Assignment 3

Team Members

Meet Nitinbhai Patel

(B00899516)

Subject:

Software Development Concepts

Professor:

Mike McAllister

Problem 1: Work with graphs

Overview

Traveling during the covid-19 pandemic might be challenging. Varied countries have their own rules. Whether you may travel straight to them from other country. And what kind of test you have to perform before you enter in that country. For example, Currently, if you want to enter in Canada from India, you will need an RTP-CR from third country before entering Canada.

When planning travel plans, travelers may have varied priorities. Some people aim to save costs as much as possible. Others choose to travel on shared flights and trains in order to save time. Others want to travel with as few stops as possible. I have developed a program that will optimize the travel plan of the travelers so that they can travel from one city to another.

Files and external data

1. **TravelAssistant.java**

Methods

- **boolean addCity(String cityName, boolean testRequired, int timeToTest, int nightlyHotelCost) throws IllegalArgumentException**
    - It is used to add the city (Source City or Destination City) in the travel plan.
    - This method returns false and throw IllegalArgumentException if cityName is null or empty. Or if the timeToTest is less than zero or nightlyHotelStary is less than or equal to 0.
    - The timeToTest is in days and hotel cost is in particular currency.
    - .In this method cityName must be unique. If you enter the city which is already present then it will retrun false.
    - If the city is successfully added in the it will return true

- **boolean addFlight(String startCity, String destinationCity, int flightTime, int flightCost)) throws IllegalArgumentException**
    - It is used to create a route from startCity to the destinationCity.

  - This method returns false and throw IllegalArgumentException if startCity or destinationCity are null or empty or if flightTime or flightCost are less than or equal to Zero.
  - This method will return false if the startCity or destinationCity is not present in the TravelAssistant.
  - It will also retrun false if we try to enter the route which is already present.
  - The flightTime is the in-air time of the flight which is in minutes and flightCost is in particular currency.
  - This method will return true if the particular route is added to the TravelAssistant

- **boolean addTrain(String startCity, String destinationCity, int trainTime, int trainCost)) throws IllegalArgumentException**
  - It is used to create a route from startCity to the destinationCity.
  - This method returns false and throw IllegalArgumentException if startCity or destinationCity are null or empty or if trainTime or trainCost are less than or equal to Zero.
  - It will also return false if we try to enter the route which is already present.
  - The trainTime is the in-air time of the flight which is in minutes and trainCost is in particular currency.
  - This method will return true if the particular route is added to the TravelAssistant.

- **List<String> planTrip (String startCity, String destinationCity, boolean isVaccinated, int costImportance, int travelTimeImportance, int travelHopImportance) throws IllegalArgumentException**
  - This method is used to find the shortest path from startCity to destinationCity based on costImportance, travelImportance , travelHopImportance and vaccination status.
  - This method returns false if the route is not present from startCity to destination city
  - the method also returns false if costImportance or travelImportance or travelHopImportance is negative
  - this method also ask if you are vaccinated or not.
  - This Methods will return a list of string containing the shortest path along with mode of travel.

    For Example: If I'm travelling from A to B to C and I fly from A to B and take the train from B to C then this planTrip method returns with a list of 3 strings: "start A", "fly B", "train C"

- **int startCityId(String startCity)**
    - returns the id of startCity

- **int destinationId(String destinationCity)**
    - returns the id of destinationCity

- **boolean isCityExists(String cityName)**
    - returns true if city exists else retrun false

- **boolean isExists(String startCity, String destinationCity)**
    - returns true if route from startCity to destinationCity exists else return false

2. DijkstraAlgo.java

Methods

- **List<String> findRoute(CreateVertex source, int costImportance, int travelTimeImportance, int travelHopImportance, int getStartCityId, int getDestinationId)**
    - this method will return a list of string contain the shortest path from source to destination based on the cost importance, travel time Importance and travel hop importance.

- **List<CreateVertex> getFinalRoute**
    - Retrun the finalRoute

- **int nodeDistance(CreateVertex node1, CreateVertex node2, int costImportance, int travelTimeImportance, int travelHopImportance)**
    - This method returns the distance between the two cities

- **List<CreateVertex> getneighbouringNode(CreateVertex city)**
    - This method returns the list of the neighbouring cities of the provided city

- **List<Integer> getIList()**
    - Returns the list of indexes of the routes which is present in the finalRoute list. this will be useful for getting the mode of transportation of the routes in in the final route

3. **CreateVertex.java**

Methods

- **int getCityID()**
    - **Return the id of City**
- **String getCityName()**
    - **Return the name of city**

4. **CreateEdge.java**

Methods

- **int getTime()** → Return the travelling time
- **int getHop()** → Return the hop
- **int getCost()** → Return the travelling cost
- **String getModeOfTransportation()** → Return the getModeOfTransportation
- **CreateVertex getSourceCity()** → Return source city
- **CreateVertex getDestinationCity()** → Return Destination City

5. **RouteDetails.java**

Methods

- **String getStartCity()** → Returns the start city
- **String getDestinationCity()** → Returns the destination City
- **int getTime()** → Return route travelling time
- **int getCost()** → Return route travelling cost

6. **Graph.java**

Methods

- **List<CreateVertex> getCities()** → returns list of cities
- **List<CreateEdge> getRoutes()** → returns list of routes

7. **CityDetails.java**

Methods

- **String getCityName()** → Return the name of city
- **boolean isTestRequired()** → Return true or false
- **int getTimeToTest()** → Return the number of days required to get the result
- **int getNightlyHotelCost()** → Return the cost of hotel for one night

Data structures and their relations to each other

Data Structure Used in the Code

1. **LinkedList**
   - This data structure is used to store details of cities and the details of routes between two cities.
   - This data structure is also used to store the list of visited node, not visited nodes and the final shortest path from source to destination.

   Advantage of using LinkedList over other data structure

   - The main advantages of using LinkedList is that element can be inserted or removed without reorganization of entire structure. i e Insertion and deletion is efficient in LinkedList as compared to other data structure.
   - And another advantage is that we don't need to define size at the time of initialization.
   - We can easily implement stack and queue using LinkedList.

For storing details of cities and routes I have used LinkedList because LinkedList id dynamic data structure. It can grow and shrink at runtime. So, it doesn't require any initial size unlike array which requires initial size. As we don't know how many cities and routes we are going to add, LinkedList would be the great option. Another advantage is that there is no memory wastage.

For storing visitedCities and not visitedCities I have used LinkedList because we are inserting and removing the cities. So, the LinkedList data structure would be great for this.

2. **Map**
   - This data structure is used to store the predecessors of the given city. The key is the predecessors and the values is the city.
   - This is also used to store the distance of the city from start city. The key is the city and the value is the distance

   Advantage of using Map

   - We can access element faster as it used the hashing technology
   - HashMaps are useful for finding values based on a key, as well as adding and removing data based on a key.

Classes used in the code

1. TravelAssistant → Manage the overall functionality of the code. From creation of city, route till getting the final shortest path.
2. DijkstraAlgo →Manages the core part of the code. It is the main class responsible for finding the shortest path.
3. CreateEdge → Manages the creation of an edge between two cities
4. CreateVertex → Manages the creation of city
5. CityDetails → Manages the creation of city with its description
6. Graph → Used to create a graph for the provided list of city and route
7. RouteDetails → Manages the creation of route with details

Assumptions

- All travel time used in the code are in minutes
- All the cost used in code are comparable
- The graph is directed in one-way. I e. If you add a route from Halifax to Toronto then it will create a route from Halifax to Toronto only. It will not automatically create a route from Toronto to Halifax

Key algorithms and design elements

The key algorithm is the Dijkstra’s shortest path algorithm which will find the shortest path from source to destination.

Dijkstra’s shortest path algorithm

- All nodes are divided into two groups by Dijkstra Algorithm
    - Visited Nodes
    - Not Visited Nodes
- Visited nodes are those whose minimum distance is known from the source node. Whereas the not visited nodes contains the list of nodes which are reachable from the source but the nodes don’t have the minimum distance from the source node
- Steps to find the shortest path
    - Set distance to startCity to 0
    - Set all other distances to the very high value / Infinite value
    - Initially we add the start node to not visited nodes set
    - Till the not visited nodes set is not empty we
        - Select a node from not visited node list set which has the lowest distance from the source.
        - Calculate new distance to the adjacent node by keeping the lowest distance at each evaluation.
        - Add the neighbor which are not visited to the not visited nodes set

The list of nodes in the visited nodes list is the shortest path from the source.

In the TravelAssistant class first the cities are created using addCity() method. After that addFlight() and addTrain() methods create the route between two cities. Then the user plans the trip by providing the startCity , destinationCity, vaccination status, cost importance, travel time Importance and travel hop importance. Based on provided input from the user the Dijkstra’s algorithm finds the shortest path between two cities. The plan trip method returns the list of cities visited along with the mode of travel. Each string in the returned list consists of the mode of travel (start, fly, train), a space, and then the city at the end of the flight or train ride.

The most efficient way to find the shortest path is to use the Dijkstra Algorithm and storing the graph in adjacency list. All this have been implemented in the code.

Limitation/Drawbacks

- If the person is not vaccinated then there are chances that some test cases might fail
- There is only one mode of travel from startCity to destinationCity. You can go either by train or by flight. You cannot travel from startCity to destinationCity to by both train and flight.

Test Cases

Input Validation (Generally, tests on bad input data for which you shouldn't crash)

Method 1: boolean addCity(String cityName, boolean testRequired, int timeToTest, int nightlyHotelCost)

- Null Value passed as cityName
- Empty string passed as cityName
- For NighttlyHotelCost
  - Value is negative
  - Value is zero

Method 2: boolean addFlight(String startCity, String destinationCity, int flightTime, int flightCost)

- Null value passed in startCity
- Empty string passed in startCity
- Null value passed in destinationCity
- Empty string passed in destinationCity
- Passed startCity doesn't exists
- Passed destinationCity doesn't exists
- For flightTime
  - Value is negative
  - Value is Zero
- For flighrCost
  - Value is negative
  - Value is Zero

Method 3: boolean addTrain(String startCity, String destinationCity, int trainTime, int trainCost)

- Null value passed in startCity
- Empty string passed in startCity
- Null value passed in destinationCity
- Empty string passed in destinationCity
- Passed startCity doesn't exists
- Passed destinationCity doesn't exists
-

- For trainTime
    - Value is negative
    - Value is Zero
- For trainCost
    - Value is negative
    - Value is Zero

Method 4: List<String> planTrip (String startCity, String destinationCity, boolean isVaccinated, int costImportance, int travelTimeImportance, int travelHopImportance)

- Null value passed in startCity
- Empty string passed in startCity
- Null value passed in destinationCity
- Empty string passed in destinationCity
- Passed route from startCity to destinationCity doesn't exists
- For costImportance value is negative
- For travelTimeImportance value is negative
- For travelHopImportance value is negative

Boundary Cases (Tests at the edge of inputs or problem structures)

Method 1: boolean addCity(String cityName, boolean testRequired, int timeToTest, int nightlyHotelCost)

- 1-character cityName
- For nightlyHotelCost value is 1

Method 2: boolean addFlight(String startCity, String destinationCity, int flightTime, int flightCost)

- 1-character startCity name
- 1-character destinationCity name
- For flightTime value is 1
- For flightCost value is 1

Method 3: boolean addTrain(String startCity, String destinationCity, int trainTime, int trainCost)

- 1-character startCity name
- 1-character destinationCity name
- For trainTime value is 1
- For trainCost value is 1

Method 4: List<String> planTrip (String startCity, String destinationCity, boolean isVaccinated, int costImportance, int travelTimeImportance, int travelHopImportance)

- 1-character startCity name
- 1-Character destinationCity name
- For costImportance value is 0
- For travelTimeImportance value is 0
- For travelHopImportance value is 0

Control Flow Test Cases (Tests of the core operations)

Method 1: boolean addCity(String cityName, boolean testRequired, int timeToTest, int nightlyHotelCost)

- cityName already present in travel Assistance
- Add city when there is no city added
- Add city when there is 1 city already added
- Add city when there are many cities already present
- For nightlyHotelCost value is more then 1
- Add city when testRequired is True
- Add city when testRequired is False

Method 2: boolean addFlight(String startCity, String destinationCity, int flightTime, int flightCost)

- Add flight when startCity doesn’t exist
- Add Flight when destinationCioty doesn’t exists
- Add flight when no flight is added
- Add flight when 1 flight is added
- Add flight when there are many flights already present
- For flightTime value is more then 1
- For flightCost value is more then 1

- Added Flight overlaps with already existing flight/route
- Add flight with same startCity and destinationCity

Method 3: boolean addTrain(String startCity, String destinationCity, int trainTime, int trainCost)

- Add train when startCity doesn't exist
- Add train when destinationCioty doesn't exists
- Add train when no flight is added
- Add train when 1 flight is added
- Add train when there are many flights already present
- For trainTime value is more then 1
- For tarinCost value is more then 1
- Added Train overlaps with already existing train/route
- Add train with same startCity and destinationCity

Method 4: List<String> planTrip (String startCity, String destinationCity, boolean isVaccinated, int costImportance, int travelTimeImportance, int travelHopImportance)

- planTrip when startCity and destinationCity doesn't exists
- planTrip when there is no flight or train from startCity to destinationCity
- planTrip when there is 1 flight or train
- planTrip when there is more than 1 flight or train
- planTrip when no trip is planned
- planTrip when 1 trip is planned
- planTrip when more than 1 trip is planned
- planTrip when costImportance, travelTimeImportance, travelHopImportance is more than 0
- planTrip when traveler is not vaccinated
- planTrip when traveler is vaccinated

Data Flow Test Cases (Tests around the order in which things are done)

Normal execution order:

addCity () → addFlight () / addTrain() → planTrip ()

Data Flow

1. boolean addCity("halifax", true, 5, 50) → *add halifax to the Travel Assistant*
2. boolean addCity("mumbai", true, 6, 60) → *add mumbai to the Travel Assistant*
3. boolean addCity("goa", true, 7, 70) → *add goa to the Travel Assistant*
4. boolean addFlight("halifax", "mumbai", 55, 200) → *add route from halifax to mumbai*
5. boolean addTrain("mumbai", "goa", 60, 250) → *add route from mumbai to goa*
6. boolean addFlight("halifax", "goa", 200, 1000) → *add route from Halifax to goa*
7. List<String> planTrip ("mumbai", "goa", true, 1, 1, 1) → *returns the shortest path between mumbai and goa ([Start halifax, fly mumbai, train goa])*