CSCI 3901 Assignment 3

Name: Dhruvrajsinh Omkarsinh Vansia

ID: B00891415

Date: 29/10/2021

Overview:

This program shows the shortest possible path between start city and destination city regrading cost, time, hop and vaccination status. Flight and train can be added between any two cities with their cost and time. Also, program checks for the covid test facility in the city for the non-vaccinated travelers.

Files and external data:

There is total 10 files in the program.

1. Main.java – This file contains main method of the program.
2. TravelAssistant.java – This file contains four methods to add city, flight, train and to plan the trip.
3. City.java – City object is created in this file.
4. Graph.java – Create adjacency list using object of Node. (Add edge in the graph)
5. Edge.java – Create edge object with attributes: start city, destination city, cost, time, mode.
6. Node.java – To create an object for the adjacency list.
7. dijkstraAlgo.java - Find the shortest path of the graph.
8. Graphforalgo.java – Create adjacency list using object of Nodeforalgo.
9. Edgeforalgo.java - Create edge object with attributes: start city, destination city, total cost, mode.
10. Nodeforalgo.java – Create an object for adjacency list of Graphforalgo.java file.

Data structure and their relationship to each other:

There are following data structures used in the program.

1. LinkedHashMap<String, City> cityInfo
   It maps city name to the its objects, namely cityName, testRequired, timeToTest, nightlyHotelCost.

2. List<List<Node>> adjList
   It stores information regarding flights and trains.

3. List<String> cityListFinal
   Stores the city name in order.
4. List<String> finalPath

This list stores the shortest possible path between start city and destination city.

Key algorithm and design elements:

In TravelAssistant.java file:

1. boolean addCity(String cityName, boolean testRequired, int timeToTest, int nightlyHotelCost) throws IllegalArgumentException
   This method is used to add city in the LinkedhashMap cityInfo. First, it checks input parameters and if all parameters are acceptable then it returns true, false if city is already added in the map. Throws illegal argument exception if parameters are unacceptable.

2. boolean addFlight(String startCity, String destinationCity, int flightTime, int flightCost)) throws IllegalArgumentException
   This method is used to add fight. Throws illegal argument if city name is null or empty, flight time and flight cost is less than 1, and list is not added by addCity method. Return true if flight is added successfully, false if already there is flight between the cities.

3. boolean addTrain(String startCity, String destinationCity, int trainTime, int trainCost))throws IllegalArgumentException
   This method is used to add train. Throws illegal argument if city name is null or empty, flight time and train cost is less than 1, and list is not added by addCity method. Return true if train is added successfully, false if already there is train between the cities.

4. List<String> planTrip (String startCity, String destinationCity, boolean isVaccinated, intcostImportance, int travelTimeImportance, int travelHopImportance) throws IllegalArgumentException
   This method finds the shortest path between start city and destination city according to the traveller preference that is cost, time and hop. To search best possible path, it uses Dijkstra algorithm. During process of path search, vaccination status is also considered with covid rules of the particular city. It also calculates the cost of hotel stay if the traveller has to do covid test in the city. This method returns list of paths if best possible path is found otherwise empty list.

In Graph.java file:

1. Graph (List<Edges> edges, TravelAssistant ta)
   It creates adjacency list ‘adjList’. In the list, it stores the flight and train information and update it for the further use.

2. void printGraph (Graph graph)
   Display the ‘adjList’ list created by Graph().

In DijkstraAlfo.java:

1. List<String> algo(List<List<Node>> adjList, String sCity, String destiCity, boolean isVaccinated, int costImportance, int travelTimeImportance, int travelHopImportance, TravelAssistant ta)
   This method finds the shortest path between start city and destination city. First, if there are two paths available between the same city then select the path which has minimum cost and travel time. Using this, it creates new adjacency list. After that, the adjacency matrix is built from the adjacency list to ease the process of the finding the shorted path. Then, through Dijkstra algorithm it finds the shortest path. Finally, this method adds mode of travel in the list of the path.

2. Int minCost(int[] cost, boolean[] checkingCondition, int nOfCities)
   This method finds the index of the next possible vertex that has minimum cost using two array and total number of cities.

Test cases:

1. Input Validation:
   addCity()
   - Null passed as cityName
   - Empty passed as cityName
   - Zero passed as nightlyHotelCost
   - Negative passed as nightlyHotelCost

   addFlight()
   - Null passed as startCity and destinationCity
   - Empty passed as startCity and destinationCity
   - Zero passed as flightTime or flightCost
   - Negative value passed as flightTime or flightCost

   addTrain()
   - Null passed as startCity and destinationCity
   - empty passed as startCity and destinationCity
   - Zero passed as trainTime or trainCost
   - Negative value passed as trainTime or trainCost

   planTrip()
   - Null passed as startCity and destinationCity
   - Empty passed as startCity and destinationCity
   - Negative value passed as travelTimeImportance, travelHopImportance and costImportance

2. Boundary cases:
   addCity()
   - Only one character passed as cityName
   - nightlyHotelCost is 1

addFlight()

- Only one character passed as startCity and destinationCity
- Flight cost and flight time is 1

addTrain()

- Only one character passed as startCity and destinationCity
- Train cost and train time is 1

planTrip()

- Only one character passed as startCity and destinationCity
- costImportance, travelTimeImportance or travelHopImportance is 0

3. ControlFlow Test:

addCity()

- Add cityName that is already exist
- Add city when there is no city
- Add city when there is one city
- Add city when there are many cities
- Test time is more than 1
- Hotel Cost is more than 1

addFlight()

- Add flight when there is no city
- Add flight when there is one city
- Add flight when both starting city and destination city is added
- Add flight when starting city and destination city are not added.
- Add flight when starting city is same as destination city

addTrain()

- Add train when there is no city
- Add train when there is one city
- Add train when both starting city and destination city is added
- Add train when starting city and destination city are not added.
- Add train when starting city is same as destination city

planTrip()

- Plan trip when there is no city
- Plan trip when there is one city
- Plan trip when both starting city and destination city is added
- Plan trip when starting city and destination city are not added.
- Plan trip when starting city is same as destination city
- Plan trip when cost importance, hop importance and time importance are more than 1

4. Data Flow Cases

    planTrip()

    - Plan trip when cost importance, time importance and hop importance are non-negative and are equal
    - Plan trip when the passenger is vaccinated
    - Plan trip when the passenger is not vaccinated
    - Plan trip when cost importance is more than time importance
    - Plan trip when cost importance is more than hop importance
    - Plan trip when time importance is more than cost importance
    - Plan trip when time importance is more than hop importance
    - Plan trip when hop importance is more than time importance
    - Plan trip when hop importance is more than cost importance
    - Set true for isVaccinated
    - Set false for isVaccinated

Efficiency of your data structures and algorithm for given tasks:
To find the shortest possible path between start city and destination city, the program uses Dijkstras algorithm. Here, the time complexity of the Dijkstras algorithm is O(V^2) where V is vertex. Its time complexity can be reduced to O(E log(V)), where E is edge, by using priority queue.

Reference:

Dijkstras Algorithm
GeeksforGeeks. 2021. Dijsktra's algorithm. [online] Available at: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/?ref=lbp> [Accessed 25 October 2021].