Author: Shivangi Ajaykumar Bhatt
B00 # : B00863408
Subject Code: CSCI 3901
Date: Oct 29th, 2021
Assignment #: 3

Travel Assistant

Overview:
The program called Travel Assistant uses a graph and using Dijkstra's algorithm, finds the shortest path between given cities.

1. addCity(): adds a City to the graph .
2. addTrain(): adds a Train to graph between two cities.
3. addFlight(): adds a Flight to the graph between two cities.
4. planTrip(): Finds the shortest path between two cities based on the cost importance, hop importance and time importance

Some cities may require a negative covid test to enter. A traveller may be vaccinated or unvaccinated. An unvaccinated traveller can only enter a city if they have had their test before entering the city while a vaccinated traveller can enter any city without having to get tested.

Files:
The program uses four java files for the execution of the program:

1. TravelAssistant: It is the major class that performs actions like adding city to the graph, adding connecting flights or trains between two cities and calculating the most optimized path between two cities based on the priorities of traveller.
2. Graph: Graph is the supporting class for travelAssistant which stores all the major information provided by the user. For instance, it stores the hashset of all the cities added and it performs various methods like:
    - Calculating the shortest path between cities from a source city.
    - Add a City to the graph
    - Get the hashset of cities
3. City: City is a supporting class that stores all the valid information of a city. For instance, city name, is test required at a city, time to test, nightly hotel cost at a city, a hashmap of connecting city with the type of transport between the cities and a List of shortestPath for the city
4. Transport: Transport is also a supporting class that stores information about an available transport between two different cities, such as startCity name, destinationCity name, means of travel(fly or train), time of travel, and travel cost.

Data structures and their relations to each other

The program uses the following data structures:

1. HashSet<City> cities:
   A hashset that stores all the objects of cities added to the travelAssistant.
2. <Map<City,List<Transport>>connectingCities:
   Each city object stores the hashmap of connecting cities with the destination city as its key and a list of transport available between the city and destination as its value.
3. List<<Map<City,Transport>> shortestPath:
   Each city object uses the list of shortest paths of a type Hashmap with a city as key and transport between that city and its connecting city.

Assumptions

Following assumptions have been made for the program:

- Once an unvaccinated traveller gets a covid test at a city, the test result is always positive.
- The vaccinated traveller does not need to get tested before entering any city.
- If the test is available at the sourceCity, the unvaccinated traveller can get tested at the sourceCity and does not need to spend on nightly hotel cost.

Key algorithms

- Firstly, the addCity() method adds the cities to the graph as vertices.
- Then the available transport are added between the connecting cities as edges by addTrain() or addFlight() method
- For finding the most optimized path between startCity and destinationCity for the planTrip() method, Dijkstra's algorithm is used.
- Dijkstra's algorithm calculates the distance to all the cities from the given source city and hence provides the list of shortest path that one needs to travel on to get from source to destination.
- The inputs for finding the shortest path from the startCity are Graph graph, City startCity, City destinationCity, boolean isVaccinated, int costImportance, int travelTimeImportance and int travelHopImportance.
- The algorithm for calculating shortest path for all the cities from source city is calculated as follows:
    1. Set the distance of the source city to 0.
    2. Set the distance of all other cities to infinite.
    3. Initialize two hashSets: settledCities (settledCities stores the cities whose shortest path from the startCity are already calculated) and Unsettled cities (UnsettledCities stores the cities whose shortest path from the startCity are yet to be calculated).
    4. Add the startCity to unsettledCities.
    5. While the size of unsettledCities does not get 0,

- Find the city from the unsettledCities with lowestDistance, say currentCity
- Loop through all the connectingCities of the currentCity :
    1. calculate the distance of all the connectingCities from source.
    2. Simultaneously find the list of shortestPath for all the connectingCities.
    3. Simultaneously add the connectingCities to the set of unsettledCities.
- Add the currentCity to the set of settledCities.

6. Return the new updated graph that has a set of Cities with their list of ShortestPath and their distance from the startCity

- This new updatedGraph can be used to search for the shortestPath of the destinationCity.

Efficiency of algorithms and data Structures

I have utilized hashSet for storing the graph and also utilized two hashSets for finding the shortest path using Dijkstra's algorithm.

- The efficiency of hashSets is O(1) when it comes to accessing elements from the hashSet which is better than a List.

- Whereas, the efficiency of Dijkstra's algorithm in the finding the shortest path of all the nodes would be O(V^2*E) for my algorithm where V are the vertices(cities) and E are the edges(Transport). I believe that the efficiency of my program would O(V^2*E) because I loop through the vertices twice and then loop through the edges as well.

Test cases

For the given Travel Assistant system following would be the test cases for its given methods:

Input Validation

addCity()

- cityName is null or empty.
  Expected output: throw IllegalArgumentException.

addFlight()

- startCity is null or empty
- destinationCity is null or empty
  The expected output for all the above cases: throw IllegalArgumentException

addTrain()

- startCity is null or empty
- destinationCity is null or empty
  The expected output for all the above cases: throw IllegalArgumentException

planTrip()

- startCity is null or empty
- destinationCity is null or empty
- Following arguments are negative
  - costImportance
  - travelTimeImportance
  - travelHopImportance

  The expected output for all the above cases: throw IllegalArgumentException

Boundary Cases

addCity()

- cityName is 1 character:
  Expected output: true
- nightlyHotelCost is less than 1.
  Expected output: throw IllegalArgumentException.

addFlight()

- flightTime is less than 1
- flightCost is less than 1
  The expected output for all the above cases: throw IllegalArgumentException

addTrain()

- trainTime is less than 1
- trainCost is less than 1
  The expected output for all the above cases: throw IllegalArgumentException

planTrip()

- costImportance is 0
- travelTimeImportance is 0
- travelCostImportance is 0
- costImportance & travelTimeImportance is 0
- travelTimeImportance & travelCostImportance is 0
- costImportance & travelCostImportance is 0
- costImportance, travelTimeImportance & travelCostImportance is 0
  ExpectedOutput: First travel path that is encountered.

<u>Control Flow Cases</u>

addCity()

- Add a city when a City with cityName does not exist in the graph
  Expected output: true
- Add a city when an object City with the same cityName already exists in the graph but with different values for testRequired, timeToTest or nightlyHotelCost as the object City.
  Expected output: false
- Add a city when an object City with the same cityName already exists in the graph but with the same values for testRequired, timeToTest and nightlyHotelCost as the object City.
  Expected output: true

addFlight()

- startCity does not exist
- destinationCity does not exist
- startCity and destinationCity are same
  The expected output for all the above cases: throw IllegalArgumentException

addTrain()

- startCity does not exist
- destinationCity does not exist
- startCity and destinationCity are same
  The expected output for all the above cases: throw IllegalArgumentException

planTrip()

- startCity does not exist
  Expected output: throw IllegalArgumentException

- destinationCity does not exist
  Expected output: throw IllegalArgumentException
- startCity and destinationCity are same
  Expected output: return List [start startCity]

Data Flow Cases

addFlight()

- A flight already exists between startCity and destinationCity.
  Expected output: throw IllegalArgumentException

addTrain()

- A train already exists between startCity and destinationCity.
  Expected output: throw IllegalArgumentException

planTrip()

- No path exist between two cities
  Expected output: null
- Multiple paths exist between two cities
  Expected output: Path with least distance from the source