CSCI 3901 Assignment 3: External Documentation

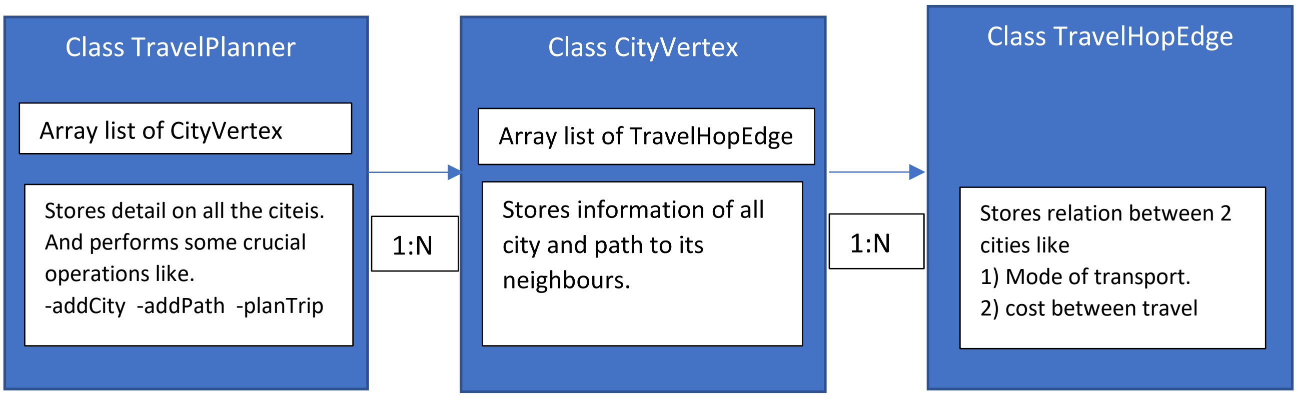
Overview:

TravelAssitant is a class which tries to find the shortest possible path based on the relative cost/time/hop path between 2 cities based on user preference or importance. It has functions to add cities into the planner, add flights and trains between cities and in turn adapts Dijkstra algorithm to find the shortest path between 2 cities chosen by the user, dynamically calculates the best path based on cost or time or number of hops.

Approach

Identified the key components and entities in the problem that is cities and paths categorized them into different classes each based on the best practices, Class CityVertex which is the vertex in our graph and for the path between cities Class Traveledges to store path data between cities, and TravelAssistant class which manages and holds the logic to find best path between source and destination based on user precedence.

class structure overview and relation between them.



Class TravelPlanner
Array list of CityVertex
Stores detail on all the citeis. And performs some crucial operations like.
-addCity -addPath -planTrip
1:N
Class CityVertex
Array list of TravelHopEdge
Stores information of all city and path to its neighbours.
1:N
Class TravelHopEdge
Stores relation between 2 cities like
1) Mode of transport.
2) cost between travel

Data structures and their relations to each other:

Travel Assistant class which holds all elements to gather have an array list of <Cities> stored and in turn each city holds all the <edges> or path from itself.

TravelAssitant *holds* -> Array list of <Cities> *holds* -> ArrayList of Paths from that city <edges>

When a city is being added with its associated data it is stored into list of cities.

When a path is added from a source city to destination, flight or train the source city hold all its paths inside array list of edge object with its associated data.

Relation between them is -> [1: N: N] one to many, one to many.

Each Travel planner can have multiple cities and each city can have multiple paths between them.

Files:

TravelAssistant.java: Travel assistant is a class that stores all the list of cities and means to add cities and paths between then and calculate the best possible path between them.

Key Functions:

1) **addCity** : Function that adds cities into the planner, return true if successfully entered into the planner, else false. Throws illegal argument exception for invalid input.

2) **addFlight** : Function that adds flights between cities, reruns true if successfully added else returns false, Throws illegal argument exception for invalid input parameter.

3) **addTrain** : Function that adds Train between cities, reruns true if successfully added else returns false, Throws illegal argument exception for invalid input parameter.

4) **planTrip** : Function that returns best path between 2 cities, throws illegal argument exception for invalid input.

CityVertex.java: City vertex class that represents each city in a travel planner that stores the state of each city and paths between edges.

TravelHopEdge.java: Travel hop edge is a class that represent each path between 2 cities and cost associated with the travel, mode of transportation, Time.

Assumptions:

1) Cities have a single flight or train between 2 cities.
2) Cities have only 2 modes of travel flight or Train.
3) Getting tested at your source node will not cost you.
4) When Testing Time is 0, you get tested and obtain results within a day.
5) For unvaccinated traveller you can get tested once and move through all the cities without getting tested.

Key algorithms and design elements:

The design was in focus to sperate key elements and process data in an organized way efficiently in anticipation of further change and following the best OOPS principles including encapsulation.

Key logics were grouped into functions like cityExits function, to reuse whenever required, many key logic blocks of code to calculate weight and traversing and finding the best path was made easy by separating it and helps in easy maintainability and improving iteratively.

Algorithm:

Step1: Add cities into the system after validation

Step2: Add paths between cities for traversal after validation of input parameters.

Step3: Get the user input about his source and destination and his precedence on cost, time, hop.

Step4: So, start from the source node which is now marked as visited, visit all edges from it and calculate the weight based on user precedence and update the neighbouring city weight and keep track of previous node through which it has been visited.

Step5: Next, choose a minimum weighted city and traverse all the cities from there and update values of it if the weight between them.

Step6: Add cost of traversal from the source node + cost of traversal from this neighbouring node and update its weight with the combined one or replace if this is minimum than the previous weight.

Step7: Go to step 4 until we have visited all the cities

Step8: After we have visited all cities back track the destination node to the source node based on the data stored previously until we find the source node and this is the shortest path.

Key Details:

The algorithm is an adaptation of Dijkstra algorithm and uses and adjacency list to store data of both cities and its edges between them.

we can have many cities and each cities stores path to its neighbouring cities, before entering a city we validate for bad data and throw an exception cannot store multiple cities, and also validate flight or train details.

We will reject and redundant flight between 2 cities, each edge will have a cost and time associated.

Based on user priority we calculate a common weight using the below formula.

[Cost Importance * Edge Cost] + [Travel Importance * Time in minutes] + [Hop imp * 1].

Effciency of Algorithms/Data Structure

Efficiency of data structure & complexity:

Since an array list is used to add elements storage complexity O (1), also retrieving it back is Time O (1), since we use array list to store both cities and edges between them and all internal validation below is the Algorithms complexity.

Storage/Space: set(index) -> **O (1)**   **Time:** get(index) -> **O (1)**

Since we use Array list we query the exact index about the data we need which makes it more efficient to use, storage in an array list adds the element at the end of the current an no prior processing is required to maintain any order and hence no time is wasted.

Algorithm Efficiency & complexity:

Complexity of the algorithm depends on the data structure used along with the below factors.

Storage or space complexity is O (V) where n is the cities stores + O (E) where e is the edges. Since we use arraylist storage space is O (n).

Time complexity: since we use while loop to constantly check if we visited all vertices and for loop in turn to see all the paths between cities and maintain the state of the variable. Time complexity in this case becomes O (n log n).

Time: O ((V + E) log V)   **Space:** O (V + E).

Where V is the vertex [cities] and E is the [edges] between them.

Limitations

1) There cannot be multiple flights or trains between cities.
2) The approach is sub optimal and cost of hotel considered for testing may or may not be at the best city possible.
3) City time to test and cost cannot be updated once entered into the system
4) Flight time and cost cannot be updated after entering into the system.
5) User cannot choose the cities he want to avoid.
6) Cannot choose the place he wants to get tested.
7) Path may or may not exists between cities.
8) Output is just a path with no much details on cost, time, based on his precedence to have more clarity on this.

Test Cases:

Input Validation (Must throw an exception)

addCity():

- Null passes as city name.
- Empty string passed as city name.
- Hotel cost is < 0.
- Hotel cost is 0.

addFlight() and addTrain():

- Null passed for Start city or destination city.
- Empty string passed as city name (start or destination)
- City does not exist in the travel planner passed.
- Flight time < 0 or Flight time = 0
- Flight cost < 0 or Flight time = 0.

planTrip():

- Null passed as city name.
- Empty string passed as city name.
- City that does not exists in the travel planner.

Boundary Cases

addCity()

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

addFlight() and addTrain()

- Single character city names
- Flight time is 1
- Flight cost is 1

planTrip():

- Single character city names
- Cost / Time / Hop importance are all 0.
- Cost / Time / Hop importance are all 1.

Control Flow Cases

Addcity():

- Duplicate city name – false
- Create a city when no city exits

- Create a city when one city exits
- Create a city when many city exits.

addFlight():

- Add a flight when no flight exits between cities
- Add flight when flights exist between cities
- Add flight when a Train path already exists
- Add a flight to the same city

addTrain():

- Add a Train when no Train exits between cities
- Add a Train when flight exits between cities
- Add a train when Train path exits between cities
- Add a Train path to the same city.

planTrip():

- Plan trip when cost / time / hop importance are all 1
- Plan trip when city paths are not connected directly.
- For unvaccinated people plan trip when the destination city requires testing and none of the cities in the path have testing.
- For vaccinated people plan trip where destination required testing.
- For Vaccinated people plan trip when none of the city requires testing.
- For unvaccinated people plan trip where all the cities in the path requires testing.
- Plan trip for unvaccinated people when you have testing at the source node.
- Plan trip for unvaccinated people when an intermediate city requires testing.
- Plan trip between 2 cities with multiple edges.
- Plan trip in a circular graph of cities.

Data Flow cases:

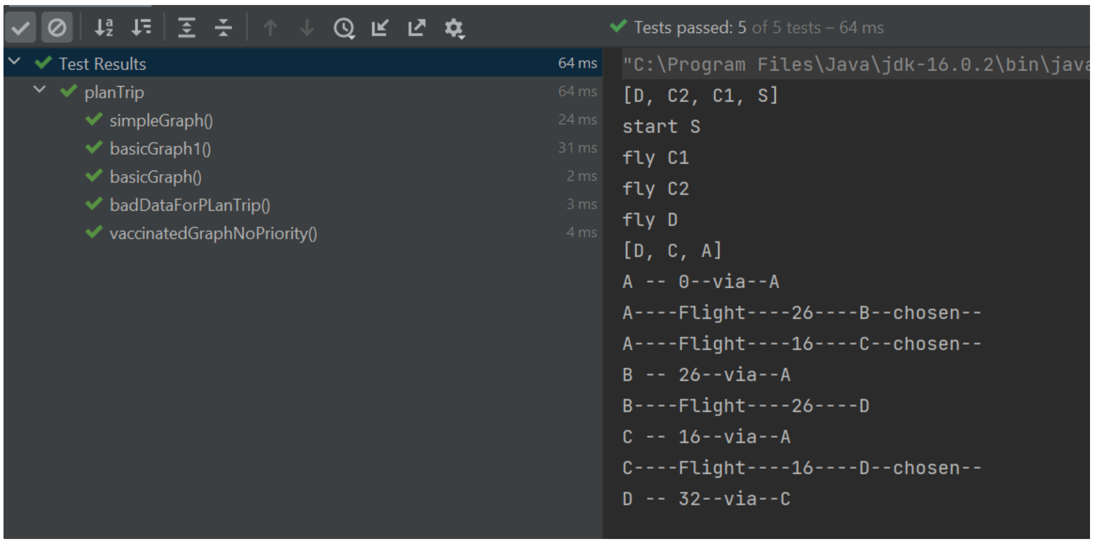
add City (): cases are coved in above scenarios: control flow.

add Train () and add Flight (): - Cases covered in control flow.

plan Trip ():

- Plan trip for several paths from different source and destination one after the other.
- Unvaccinated traveller travelling to city where testing is not required.
- Vaccinated traveller travelling when test is required.
- Unvaccinated traveller in a city where no test is available and the next city requires testing.
- Plan trip when source and destination are same.
- Plan trip for a city with no path.
- Plan trip unvaccinated person from city surrounded by test required cities and test is not available in the source city.

Here are some of the Junit test Cases written for testing the above scenarios.

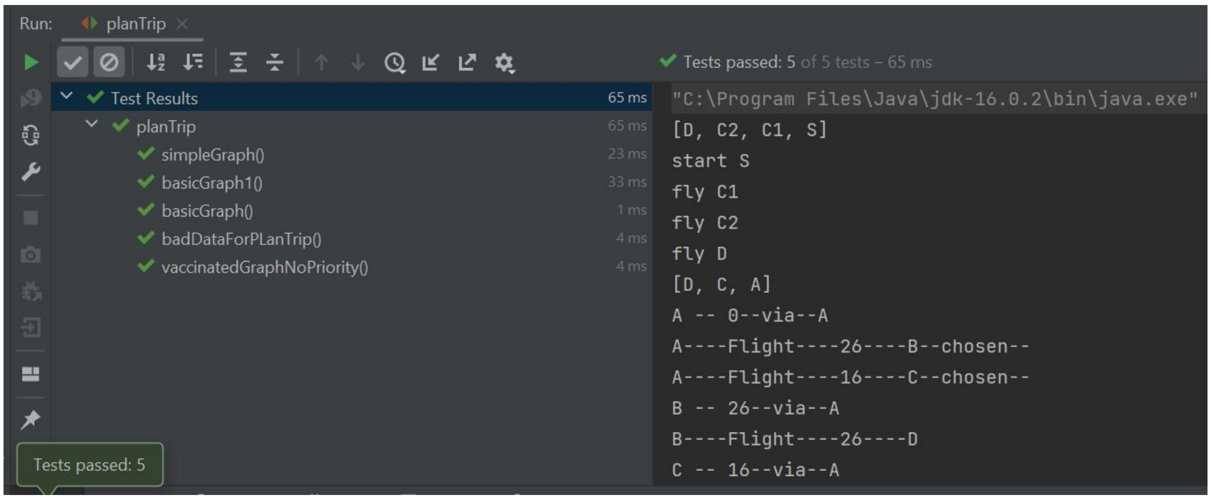


```
Test Results                    64 ms
  planTrip                      64 ms
    simpleGraph()               24 ms
    basicGraph1()               31 ms
    basicGraph()                 2 ms
    badDataForPLanTrip()         3 ms
    vaccinatedGraphNoPriority()  4 ms
```

Tests passed: 5 of 5 tests – 64 ms

```
"C:\Program Files\Java\jdk-16.0.2\bin\java
[D, C2, C1, S]
start S
fly C1
fly C2
fly D
[D, C, A]
A -- 0--via--A
A----Flight----26----B--chosen--
A----Flight----16----C--chosen--
B -- 26--via--A
B----Flight----26----D
C -- 16--via--A
C----Flight----16----D--chosen--
D -- 32--via--C
```

Bad data validation for city:



Run: planTrip

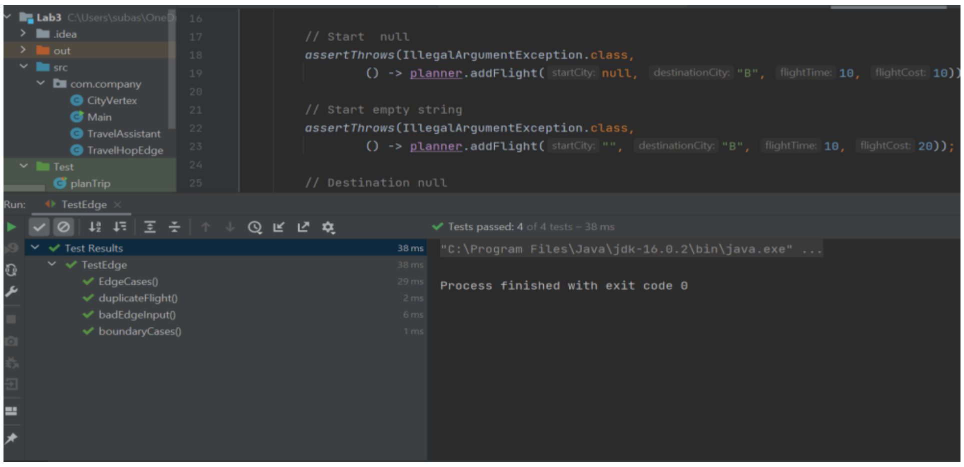
```
Test Results                    65 ms
  planTrip                      65 ms
    simpleGraph()               23 ms
    basicGraph1()               33 ms
    basicGraph()                 1 ms
    badDataForPLanTrip()         4 ms
    vaccinatedGraphNoPriority()  4 ms
```

Tests passed: 5 of 5 tests – 65 ms

Tests passed: 5

```
"C:\Program Files\Java\jdk-16.0.2\bin\java.exe"
[D, C2, C1, S]
start S
fly C1
fly C2
fly D
[D, C, A]
A -- 0--via--A
A----Flight----26----B--chosen--
A----Flight----16----C--chosen--
B -- 26--via--A
B----Flight----26----D
C -- 16--via--A
```

Bad data validation for :



```
        // Start  null
        assertThrows(IllegalArgumentException.class,
                () -> planner.addFlight( startCity: null,  destinationCity: "B",  flightTime: 10,  flightCost: 10))

        // Start empty string
        assertThrows(IllegalArgumentException.class,
                () -> planner.addFlight( startCity: "",  destinationCity: "B",  flightTime: 10,  flightCost: 20));

        // Destination null
```

Tests passed: 4 of 4 tests – 38 ms

```
Test Results          38 ms
  TestEdge            38 ms
    EdgeCases()       29 ms
    duplicateFlight()  2 ms
    badEdgeInput()     6 ms
    boundaryCases()    1 ms
```

```
"C:\Program Files\Java\jdk-16.0.2\bin\java.exe" ...

Process finished with exit code 0
```