|  |
| --- |
| **CSCI 3901** |
|  |
| Assignment 3 |
|  |

# Overview

The objective of this problem is to generate a travel path between two or more cities while considering various factors such as cost, time and ease of transit (the number of hops). The general idea is as follows: The path from a City – SOURCE to a City – DESTINATION is calculated through an adaption of Dijkstra’s algorithm. The adaption integrates various factors that are of importance to travelers and generates an optimal route corresponding to a traveler’s priorities.

The program functions as follows. Users can make use of commands to add a City, add a Flight path between two cities and add a Train path between two cities. Subsequently they can plan a trip between two cities, utilizing the information that was previously entered.

The relative cost of travel is used to determine the shortest path from SOURCE to DESTINATION. The relative cost of travel can be termed as the weight of each edge (path) between two vertices (Cities).

# Files & External Data

All input & output is generated through the console / terminal. There’re no Input / Output files.

# Data Structures & Classes

The approach used to implement Dijkstra’s algorithm primarily utilizes

1. **Priority Queue** - To sort Cities by ascending order of relative cost of travel
2. **HashMaps -** To store relative costs for each City, to store the departure city for each destination and to store the number of hops from SOURCE to DESTINATION
3. **ArrayLists -** To store Cities, Flights & Trains
4. **HashSet –** To store Vertices that’ve been visited (in Dijkstra’s algorithm)

Classes:

## City.java

This class represents Cities:

Each object of type City is associated with the following attributes:

Variables:

cityName - City name  
 testRequired - States if a test is Required for Unvaccinated individuals to the city  
 timeToTest - Time to Take a test  
 nightlyHotelCost - Hotel Cost for 1 night  
 cost\_from\_source - Cost to Travel from source to this City

## Flight.java

This class represents Flights between cities.

Each object of type Flight is associated with the following attributes:

Variables:

startCity - Start city  
 destinationCity - Destination city  
 flightTime - Time to travel  
 flightCost - Cost of travel

## Train.java

This class represents Trains between cities.

Each object of type Train is associated with the following attributes:

Variables:

startCity - Start city  
 destinationCity - Destination city  
 flightTime - Time to travel  
 flightCost - Cost of travel

## Hop.java

This class represents a path/edge between two cities.

Each object of type Hop is associated with the following attributes:

Variables:

start - Start city

destination - Destination city

relative\_cost - Relative cost of visiting city from the source

time - Time to travel

cost - Cost of travel

mode - Mode of travel (Fly / Train)

## Graph.java

This class represents a path/edge between two cities.

Each object of type Hop is associated with the following attributes:

Variables:

Vertices – A HashSet which stores all vertices (Cities)  
adjVertices – A HashMap which stores a set of Cities that are connected to each City. This serves as the adjacency list.

## TravelAssistant.java

This class represents a path/edge between two cities.

Each object of type Hop is associated with the following attributes:

Variables:

Cities – Stores a unique list of Cities

Hops - Stores a unique list of Travel Paths between Cities

Flights – Stores a unique list Flight paths between Cities

Trains – Stores a unique list Train paths between Cities

Methods:

## encode():

This method accepts an input file from the standard input, parses the contents of the file, encodes it and stores the output in another text file.

### Parameters:

**String input\_filename** - Path to the file which’s to be encoded

**int level** – The maximum number of characters which can be parsed in the last cycle is determined by 2^level.

**boolean reset** – If reset is true, frequency counts are reset at the beginning of every cycle.

**String output\_filename** - The path of the file, in which the output is stored.

# Assumptions

In addition to the assumptions stated under the section titled “Assumptions” in the document – CSCI 3901 Assignment 3, the following additional assumptions were made.

1. There cannot exist two identical modes of transport (with different time to travel / cost of travel) between two cities.
2. The Travel Assistant should not contain duplicate information for Cities.
3. The Time taken for a COVID 19 test can be zero.
4. Time to Travel, Cost to Travel and Hotel cost cannot be zero.

# Algorithm & Design

Approach to determining the shortest path between two cities:

## Relative Cost – Equation

**Relative Cost**= (cost Importance \* Total Cost Incurred) + (Time Importance \* Time to Travel) + (Hop Importance \* Number of Hops from SOURCE to Destination);

**Total Cost Incurred** = Cost of Travel + Cost to Test

**Cost to Test** = 0 [If a traveler is vaccinated]

**OR**

**Cost to Test** =(time to test (days) \* hotel cost) [If a traveler is unvaccinated]

**Number of Hops from SOURCE to Destination** = Number of Hops from Source to Previous City + number of Hops from Previous City to Destination

## Adaptation of Dijkstra’s Algorithm:

1. Obtain the list of Cities and the Start City.
2. Set the relative cost of travel from Start City to Start City as 0. Add the start city to the priority queue.
3. Set the relative cost of travel from Start City to other Cities as “Infinity”.
4. While the number of Visited Nodes is lesser than the number of Cities repeat steps 5-6.
5. Remove a **CityA** from the priority queue and add it to the set of visited Cities.
6. For every City **City1** that is connected to the visited CityA, repeat steps 7-10.
7. If a Traveler is unvaccinated and City1 requires unvaccinated travelers to be tested but doesn’t have facilities to conduct tests, set the relative cost to this City as “Infinity” and process the next City.
8. If a Traveler is unvaccinated and City1 requires unvaccinated travelers to be tested and has facilities to conduct tests, determine the cost of **testing** in City1.
9. Determine the relative cost of travelling to City1.
10. Compare this new cost to the existing relative cost of travelling to City1. If the new cost is lesser, update the relative cost of travelling to with the new cost.
11. Using the relative cost to travel to each City determine the path (The cities and the mode of travel to each city).
12. Display the path to the traveler.

# **References**

1. <https://www.geeksforgeeks.org/integer-max_value-and-integer-min_value-in-java-with-examples/> - How to assign a maximum / infinite value to a variable?
2. [What Is the Best Shortest Path Algorithm? | MyRouteOnline](https://www.myrouteonline.com/blog/what-is-the-best-shortest-path-algorithm)
3. [Graphs in Python: Dijkstra's Algorithm](https://stackabuse.com/dijkstras-algorithm-in-python/)
4. [Dijkstra Algorithm in Java | Baeldung](https://www.baeldung.com/java-dijkstra)
5. [Java Map - javatpoint](https://www.javatpoint.com/java-map)
6. [GitHub - vasuksh/Dijkstra-Algorithm-efficient-implementation: Using Dijkstra’s Algorithm to find the shortest path between two destination which will be used for flight/train bookings. Time complexity: O(V + E log(E))](https://github.com/vasuksh/Dijkstra-Algorithm-efficient-implementation)
7. [Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction](https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/)
8. [Throwing an exception in Java](https://www.javamex.com/tutorials/exceptions/exceptions_throwing.shtml)
9. [Keeping track of paths - Shortest paths with Dijkstra's Algorithm](https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/keeping-track-of-paths)

# Test Cases

1) Input validation tests:

1. **addCity()**
   * Null value passed as cityName
   * Empty string passed as cityName
   * Hotel Cost is lesser than zero
   * Hotel Cost is equal to zero
   * Hotel cost is greater than 2,147,483,647
2. **addTrain()**
   * Null value passed as startCity
   * Empty string passed as startCity
   * Null value passed as destinationCity
   * Empty string passed as destinationCity
   * startCity is equal to destinationCity
   * Time to Travel is lesser than zero
   * Time to Travel is equal to zero
   * Time to Travel is greater than 2,147,483,647
   * Cost of Travel is lesser than zero
   * Cost of Travel is equal to zero
   * Cost of Travel is greater than 2,147,483,647
   * Add a path that has the same start and destination cities as an existing path but has different parameters (cost of travel and time to travel)
3. **addFlight()**
   * Null value passed as startCity
   * Empty string passed as startCity
   * Null value passed as destinationCity
   * Empty string passed as destinationCity
   * startCity is equal to destinationCity
   * startCity doesn’t exist
   * destinationCity doesn’t exist
   * startCity & destinationCity do not exist
   * Add a travel path that already exists
   * Time to Travel is lesser than zero
   * Time to Travel is equal to zero
   * Time to Travel is greater than 2,147,483,647
   * Cost of Travel is lesser than zero
   * Cost of Travel is equal to zero
   * Cost of Travel is greater than 2,147,483,647
4. **planTrip()**
   * Null value passed as startCity
   * Empty string passed as startCity
   * Null value passed as destinationCity
   * Empty string passed as destinationCity
   * Cost Importance is negative
   * Time Importance is negative
   * Hop Importance is negative
   * Cost Importance is 0
   * Time Importance is 0
   * Hop Importance is 0
   * Cost Importance is greater than 2,147,483,647
   * Time Importance is greater than 2,147,483,647
   * Hop Importance is greater than 2,147,483,647
   * Plan Trip when there’re 0 Cities in the Travel Assistant

2) Boundary tests

1. **addCity()**
   * Single Character cityName
   * Multiple Characters in cityName
   * Add 1 City
   * Add Many Cities
   * Add a City when no Cities exist
   * Time to Test is negative
   * Time to Test is 0
   * Hotel Cost is 1
2. **addTrain()**
   * Single Character startCity name
   * Multiple Characters startCity name
   * Single Character destinationCity name
   * Multiple Characters as destinationCity name
   * Time to Travel is 1
   * Time to Travel is 2,147,483,647
   * Cost of Travel is 1
   * Cost of Travel is 2,147,483,647
   * Cost of Travel is greater than 2,147,483,647
3. **addFlight()**
   * Single Character startCity name
   * Multiple Characters startCity name
   * Single Character destinationCity name
   * Multiple Characters as destinationCity name
   * Time to Travel is 1
   * Time to Travel is 2,147,483,647
   * Cost of Travel is 1
   * Cost of Travel is 2,147,483,647
   * Cost of Travel is greater than 2,147,483,647
4. **planTrip()**
   * Single Character startCity name
   * Multiple Characters startCity name
   * Single Character destinationCity name
   * Multiple Characters as destinationCity name
   * Cost Importance is 1
   * Time Importance is 1
   * Hop Importance is 1
   * Cost Importance is 2,147,483,647
   * Time Importance is 2,147,483,647
   * Hop Importance is 2,147,483,647
   * Plan Trip when there’s a single City in the Travel Assistant
   * startCity name doesn’t exist in the Travel Assistant
   * destinationCity name doesn’t exist in the Travel Assistant
   * Both startCity name and destinationCity name do not exist in the Travel Assistant

3)Control flow tests

1. **addCity()**
   * Add a City with testRequired as True
   * Add a City with testRequired as False
   * Add a City with timeToTest > 0
2. **addTrain()**
   * Add a travel path when 2 Cities exist
   * Add a travel path when many Cities exist
3. **addFlight()**
   * Add a travel path when 2 Cities exist
   * Add a travel path when many Cities exist
4. **planTrip()**
   * Plan a trip with isVaccinated as True
   * Plan a trip with isVaccinated as False
   * Plan a trip with all relative weights as 0
   * Plan a trip with all relative weights as 1
   * Plan a trip with costImportance >1
   * Plan a trip with timeImportance >1
   * Plan a trip with hopImportance >1