Assignment 3

Overview:

During the COVID times, traveling between cities/countries is not convenient. Depending on the traveler's priorities, an optimal path is required to reach the destination city. The travel assistant program allows the traveler to choose an optimal path based on their priorities of cost, travel time, and the number of hops.

Note: A comprehensive description of the travel assistant is in the "CSCI 3901 Assignment 3.pdf" file.

Implementation Overview:

I have built a graph for keeping track of connections between different cities. The City class acts as the vertex of the graph. The TravelHop class acts as the edge of the graph. I have used Dijkstra's shortest path algorithm to find the shortest route between any two cities. Depending on whether an individual is vaccinated or not, the optimal path might change.

Files and external data:

1. TravelAssistant.java
   This file contains the *TravelAssistant* class. The class has public methods for adding cities, adding flights, adding trains and planning trips. It also has other private methods acting as the helper methods for the public methods.
2. TravelHop.java
   This file contains the *TravelHop* class. This class acts as an edge of the graph.
3. City.java
   This file contains the *City* class. The City class acts as the vertex of the graph.
4. CityWeight.java
   The class holds the priority of the city while traversing the graph using Dijkstra's shortest path algorithm. The class also holds the information of whether the test report is negative.
5. Junit files: TestAddCity.java, TestAddFlight.java, TestAddTrain.java, TestPlanTrip.java
   These files contain the Junit test cases for testing following methods
   i. addCity()
   ii. addFlight()
   iii. addTrain()
   iv. planTrip()

   Every file tests methods separately. In each file, the tests have been divided into following categories

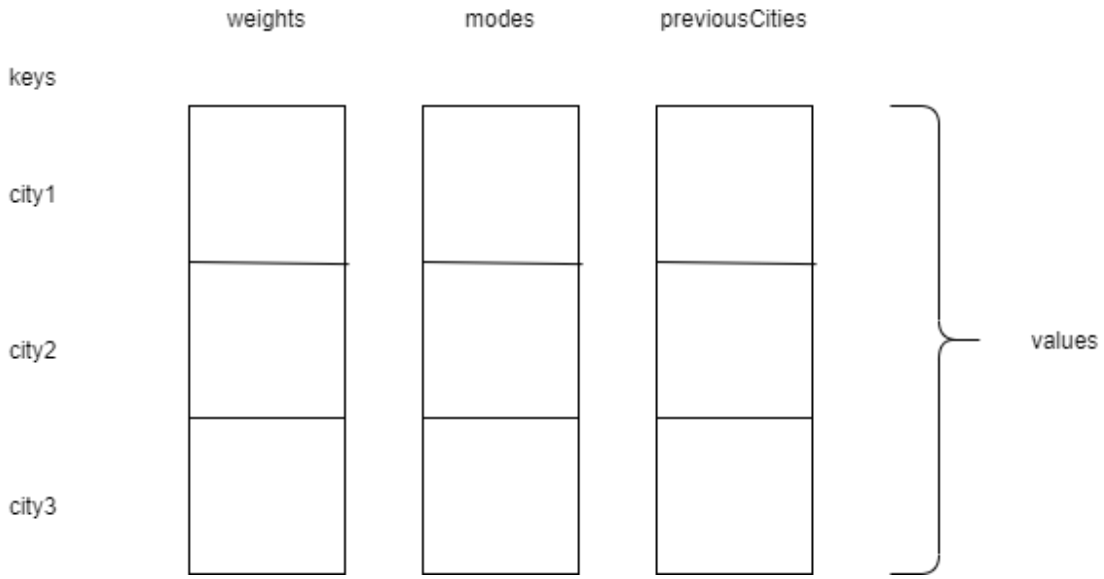
   i. Input Validation
   ii. Boundary Cases
   iii. Control Flow Cases

iv. Data Flow Cases

Efficiency of the data Structures and algorithms/ Relationship to each other

A. Graph structure:
   1. I have used a map to store the cities (vertices). The map has city name as the key and the city object as the value (*Map<String, City>).* With the map, the city object will be accessible in the constant time.
   2. I have used a map to store the adjacency list. The map has the city object as the key and the list of travel hops (edges) as the value (*Map<City, List<TravelHop>>).* The map allows accessing the hops connected to a city in the constant time.
   3. The first map allows accessing the city object using the name of the city in the constant time. In the second map, with the help of the first map the, the list of edges connected to the cities can be accessed in the constant time.

B. Data structures for holding information about the weights, modes and previous cities while following Dijkstra's algorithm:



   1. I have used three maps to store the information in a format similar to a table.
      Map<City, Integer> for weights
      Map<City, City> for previousCities
      Map<City, String> for modes
   2. The maps allow the information to be stored in a simple tabular format.
   3. All three maps have the City class as the key. If a lookup is needed of which city is connected to the source city, it is possible to get the mode of travel from the previous city, previous city and the weight from the source city all at once in the constant time.

C. Keeping track of the visited cities:

I have used a set to keep track of already visited cities in the Dijkstra's algorithm(*Set<City>).* The set only allows unique cities. Hence, an already visited city cannot be visited again.

D. A priority queue for breadth first traversal:
I am using a priority queue for processing cities based priority/weight. The city with the shortest weight gets visited first (*Queue<CityWeight>).* The priority queue removes the city with the smallest weight in order.

E. A stack for tracing the shortest path
I have used a stack to trace back the path from the destination city to the source city using the tables that I mentioned in the section B. The stack then presents the path in the correct order.

F. Planning trip:
   1. The planning trip method does the calculation of finding the shortest distances between the start city and all other cities.
   2. I have used the three maps to keep track of the weights to travel from source city, the mode of travel and the previously visited city.
   3. A priority queue is used for doing the breadth first traversal.
   4. The algorithm starts by adding the source city to the queue.
   5. The city gets removed and all the adjacent city weight gets calculated.
   6. There's a check if see if it is possible to visit the city based on the conditions mentioned in the section E.
   7. If it is not possible to visit the city, then that city gets skipped.
   8. If it is possible to visit a city, then it checks whether a test is needed in the current city.
   9. If the test is needed, then it adds the hotel cost to the weight of traveling between two cities.
   10. The formula for calculating the weight is as follows:
   weight = cost * costImportance + travelTime * timeImportance + hop * hopImportance
   If the calculated weight is less than the current weight to reach from the source city, then the three tables get updated with the latest values.
   If there's no path, then the method returns null. Otherwise, it returns the list of paths.
   11. The getPath method uses a stack to trace back the path to the source city.
   12. The stack gets popped to find the correct path to the destination city.

Assumptions:

1. Flight/Train/Hotel cost cannot be 0
2. Flight/Train time cannot be 0
3. All costs are in same currency
4. Travel times are in minutes
5. Time to test implies two complete days of stay in the city

Key algorithms and design elements:

A. Adding city:
   The *addCity* method allows adding a city to the graph. The method adds the city to the map as well as the adjacency list by initializing an empty list of travel hops. The method returns false if a city already exists in the system.

B. Adding flight/train:
   I have used a new method called *addTravelHop* to add both flights and trains. The method checks if the flight/train already exists. If it doesn't exist, then it adds it to the adjacency list.

C. Method to check if the travel hop is already present:
   When a hop (train/flight) already exists between two cities, the system should not allow adding that hop. I have created a method called *isTravelHopPresent* to check if the hop is already present by iterating over the source city adjacency list.

D. Printing graph:
   I have created a print method to show the connections between different cities. The output appears as follows.

E. Method to check if it is possible to visit the city:
   The *isVisitable* method checks if it is possible to visit the city based on whether the traveler is vaccinated, negative report available, the test required criteria in the next city and the testing availability in the current city.

F. Method to check if the test is needed in the current city
   The *isTestNeeded* method checks if a test is needed in the current city based on the status of the traveler (Vaccinated/Negative report) and the testing requirements (next city)/testing availability (current city).

G. Getting path:
   The *getPath* method is used to get the shortest path between the start city and the destination city.

Limitations:

1. Only one flight/train can be added in one direction between two cities.
2. Cannot find a path when all three importance parameters are 0.
3. An optimal path is not guaranteed for an unvaccinated individual.
4. Possibility of not having any path between the source and the destination city.

Test cases:

A. Input Validation

addCity():

- Empty string passed as city name
- Null value passed as city name
- Nightly hotel cost is 0
- Nightly hotel cost is negative

addFlight():

- Empty string passed as start city name
- Null value passed as start city name
- Empty string passed as destination city name
- Null value passed as destination city name
- Flight time is negative
- Flight time is 0
- Flight cost is negative
- Flight cost is 0

addTrain():

- Empty string passed as start city name
- Null value passed as start city name
- Empty string passed as destination city name
- Null value passed as destination city name
- Train time is negative
- Train time is 0
- Train cost is negative
- Train cost is 0

planTrip():

- Empty string passed as start city
- Null value passed as start city
- Empty string passed as destination city
- Null value passed as destination city
- Cost importance is a negative value
- Time importance is a negative value
- Hop importance is a negative value

B. Boundary cases

addCity():

- 1 character city name
- Nightly hotel cost is 1
- Time to test is 0

addFlight():

- 1 character source city
- 1 character destination city
- Flight time is 1 minute
- Flight cost is 1

addTrain():

- 1 character source city
- 1 character destination city
- Train time is 1 minute
- Train cost is 1

planTrip():

- 1 character start city name
- 1 character destination city name
- Cost importance is 0
- Time importance is 0
- Hop importance is 0

C. Control Flow Cases

addCity():

- Duplicate city name
- Create a city when there are no cities already created
- Create a city when there is one city already created
- Create a city when there are many cities already created
- Time to test is more than 0 days
- Nightly hotel cost is more than 1
- Time to test is less than 0
- Test required is false
- Test required is true

addFlight():

- Duplicate flight between two cities

- Add flight when there are no flights already present
- Add flight when one flight is already present
- Add flight when many flights are already present
- Add flight from cities A to B when a flight B to A already exists
- Flight time is more than 1 minute
- Flight cost is more than 1
- Start city and destination city are the same city

addTrain():

- Duplicate train between two cities
- Add train when there are no trains already present
- Add train when one train is already present
- Add train when many trains are already present
- Add train from cities A to B when a train B to A already exists
- Train time is more than 1 minute
- Train cost is more than 1
- Start city and destination city are the same city

planTrip():

- Two same plan trips back to back.
- Plan a trip when no trips already exist
- Plan a trip when one trip already exist
- Plan a trip when many plan trips exist
- Start city does not exist
- Destination city does not exist
- Both start and destination city exist
- isVaccinated is true
- isVaccinated is false
- Cost importance 0, Time importance is 0, Hop importance is 0
- Cost importance 1, Time importance is 0, Hop importance is 0
- Cost importance 0, Time importance is 1, Hop importance is 0
- Cost importance 0, Time importance is 0, Hop importance is 1
- Cost importance 1, Time importance is 1, Hop importance is 1
- Cost importance 1, Time importance is 1, Hop importance is 0
- Cost importance 0, Time importance is 1, Hop importance is 1
- Cost importance 1, Time importance is 0, Hop importance is 1
- Cost importance > 1
- Time importance > 1
- Hop importance > 1
- Plan a trip when only one city is defined
- Plan a trip when no city is present

- Plan a trip when many cities are present
- Plan a trip when no connections are present between cities
- Plan a trip when a few connections are present between cities
- Plan a trip when all the cities are connected to each other in one direction
- Plan a trip when all the cities are connected to each other in both the directions
- Plan a trip when the cities are connected by both trains and flights

D. Data Flow Cases

addFlight():

- Add flight when the start city does not exist
- Add flight when the destination city does not exist
- Add flight when the start city and destination cities both do not exist
- Add flight when the start and the destination city exist
- Add flight when a train exists in the same path

addtrain():

- Add train when the start city does not exist
- Add train when the destination city does not exist
- Add train when the start city and destination cities both do not exist
- Add train when the start and the destination city exist
- Add train when a flight exists in the same path

planTrip() :

- Unvaccinated, start city timeToTest >= 0, cities connected to source cities testRequired is true
- Unvaccinated, start city timeToTest < 0, cities connected to source cities testRequired is true
- Unvaccinated, start city timeToTest < 0, cities connected to source cities testRequired is false
- Vaccinated, start city timeToTest >= 0, cities connected to source cities testRequired is true
- Vaccinated, start city timeToTest < 0, cities connected to source cities testRequired is true
- Vaccinated, start city timeToTest < 0, cities connected to source cities testRequired is false
- Unvaccinated, start city timeToTest < 0, only one of the adjacent city of the source testRequired is false
- Unvaccinated, start city timeToTest < 0, two consecutive cities to the destination path test required is false, destination test required is true
- Unvaccinated, start city timeToTest < 0, adjacent city test required is false, the next city test required is true, destination test required is true
- Unvaccinated, start city timeToTest < 0, a series of cities where the test is not required, destination city test required is true
- Unvaccinated, start city timeToTest < 0, a series of cities where the test is not required, destination city test required is false