Name: Thomas Kojo Quarshie

ID: 43102024

First off, I started my work by creating a class for each csv file (i.e., Airport, Airline, and Routes) with the columns in each file as the instance variable. This was a very good head start for me because I was able to create getter methods to access the records in the file easily. In reading the files into an ArrayList using the BufferedReader object, I realized that the existence of extra commas in some records influenced their columns to increase more than the number of columns in a table. This problem persisted particularly in the Airport file. Therefore, in fixing this issue, I created a method that returns all Airport objects that have their length greater than 14 (which is the original number of columns in the Airport file). With this implementation, I was able to see where the extra commas occurred in the record and created a constraint based on the specific Airport ID. For instance, I combined the second and third column as one for ID's that had extra commas in the Airport name. Moreover, I implemented a Route distance method that takes the "SourceAirportcode" and "DestinationAirportcode" and computes the distance between them using the Haversine formula referenced from Jason Winn on GitHub. This was very important to be implemented because I incorporated it in my search algorithm to compute the path cost from one "SourceAirportcode" to its succeeding city. In my Search algorithm I applied the Hashmap data structure which was efficient in keeping track of the "SourceAirportcode" as my Key and the values as an ArrayList having the "DestinationAirportcode" and path cost. This way, it was easier to access the destination from a given starting point and it's resulting path cost with the use of a solution path method. My Search was an implementation of the uniform cost search algorithm which applied the Priority Queue data structure as the frontier. This made my search efficient in the sense that the node with the highest priority will be popped from the frontier first to explore its successor nodes. Moreover, the search algorithm returns the optimal path by comparing the new path cost it has found to its previous. If the path cost is less than the previous and that node is the goal, then it returns it as the solution. With this individual project, I have really acknowledged how important the design and data structures used in one's code make it either efficient, optimal or not. Finally, I have also learnt to take my time and use a UML diagram to give me an overview on how I want to implement my solution to reduce stress and confusion.