Outline of Problem

For this project we were tasked to “calculate the best round trip route (cheapest cost) for an itinerary returning to the home airport using each airport once (5 airport solution)”.

While this may seem like an easy problem on the surface there are a number of key parameters which must be taken into consideration. For example, we also need to consider different fuel costs in different airports, whether an aircraft has sufficient fuel capacity to make a journey, what model the aircraft is (which determines its fuel capacity), and how to calculate the distance between two airports.

Solution Design

The design of this solution hinges around a number of key components, namely class structure for organisation, understanding and reuse; appropriate use of data structures to match a given task, and opting to use a sufficient greedy algorithm to find the cheapest route instead of opting for an optimal (yet less efficient) solution.

Class Structure and associated Data Structures

To address different areas of the problem I decided to group my code into classes where each class would have a specific role when it came to implementing the solution. Making the code more modular in this way made it easier for me to understand the problem. It also provided an interface for solving the sub problems in this project, which made my main.py program easier to develop and much neater. Finally, it also made for easier testing as I could easily distinguish individual functions for testing. An overview of the classes I implemented for this project is given below

Airport.py

This class allowed for important details about a given airport to be stored in the attributes of an instance of the class. The most important attributes were airport code, latitude, longitude and country. The class also had associated getter methods to allow for easy access to these attributes.

AirportAtlas.py
This class parses a csv file (in this case airport.csv) containing information about a number of airports. An Airport object is created using Airport.py for each of these classes which are then stored in a dictionary attribute of the AirportAtlas object called Atlas. The code for the airport is used as a key and using this python dictionary, or hash table, allows for efficient access to these airport objects with an average time complexity of O(1).

Aircraft.py

This class allowed for important details about a given aircraft to be stored in the attributes of an instance of the class. The most important attributes were aircraft code, and fuel capacity.

AircraftTable.py

This class parses a csv file (in this case aircraft.csv) containing information about a number of aircraft. An Aircraft object is created using Aircraft.py for each of these classes which are then stored in a dictionary attribute of the AircraftTable object called Table. The code for the aircraft is used as a key

and using this python dictionary, or hash table, allows for efficient access to these airport objects with an average time complexity of O(1).

Currency.py

This class parses two csv files (in this case countrycurrency.csv and currencyrates.csv) which contain information about the currency in a given country and the to EUR rate for that currency. Using the country attribute of an Airport object we are able to use a Currency object to get the rate for this country. This is important as this is the rate used to calculate fuel prices in this project.

GraphDirected.py

Using this class I implemented a generic weighted graph data structure using a dictionary to store the vertices and their edges/weights. It also uses a set to store the vertices as the order of the vertices is not important to us (this is captured in the graph itself). An example of the graph is given below with fictional weights

{'DUB', {'ORK': 50, 'SNN': 300},
'ORK', {'DUB': 300, 'SNN': 100},
'SNN', {'DUB': 400, 'ORK': 100}}

So, for example, the weight on the edge from DUB to ORK is 50 whereas the weight on the edge form ORK to DUB is 300.

The graph data structure allows for a very nice way to visualise our problem and it also has the advantage of having being studied extensively in mathematics and having a number of useful algorithms associated with it.

GraphRouteConstructor.py

This graph class extends the GraphDirected class above. It is specific to the problem being considered and has a number of associated methods for this. First of all, when an instance of it is created it uses a breadth first search algorithm to construct the graph from a given route itinerary. The starting point of the itinerary, the airports in the route and the aircraft code for the journey are all also stored as attributes.

This class also has a shortest path method, which is discussed in the algorithms section.

Itinerary.py

This class will parse a given itinerary csv file containing 6 columns of data, where the first five columns are be valid three letter airport codes and the last column is be an aircraft code. Instances of this class will be implemented using a nested Python list (dynamic array) containing Python lists of each row in the itinerary. This data structure is sufficient for the purposes of this project as these itineraries will mainly be iterated over and accessed and access is O(1) for a python list.

Algorithm Design

As mentioned above the algorithm I implemented to find the cheapest path was a greedy algorithm that traded an optimal solution for a much faster running time. An optimal solution which calculates all of the possible solutions will be O(n!) whereas the algorithm I used is O(n^2).

I implemented the algorithm as a method of he GraphRouteConstructor class and it functions as follows:

At each step in traversing the graph the neighbour which has the lowest weighted edge and has not yet been visited is moved to (i.e. set to current) This neighbour is also added to the path and the cost is added to the total cost. The neighbour which has now been visited no longer needs to be visited.

A simplified version of the algorithm below can help show how this is O(n^2):

```
While there are nodes to visit ##n-1 times (start already visited)
    for node in neighbours check if it's the cheapest node  ## n-1 times
        if node is closest
            move to this node
            remove node from nodes to be visited
```

This is O((n-1)^2) and therefore O(n^2). I believe that this could also be made more efficient by not checking the neighbours that have already been visited in the for loop.

Note: I did not use a queue when building the path in my implementation as I was only using a string representation of the path when outputting the data to bestroutes.csv