PROJECT 3 – AIRLINE HUB MANAGER PROPOSAL

Somesh Herath Bandara

Gabriel Gloss

CSC316: Data Structures for Computer Scientists

sgherath@ncsu.edu

gkgloss@ncsu.edu

North Carolina State University

Department of Computer Science

October 25th, 2017

Black Box Test Plan

The following file, "airports.txt", exists in the **input** program directory and will be used for the black box tests:

```
AIRPORT_CODE,LATITUDE,LONGITUDE
SIN,1.35019,103.994003
FRA,50.0333333,8.5705556
LHR,51.4706,-0.461941
YYZ,43.6772003174,-79.63059997559999
IAH,29.984399795532227,-95.34140014648438
```

Another file, "shortFlight.txt", exists in the **input** program directory with the following contents:

```
AIRPORT_CODE,LATITUDE,LONGITUDE
SIN,1.35019,103.994003
FRA,50.0333333,8.5705556
```

AirlineHubManagerUI.java starts the program.

Test ID	Description	Expected Results	Actual Results
testFilenames **(ECP – inputting an invalid filename followed by a valid filename)**	**Preconditions:** • The program AirlineHubManagerUI has started successfully. • The file "badName.txt" does *not* exist. • The file "input/airports.txt" exists. **Steps:** 1. User enters "badName.txt" 2. User enters "input/airports.txt"	*Please input the airport information filename:* 1. **The file "badName.txt" could not be found. Please try again:** 2. **File "input/airports.txt" successfully loaded.**	
testHelp **(ECP – inputting commands into the program)**	**Preconditions:** • The program AirlineHubManagerUI has started successfully. • The user has	*Please input a command:* 1. **The following commands are available: help, quit, report, list** *Please input a command:* 2. **Exiting program**	

	successfully loaded the "input/airports.txt" file into the program. **Steps:** 1. User enters "help" 2. User enters "quit"	**cleanly.**	
testList **(BVA – check a small file that has less than 3 airports, so no hub is possible)**	**Preconditions:** • The program AirlineHubManagerUI has started successfully. • The user has successfully loaded the "input/shortFlight.txt" file into the program. **Steps:** 1. User enters "list"	*Please input a command:* 1. **FlightList[Flight[airport1=SIN, airport2=FRA, distance=6387.2]]**	
testBadCommands **(DT – entering bad commands)**	**Preconditions:** • The program AirlineHubManagerUI has started successfully. • The user has successfully loaded the "input/airports.txt" file into the program. **Steps:** 1. User enters "badCommand" 2. User enters "12345" 3. User enters "abc_!@#123" 4. User enters "quit"	*Please input a command:* 1. **Not a valid command! Please try again:** 2. **Not a valid command! Please try again:** 3. **Not a valid command! Please try again:** 4. **Exiting cleanly.**	
testReport **(BVA – generate list when no airports can be considered as hubs)**	**Preconditions:** • The program AirlineHubManagerUI has started successfully. • The user has successfully loaded the "input/shortFlight.txt" file into the program. • None of the airports	*Please input a command:* 1. **No airports have at least 3 connecting flights.**	

	in the file can be considered a hub **Steps:** 1. User enters “report”		

Algorithm Design

Algorithm getPossibleHubs(Graph G)
   Input *a Graph, g, that contains the flights that minimize distance required to connect all airports (given as an adjacency list)*
   Output *the list of airports that are involved in at least 3 flight connections*

```
LinkedList out <- new LinkedList
for all vertices in G
   // unique means not checked before
   for all unique neighbors of each vertex
      G.currentVertex.airport.hubCount++
      G.neighborVertex.airport.hubCount++
   end for
   if G.currentVertex.airport.hubCount >= 3
      out.insert(new ListNode(G.currentVertex.airport))
   end if
end for
return out
```

DATA STRUCTURES

For this project, we will use the following data structures:

- **edges**: an adjacency list representing the flights that minimize connections between AirPorts. Each vertex represents in this Graph ADT represents an airport, and an edge weight of this graph ADT holds the distance to another vertex (airport). Created when reading input files.
- **hubList**: a LinkedList of AirPorts with 3 or more connections. Created by the getPossibleHubs method that iterates through edges and inserts AirPorts with 3 or more connections to hubList.

edges is required to insert, remove, and get vertices and edges to an adjacency list. The adjacency list will be implemented using a LinkedList. See below for more details about the LinkedList structure. The following operations will be required of **edges**:

- **getNeighbors(Vertex)**: Gets all the neighbors of a vertex (vertices with edges to this vertex)
- **addVertex(Vertex)**: Adds the specified vertex
- **removeVertex(Vertex)**: Removes the specified vertex
- **getEdge(Vertex, Vertex)**: Gets edge value between two vertices
- **addEdge(Vertex, Vertex)**: Adds an edge between two vertices
- **removeEdge(Vertex, Vertex)**: Removes an edge between two vertices
-

An alternate data structure for **edges** that ensures similar running time would be an adjacency matrix.

Operation	LinkedList	ArrayList
getNeighbors(Vertex),getEdge(Vertex,Vertex)	O(n)	O(n)
removeVertex(Vertex), removeEdge(Vertex,Vertex)	O(n)	O(n)
addVertex(Vertex), addEdge(Vertex,Vertex)	O(1)	O(1)

hubList is required to insert AirPorts to a LinkedList, and get AirPorts contained in a LinkedList. Airports are contained within ListNode objects. Therefore, the LinkedList **hubList** will be designed to use the following operations:

- **insert(ListNode)**: Adds a ListNode to a LinkedList of AirPorts, at the end
- **getNext()**: Returns the next ListNode in a LinkedList of AirPorts, using an iterator

An alternate data structure for **hubList** that ensures similar running time would be an ArrayList made of Airport objects.

Operation	LinkedList	ArrayList
insert(ListNode)	O(1)	O(1)
getNext()	O(1)	O(1)

ALGORITHM ANALYSIS

The estimated running time T(n) for the getPossibleHubs(Graph G) algorithm is dependent on the number of vertices in the adjacency list. We will denote the number of vertices as *n*. Then we can show that T(n) = $O(n^2)$, which is overall $O(n^2)$ as shown below using the limit test:

Line Number	Code	Number of Operations
1	*LinkedList out <- new LinkedList*	1
2	*for all vertices in G*	n
3	*for all unique neighbors of each vertex*	n
4	*G.currentVertex.airport.hubCount++*	1
5	*G.neighborVertex.airport.hubCount++*	1
7	*if G.currentVertex.airport.hubCount >= 3*	1
8	*out.insert(new ListNode(G.currentVertex.airport))*	1
11	*return out*	1
Total		$2(n^2 + n + 1)$

$f(n) = 2(n^2 + n + 1)$

$g(n) = n^2$

According to the limit test:

$$if \lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & then\ f(n)\ is\ O(g(n)),\ but\ g(n) is\ NOT\ O(f(n)) \\ \infty & then\ g(n)\ is\ O(f(n)),\ but\ f(n)\ is\ NOT\ O(g(n)) \\ c, 0 < c < \infty & f(n) is\ O(g(n))\ and\ g(n) is\ O(f(n)) \end{cases}$$

For our algorithm: $\lim_{n\to\infty} \frac{2(n^2+n+1)}{n^2} = 2$

Since 2 is a constant, then T(n) = $O(n^2)$ by the limit test.

SOFTWARE DESIGN

Our software design implements a simple command line interface to interact with the input data from the file. The MVC (Model-View-Controller) programming pattern will be used in our software design with the AirlineHubManager class behaving as the model and the controller. The command line interface provided by the shell will act as the view. Implementing MVC friendly code will help us separate the functionality of each of our classes and make thinking about programming each class easier. Additionally, we will use the Singleton design pattern for our AirlineHubManager class. This means that only one instance of our manager class will be running at any time. Using Singleton helps greatly when thinking about the programming and helps allow for mutual exclusion.