function does the following tasks:

* Take the pointer to the input and output file and reads through the input file.
* The graph details are extracted by splitting input file lines that contains “\”, splitting the string based on it and removing any white spaces. It is taken care that even if the vertex “name” is not a single letter, it is properly stored, and path found
* List of unique vertices is added to a list which is then sorted. This list contains only vertices and not edges and will serve as a means to check the position of a vertex in sorted order. Accordingly, the sorted position serves as the index of a vertex while creating adjacency matrix of the graph
* Checks for missing or invalid inputs and write error message to output file if either one of the inputs (house, pharmacy) is missing or one of them is not in the graph. In this situation, the function returns sets an invalid variable and program is terminated in main branch
* If no error condition is encountered, an empty graph (as nested list) is created with NxN size where N is the number of vertices in the graph

**Function createGraphEntry**

This function takes two vertices and the weight between them to create individual entries in the list “graph” that represents the adjacency matrix of the graph. The sorted order of the vertices is taken as their index position in this matrix (e.g. if vertex v1 is at index position 3 and v2 at index position 5 in sorted order, matrix entry will be created at position [3,5] with the weight)

**Functions writeResult and writeResultMultiple**

These functions write the final shortest path route and the number of containment zones encountered to the output file. The function writeResultMultiple is created to cover the case where the shortest path to both the pharmacies have equal number of containment zones. These functions work as following:

* The shortest path tree is passed to the function in the variable “path”.
* To extract the actual path, this tree is traversed starting from leaf node which is the destination pharmacy not passed in variable “dest”. Starting with destination, each parent node is pushed to a list called “stack”. Other sibling nodes are skipped by setting last\_node as the parent.
* The list named “stack” is then iterated in reverse order and individual nodes written to output file.

**Functions minWeight and shortestPath**

These functions implements the Dijkstra’s algorithm which works in following way:

* Create a list called ‘weights’ that contains edge weights for vertices being traversed. Initialize all weights to maxsize except for weight at index 0 which represents source node
* Create a list called “spt” that is intended to contain the shortest path tree of the given graph. Initialize all elements of this list to False. As the individual vertices are reached their respective position will be set to True indicating they are part of the shortest path tree
* The shortest path itself is stored in a list called shortestPath
* Starting with source node, individual nodes are traversed each time checking which of the adjacent nodes have minimum weight.
* The node with minimum weight is added to spt and in next iteration all its adjacent node “actual” weights are updated in list “weights” replacing maxsize
* Once destination node is reach, variable path\_found is set to 1 and loop is broken. Or if path is not found after traversing whole graph, then variable path\_found remains at 0 and appropriate message written to output file

**Main Program**

The main program has following steps:

1. Open input and output file for reading and writing respectively.
2. Create object of class PharmacyRun by passing the file pointers.
3. If there is missing or incorrect input, appropriate messages are written, graph is not created and variable invalid is set to 1 by the \_\_init\_\_ function. In this case program is terminated.
4. Program then reopens the input file to read individual vertices and weights between them to create adjacency matrix. House and Pharmacy nodes and their respective indexes are also determined
5. Shortest path is calculated by calling function shortestPath between house - > pharmacy 1 and house -> pharmacy 2
6. If path is not found to either or both the pharmacies appropriate messages are written.
7. Otherwise the distances are compared and the path which is shorter among the two is written to output file.
8. If both pharmacies are equi-distance (same no of containment zones), then both paths are written to output file
9. Input and output files are closed

**Error Handling**

The program handles following situations due to potential incorrect data entry:

1. Either the house or the pharmacies are not specified in input file.
2. Specified house node or the pharmacy nodes are not in the graph.
3. Path to one or both the pharmacies not found starting from the house. If path is available to at least one pharmacy that is sent to the output file.

**Special Cases Handling**

The following special conditions are taken care of by the program:

1. Vertices if specified in more than 1 letter is taken care of
2. If path to both pharmacies have same number of containment zones, both paths are written.

**Time Complexity**

For representing time complexity in asymptotic notation, let’s assume V to represent number of vertices and E to represent number of edges.

* Reading the input file and creating adjacency matrix for the graph in functions \_\_init\_\_ and createGraphEntry will be O(E) as it has to just go through edge once
* Functions to write results will also be O(E) as in worst case the shortest path will span all the edges
* Function minWeight will be O(V) as in worst case it goes through all the Vertices to find the one with minimum weight
* Function shortestPath has an outer for loop that will have max V iterations and it has a call to minWeight as well as another for loop both of which can go for max V iterations. So time complexity will be O(V\*(V+V)) or O(V2). In a tight bound evaluation, the function minWeight as well as the inner for loop in function shortestPath, will not execute the enclosed statements for all the V iterations. And so in Theta notation, performance will be much better than O(V2) depending upon the layout of the graph.

**Main Program Time Complexity**:

Main program just calls the constructor of PharmacyRun class, createGraphEntry function and 2 calls to function shortestPath. Function to write shortest path (writeResult and writeMultipleResult) are also called once which in worst case will have complexity of O(E). So overall complexity will be O(V2 + E).