CSCI 3901

Software Development Concepts

Assignment 3

Team Members

B00874162 - Vivekkumar Rameshbhai Patel

Shortest Path(Dijkstra’s algorithm):

Optimize Travel Plan

Assignment 3: Overview

There are more than one mode of commutation from one city to another such as Train or Flight directly or indirectly but in the adverse time of covid-19, each cities has their own requirement that whether direct arrival is allowed in any specific city or not or requirement of vaccination status of each individual passing through in that specific cities. Hence, **time required to reach destination**, **cost of travelling** and **number of cities** that individual pass through has become crucial to make optimize travel plan.

This program optimize the travel path to travel from one city to another. In order to optimize a travel plan **TravelAssitant** class needs to be implemented, which takes input from the user such as through which cities they are willing to travel, mode of transportation such as flight or train between two cities, vaccination requirement in any specific city, importance for the travel cost, travel time and number of indirect route. Eventually based on the origin and destination cities that traveler wants to travel it calculates optimize travel path.

In order to find optimal travel path, Graph data structure has been created based on the input provide by the user, to make a travel network. In which each city is represented as the vertex of the graph and the total cost including cost of commutation and cost required to stay in hotel based on time required to get the covid test in any specific city is considered as the edge of the graph. Problem of making optimal travel path has been resolved using Dijkstra’s algorithm, in which cost of travelling between any two cities are calculated first and that cost is compared further if one specific city has more than one way to reach the destination and then path with minimal cost is considered for further commutation.

Strategy/Algorithm to find optimal travel path:

For each method of providing input such as city, flight and train details, first acceptability of the provided input is checked and then decision is taken to preserve the input in to the system or not. Moreover based on the acceptable input, network of traveling is created and then this network of travel with all mandatory details is used further to get optimal travel path.

Once, necessary information are available for travelling, procedure of finding optimal path starts with calculating cost of travelling for each cities. Initially each cities are marked with infinite cost. Then origin city is marked as current visiting city as well as visited city and cost of travelling from origin cities to all its neighbor cities are calculated. Furthermore, cost of travelling to each neighbor is compared with current cost of the respective neighbor which is infinite and minimal cost is set for the neighbor. Now cost of travelling to neighbor is compared with other neighbor and then the neighbor with the most minimal cost is selected. Then this neighbor city is marked as currently visited city. The above procedure is done for the all the neighbor cities of the new visited city and it is repeated until all cities are not marked as visited city. Once the all the cities are marked as visited city, each node has minimal path to travel from the source and path to travel destination from source is retrieved.

Degree of efficiency:

- I have used Hash Map data structure to store detail of the city, flight and train to obtain **O(1)** time complexity while searching and storing the information in computation of optimal path.
- To store the graph I have used the concept of adjacency list. In which I have optimized the searching time by using hash map for storing details of each vertex and edge.
- Time complexity of the finding minimal path using Dijkstra’s algorithm in my solution is **O(V^2)**.Where **V** is vertex of the graph. In this approach I have used greedy approach to find minimal path. Hence, cost of each neighbour vertex of specific vertex is calculated and then minimal path is

derived for commutation. Hence it tries every possible neighbour vertex of the specific vertex. Mode of transportation is selected in the similar iteration hence no extra computation is made for it.

Files and external data:

There are total 5 files:

- **TravelAssistantInterface.java:** It contains methods that needs to be implemented for implementation of **TravelAssistant**.
-**TravelAssistant.java:** Class that contains implementation of the methods available in **TravelAssistantInterface** to obtain optimal path of travelling.
-**City.java:** This class has all attributes and methods to store details of the provided cities including flights and train details.
-**Flight.java:** Class contains all the necessary methods and attributes to store details of the flight between cities.
-**Train.java:** Class contains all the necessary methods and attributes to store details of the train between cities

Data structures and their relations to each other:

Data structured used in various classes

TravelAssistant.java:

listOfCities: It is hash map used to store details of cities.In which name of city is used as a key and object of city class as a value.In addCity(),addFlight(),addTrain() and planTrip() methods it is used to get the details of the city.

City.java:

-**listOfFlightFromSrcToDest**:It is hash map used to store details of flight between source to destination. In which name of destination city is stored as a key and object of flight as a value. To calculate flight cost to travel from source to destination , details of source city is retrieves from the hash map(**listOfCities**) and from that city object retrieve the details of the flight between source to destination from this hash map.

-**listOfTrainFromSrcToDest**: It is hash map used to store details of train between source to destination. In which name of destination city is stored as a key and object of flight as a value. To calculate train cost to travel from source to destination , details of source city is retrieves from the hash map(**listOfCities**) and from that city object retrieve the details of the trains between source to destination from this hash map.

-**neighBourCities:**

It is hash set. Which maintains list of all cities which are adjacent to this specific city.

minmumCostRoute:

It is arraylist which maintains list of all route with minimal cost to reach the destination

Test Cases

Input Validation:

addCity():

- City Name is null or empty throw IllegalArgumentException.
- Nightly hotel cost is negative or 0 throw IllegalArgumentException.

addFlight():

- For flight name of start city is null or empty throw IllegalArgumentException.
- For flight name of destination city is null or empty throw IllegalArgumentException.
- For flight, flight time is negative or 0 throw IllegalArgumentException.
- For flight, flight cost is negative or 0 throw IllegalArgumentException.

addTrain():

- For train name of start city is null or empty throw IllegalArgumentException.
- For train name of destination city is null or empty throw IllegalArgumentException.
- For train, train time is negative or 0 throw IllegalArgumentException.
- For train, train cost is negative or 0 throw IllegalArgumentException.

- For train, if train details is already exist for the given source and destination return false.

planTrip():

- To get optimal path if name of the provided source city is null or empty throw IllegalArgumentException.
- To get optimal path if name of the provided destination city is null or empty throw IllegalArgumentException.
- To get optimal path if priority of cost, time and hop is negative throw IllegalArgumentException

Boundary Tests:

addCity():

- City name is of single character return true.
- Time to test is of single digit return true
- Cost of hotel is of single digit return true

addFlight():

- Flight time is of single digit return true.
- Flight cost is of single digit return true

addTrain():

- Train time is of single digit return true.
- Train cost is of single digit return true.

planTrip():

- Value for priority of cost, time and hop is of single digit.

Control Flow Tests:

addCity():

- The provided city detail is already exist return false.

- For flight, name of start city does not exist in the created travel network throw IllegalArgumentException.
- For flight, name of destination city does not exist in the created travel network throw IllegalArgumentException.
- For flight, if flight details is already exist for the given source and destination return false.

<u>addFlight()</u>:

- For flight, name of start city does not exist in the created travel network throw IllegalArgumentException.
- For flight, name of destination city does not exist in the created travel network throw IllegalArgumentException.
- For flight, if flight details is already exist for the given source and destination return false.

<u>addTrain()</u>:

- For train, name of start city does not exist in the created travel network throw IllegalArgumentException.
- For train, name of destination city does not exist in the created travel network throw IllegalArgumentException.
- For train, if train details is already exist for the given source and destination return false.

<u>planTrip()</u>:

- To get optimal path if the provided source city does not exist in the created travel network throw IllegalArgumentException.
- To get optimal path if the provided destination city does not exist in the created travel network throw IllegalArgumentException.
- If path does not exist between source and destination city return null

Dataflow Tests:

- Call to addFlight() method before calling addCity()
- Call to addTrain() method before calling addCity()
- Call to planTrip() method before calling addCity()
- Call to addFlight() method after calling addCity()
- Call to addTrain() method after calling addCity()
- Call to planTrip() methods after calling addCity() and addFlight()
- Call to planTrip() methods calling addCity() and addTrain()
- Call to planTrip() after calling addCity() and before calling addFlight() or addTrain()

References:

-To understand the maximum value of Integer.MAX_VALUE:
https://docs.oracle.com/javase/6/docs/api/java/lang/Integer.html?is-external=true
-To get the understanding of the Dijkstra algoritham:
https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/