CSCI 3901: Assignment 3

Submitted by Nadipineni Hemanth Kumar [B00899473]

Date: 25-10-2021

Under the supervision of: Dr. Michael McAllister

Documentation:

Overview of The Assignment:

The overview of the problem is creating a travel network and optimizing the travel plan during covid-19 with multiple cities having different testing requirements. The solution can be achieved using Graphs. In this solution, Dijkstra’s algorithm, and adjacency list to store graphs were used.

File and External Data:

1) Main.java – The main file to call the methods and take inputs. This file is required for testing internally. This file was created to test multiple inputs manually.

2) TravelAssistant.java – The file with methods such as addCity(), addFlight(), addTrain(), planTrip(). The data from the test cases is passed as arguments and processed to generate a travel network and then finding the shortest distance using Dijkstra’s algorithm. This file had to be created to implement the methods mentioned above.

3) CityVertex.java – A class to store data related to a city such as cityName, testRequired, timeToTest, nightlyHotelCost. The purpose of creating this file is to have the above mentioned data and this class can be instantiated.

4) FlightHope.java – A class to store data related to a flight such as startCity, destinationCity, flightTime, flightCost. The purpose of creating this file is to have the above mentioned data and this class can be instantiated.

5) TrainHop.java – A class to store data related to a train such as startCity, destinationCity, trainTime, trainCost. The purpose of creating this file is to have the above mentioned data and this class can be instantiated.

6) Vertex.java – A class to store data related to vertex while creating a travel network. This class also stores the list of adjacent vertices.

7) AdjVertex.java – A class to store data related to an adjacent vertex while creating a travel network. This file was created so that adjacency vertices can be instantiated for several vertices.

8) listToReturn.java – A class to store the details of the travel plan list to be returned in the specified way. This class file was created to have data of list to be returned finally.

Data Structures and Their Relations to Each Other:

Used ArrayLists to store flights list, trains list, cities list and a list to store only cities (used to check whether a flight is in between existing cities).

Used ArrayLists to store Vertices, vistiedVertices while calculating shortest distance.

Used Sets to store visited vertices prevent visiting the same vertex again and again.

Used Map to keep distances of each adjacent vertex from a vertex. Initially it was initialized to an infinity distance and then changed according to the traversal.

Used Lists, Integers for other functionalities.

Efficiency of the data structures used: As we are using List as abstract data type, not like using ArrayList concrete data type, they are very efficient in scaling and storing the data as per the need. Also, I used sets to validate duplicate visits to the vertices. And a map to store distances for each vertex which are efficient by usage. All the variables inside the classes are listed private to avoid illegal data modification.

Key Algorithms and Design Elements:

Dijkstra’s Algorithm was used to find the shortest path between the starting city and destination city with adaptions as mentioned in the teams’ channel and class.

Adjacency Lists were used to store the travel network (Graph).

Class attributes were declared private in all classes.

Setter and Getters were used to access private attributes.

Static variable declarations were used when necessary.

Final variable declaration was used when required.

Enhanced for loops, ternary operators were used to reduce code redundancy.

Constructors were used to encapsulate data inside a class.

Variables were declared appropriately such as global and local to method as per the requirement.

Methods for Travel Network generation[travelNetwork()] and finding shortest path [dijkstra()] are separated from planTrip to increase readability.

Efficiency of the algorithm used: In the worst case, assuming that the graph is dense i.e,fully connected, it traverse through each vertex for finding shortest path of each vertex and also it has to traverse through the each edge for each vertex. So, the complexity can be O(ElogV) + O(VlogV). That gives us the final complexity as O(logV\*(E+V)). In normal conditions where E>>>V, we can ignore V, so the complexity can come to O(ElogV). As we are using adjacency list instead of adjacency matrix, this is much efficient.

Limitations:

There are limitations when negative importance is given as Dijkstra Algorithm doesn’t work for negative weight edges.

Travel modes are limited to fly and train only.

Assumptions:

All assumptions mentioned in the assignment pdf were considered.

Any cost cannot be zero such as nightlyHotelCost, flightCost and hotelCost. In other way, I didn’t consider 100% OFF or any other benefits.

References:

<https://www.youtube.com/watch?v=gXgEDyodOJU>

<https://www.youtube.com/watch?v=GazC3A4OQTE>

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

https://stackabuse.com/dijkstras-algorithm-in-python/

Note: Had a discussion with friend about the program, but code was never shared or showed.

List of Test Cases:

Input Validation (Should not crash for bad input data)–

addCity():

* Null value passed as the cityName
* Empty string passed as cityName
* testRequired is non-boolean
* nightlyHotelCost is less than 1

addFlight():

* Null value passed as the startCity
* Empty string for the startCity
* Null value passed as the destinationCity
* Empty string for the destinationCity
* 0 passed as flightTime
* flightTime is negative
* flightCost is less than 1

addTrain():

* Null value passed as the startCity
* Empty string for the startCity
* Null value passed as the destinationCity
* Empty string for the destinationCity
* 0 passed as trainTime
* trainTime is negative
* trainCost is less than 1

planTrip():

* Null passed as the startCity
* Empty string passed for the startCity
* Null passed as the destinationCity
* Empty string passed for the destinationCity
* Non-boolean value passed for isVaccinated
* Negative value for costImportace
* Negative value for travelTimeImportance
* Negative value for travelHopImportance

Boundary Cases (tests at the edge of inputs) –

addCity():

* 1 character cityName
* testRequired is true
* testRequired is false
* nightlyHotelCosts is 1

addFlight():

* 1 character startCity
* 1 character destinationCity
* flightCost is 1

addTrain():

* 1 character startCity
* 1 character destinationCity
* trainCost is 1

planTrip():

* 1 character startCity
* 1 character destinationCity
* isVaccinated is true
* isVaccinated is false
* costImportance is 0
* travelImportance is 0
* travelHopImportance is 0
* Plan a trip when only two cities are there and only 1 flight between them
* Plan a trip when only two cities are there and only 1 train between them

Control Flow Cases (tests of the core operations) –

addCity():

* Create a city when there are no cities
* Create a city when there is 1 city already created
* Create a city when there are many cities created
* Unique cityName is passed
* Non-unique (Duplicate) cityName is passed
* timeToTest is greater than 0
* timeToTest is negative
* timeToTest is zero (same day result)
* nightlyHotelCosts is greater than 0

addFlight():

* Add a flight when there are no flights
* Add a flight when there is 1 flight already added
* Add a flight when there are many flights added
* A flight already exists between the start city and destination city
* Add a flight when there is a train already exists between the same cities
* Flight between the same city (source==destination)
* Value of flightTime is greater than or equal to 1
* Value of flightCost is greater than or equal to 1

addTrain():

* Add a train when there are no trains
* Add a train when there is 1 train already added
* Add a train when there are many trains added
* A train already exists between the start city and destination city
* Add a train when there is a flight already exists between the same cities
* Train between the same city (start city and destination)
* Value of trainTime is greater than or equal to 1
* Value of trainCost is greater than or equal to 1

planTrip():

* Plan a trip with valid startCity and destinationCity with no flight routes between them
* Plan a trip with valid startCity and destinationCity and only 1 flight route between them
* Plan a trip with valid startCity and destinationCity and many flight routes defined
* Plan a trip with valid startCity and destinationCity with no train routes between them
* Plan a trip with valid startCity and destinationCity and only 1 train route between them
* Plan a trip with valid startCity and destinationCity and many train routes defined
* Plan a trip with invalid startCity or destinationCity or both when there is one city defined
* Plan a trip with invalid startCity or destinationCity or both when there are many cities defined
* Plan a trip when person is Unvaccinated and test is not available at any city
* Plan a trip when person is Unvaccinated and test is available only at start city
* Plan a trip when person is Unvaccinated and test is available only at destination city
* Plan a trip when person is Vaccinated irrespective or test availability
* Plan a trip with all costImportance, travelTimeImportance, travelHopImportance are in the same proportion.
* Plan a trip with only costImportance being non-zero and others being zero.
* Plan a trip with only travelImportance being non-zero and others being zero.
* Plan a trip with only travelHopImportance being non-zero and others being zero.
* Plan a trip with costImportance and travelImportance being non-zero and travelHopImportance being zero.
* Plan a trip with travelImportance and travelHopImportance being non-zero and costImportance being zero.
* Plan a trip with travelHopImportance and costImportance being non-zero and travelImportance being zero.

Data Flow Cases (tests around the order in which things are done) –

planTrip():

* Plan a trip from valid startCity to destinationCity with a direct flight
* Plan a trip from valid startCity to destinationCity with indirect (or multiple) flights
* Plan a trip from valid startCity to destinationCity with a direct train
* Plan a trip from valid startCity to destinationCity with indirect trains
* Plan a trip with valid startCity and destinationCity when a test is not required
* Plan a trip with valid startCity and destinationCity when no test is required
* Plan a trip with valid startCity and destinationCity when the person is vaccinated

- Plan a trip with valid startCity and destinationCity when the person is Unvaccinated