COMP20230 Data Structures & Algorithms – Term Project Report

17205389 - Daniel O'Byrne

Function Specification

My solution to the presented problem takes a brute force approach to calculate the cheapest route. This route is calculated with a given a start/end destination, 4 stops that must be visited and an aircraft type which has a range in kilometres. The program makes this calculation in the following manner:

- The CSV files for airport, currency and aircraft information are read in. They are used to create instances of the AirportAtlas, CurrencyAtlas and AircraftAtlas classes, respectively.
- Each of these classes creates dictionaries that hold all the necessary information required to calculate the cheapest route. These dictionaries contain objects, which are instances of the Airport, Currency and Aircraft classes. These classes have the relevant getter methods to extract this information.
- A test routes CSV is then read in which gives stops in a route and the aircraft to be used. Blank spaces in this file will be flagged and those particular lines will be skipped. For every route, a dictionary is created, called comboKeys. This holds all possible routes available for the given airports. The dictionary has 4 keys. Each key is a possible second stop on the route (as the first stop is fixed). The values associated with each key is every possible combination of the entire journey, where the second stop on the route is always the key. For example:
    - `{'BBO': ['LOV', ('BBO', 'BHI', 'VKO', 'VCS'), 'LOV', 'LOV', ('BBO', 'BHI', 'VCS', 'VKO'), 'LOV'`...
    - This is the beginning of an example the comboKeys dictionary. The key is one of the route options, 'BBO'. The value is every possible route option of the given route, where 'BBO' is the second stop, as 'LOV' is always the start/end stop on the route.
- The program then slices this value to extract every route option. In this case the first route option would be: `'LOV', ('BBO', 'BHI', 'VKO', 'VCS'), 'LOV'`
- For every leg of the journey, the distance is checked against the range of the aircraft, to check if it can reach. If it can, the cost of this leg is calculated. Otherwise, the program will print that it is not possible.
- If each leg is possible, the total cost of the trip is stored. This process is repeated until all possible combinations of the journey have been validated and their cost calculated. Each time a route option is valid, it is compared to the lowest previous total cost. If a lower cost is found, it becomes the new lowest cost, and the route becomes the new lowest cost route.
- Thus, at the end of this process, the cheapest total cost will be returned, with the associated route option.
- This result will be printed to the console for the user to see. The program also takes this results and formats it in such a manner that it writes to a 'bestroutes.csv' in a neat fashion.
- If the route is not feasible, it will print appropriately to the console, and the 'bestroutes.csv'.

Design

This task was very much an iterative process of testing methods and verifying whether they made sense in terms of the specifications. Upon initial inspection of the problem presented, I began to create an algorithm that somewhat represented a Dijsktra's shortest path algorithm. Each route that was created was done so by calculating the cheapest next stop. With each stop chosen, this would be appended to a flight plan, until I had a full route. This however, was not representative of Dijsktra's, and was more of a breadth first search algorithm. Either way after its implementation, I felt it was not the right approach. I changed tact and opted for a brute force algorithm.

I felt for the purpose of this assignment, accuracy was more important than efficiency. This brute force algorithm, although economically not as efficient as Dijsktra's, would provide highly accurate results, returning the cheapest route every time. However, I can appreciate the usefulness of Dijkstra's solution. For this problem, the algorithm would continuously calculate and update the cheapest route from the source to the destination, through all other nodes. Pseudocode to implement this strategy:

```
function Dijkstra(Graph, source):
    for each airport v in Graph:                    // Initialization
        cost[v] := infinity                         // initial cost from source to airport v is set to infinite
        previous[v] := undefined                    // Previous node in optimal path from source
    cost[source] := 0                               // Cost from source to source
    Q := the set of all airports in Graph           // all nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                           // main loop
        u := node in Q with cheapeast cost[ ]
        remove u from Q
for each neighbor airport v of u:                   // where v has not yet been removed from Q.
            alt := cost[u] + cost_between(u, v)
        if alt < cost[v]                            // Relax (u,v)
                cost[v] := alt
                previous[v] := u
    return previous[ ]
```

Adapted from source [1]

Going ahead with the brute force approach, for each of the CSV files, I created a class to deal with their implementation. Each class creating relevant dictionaries. For example, in the Airport and AirportAtlas classes respectively:

```
def __init__(self, code, country, airportName, airportLat, airportLong ):
    # the 5th column in the csv is the three letter airport code, index 4
    self.code = code
    self.country = country
    self.airportName = airportName
    self.airportLat = airportLat
    self.airportLong = airportLong
```

```
def loadData(self, csvFile):
    with open(os.path.join(csvFile), "rt") as f:
        reader = csv.reader(f)
        for line in reader:
            self.codes[line[4]] = Airport(line[4], line[3], line[1], line[6], line[7])
```

(*source*[1] = http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html)

For each airport in the CSV, I create an entry in the dictionary, 'codes'. This dictionary has keys which are the three-letter code that identify each airport, and the corresponding values are objects of the Airport class.

The reason I use a dictionary ADT to store this airport information, is two-fold:

- The three letter codes of each airport should unique. Therefore, setting this code as the key means that this uniqueness criteria is satisfied, as no two keys can be the same in a dictionary. Thus, using a dictionary validates the input values from the CSV and ensures there are no two codes the same.
- Economic sense: As the input file to test this code would have to retrieve information for airports at random, using a key value relationship to store the data, made the most economical sense. If these key value pairs were stored in a list, and the required airport was at the end of the list, the entire list would have to be traversed to extract the necessary information. This ADT would result in O(n) complexity. Using a dictionary ADT results in a complexity of O(1) as the required information can be directly retrieved using a key.

I use this exact technique to create dictionary ADTs for the aircraft and currency information, for the same reasons outlined above.

For the currency dictionary I have set the keys as country name. This allows me to extract relevant currency data using the name of a country. When the three-letter airport code is provided, my getter methods in the Airport class retrieve this country name, which is used to access the currency data:

- `countryName = airportDict.codes[option1[i]].getCountry()`
  - airportDict is the instance created of the AirportAtlas class
  - codes is the aforementioned dictionary, holding all airport data
  - option1 is a sequence of airports, that make up a possible route
- `euroRate = currencyDict.currencies[countryName].getToEuroRate()`
  - currencyDict is the instance created of the CurrencyAtlas class
  - Currencies is the dictionary holding all currency data

Once I created these dictionaries, I had to use their data to assess cheapest route of a given set of airports and an aircraft. As the first airport in the input is the home/finish stop on every possible route, I had to consider the other four airports. For my brute force algorithm to work, I would have to create every possible combination of these four airports. Firstly, I needed a data structure to store these four airports. The order of these airports does not matter, as I want to check every possible combination. Thus, I thought a queue ADT would be appropriate. I could access an airport from the head of the queue, then dequeue it when I was finished with its use.

Pseudocode:

```
route = Queue([stops[1], stops[2], stops[3], stops[4]])
    # use the head of the queue
route.get(head) # remove the head
```

However, I soon realised my method of considering every possible combination of these four stops, did not support the use of this type of queue. This type of queue, only accesses elements at the head and tail as this is where elements are dequeued and enqueued. As I used the itertools module to create each permutation of the routes, it needed to iterate over each stop, not possible with this queue. Thus, I kept these stops in a list (array).

```
route = [stops[1], stops[2], stops[3], stops[4]]
```

I did however treat it like a queue, in checking route validity. Once I was finished considering one leg of the journey, I would remove the head of the array. I did this until the array was empty i.e. I had checked the validity and cost of each leg. This treatment showed queue like behaviour:

```
# while the queue is not empty, use the head of the queue to carry out the below
while len(route) != 0:
    # check if leg is valid, and get cost using the head and next element...

# when finished with the top element of the queue, remove it
route.remove(route[0])
```

With every possible route, my next design decision was how I was going to store one possible combination. With one possible route I would validate stop to stop distance, then calculating the associated costs. I used a list for this purpose. As list is a sequence type of data structure, this choice made sense. The order of the stops here matters as the program assesses the validity of a particular route. Thus, a list was more appropriate than a set for example, where the order does not matter.

```
option1 = list([option[0], option[1][0],  option[1][1],  option[1][2], option[1][3], option[2]])
# this is a list (sequence) as the order matters
```

With the list, I could then iterate over each element, checking the distance and cost of each leg:

```
for i in range(0,len(option1)-1):
    # check distance validity and check associated cost
```

In terms of data structures, there were no more decisions to be made. The rest of the code is focused around storing the lowest cost/route and formatting the output in such a way that it prints neatly to the CSV.

Testing

In order to validate my code/functions I created and applied test throughout the code writing process. Each time I added functionality to my solution, I would ascertain the output versus what I knew to be true. I have created four test files, one for each of the functional code files. In each, I have tested the necessary functions with assertion statements. I did not test all of the getter methods as I found it unnecessary. These methods are tested as part of other functions anyway. My tests ensure input validities, correct dictionary creation, calculation of cheapest route, output validity and more. See appendix for successful test outcomes.

Features to be implemented

Add visualisation of routes using graphs or maps: I have included a Graph.py file which was taken from the file provided. If I had more time I would have tried to implement this approach to show a graph of a single route.

Conclusion

This project has been extremely valuable in my understanding of data structures and algorithms, and their implementation in python. Throughout this module I have gained knowledge about different data structures and when they are most appropriately used. However, at the time of learning, it was somewhat difficult to envisage how they could be used in a practical sense. This project cleared a lot of these issues up for me as I witnessed and understood fully how to implement said data structures. This project has also allowed me to gain further skills and experience in python. My OOP python knowledge before this project was limited. However, having completed the project, I feel much more proficient using python. This will no doubt help me in this summer's research practicum.

Appendix

Tests:

```
obyrned1@obyrned1-Aspire-V5-122P:~/eclipse-workspace-comp30670/DS/Term_Project$ py.test --verbose testAirport.py
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34, pluggy-0.4.0 -- /home/obyrned1/anaconda3/bin/python
cachedir: .cache
rootdir: /home/obyrned1/eclipse-workspace-comp30670/DS/Term_Project, inifile:
collected 5 items

testAirport.py::testLoadData PASSED
testAirport.py::testGreatCircledlist PASSED
testAirport.py::testgetDistanceBetweenAirports PASSED
testAirport.py::testCostOfTrip PASSED
testAirport.py::testPossibleCombinations PASSED

========================== 5 passed in 6.79 seconds ==========================
obyrned1@obyrned1-Aspire-V5-122P:~/eclipse-workspace-comp30670/DS/Term_Project$ py.test --verbose testCurrency.py
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34, pluggy-0.4.0 -- /home/obyrned1/anaconda3/bin/python
cachedir: .cache
rootdir: /home/obyrned1/eclipse-workspace-comp30670/DS/Term_Project, inifile:
collected 2 items

testCurrency.py::testJoinCsv PASSED
testCurrency.py::testLoadData PASSED

========================== 2 passed in 2.05 seconds ==========================
obyrned1@obyrned1-Aspire-V5-122P:~/eclipse-workspace-comp30670/DS/Term_Project$ py.test --verbose testAircraft.py
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34, pluggy-0.4.0 -- /home/obyrned1/anaconda3/bin/python
cachedir: .cache
rootdir: /home/obyrned1/eclipse-workspace-comp30670/DS/Term_Project, inifile:
collected 2 items

testAircraft.py::testLoadAircrafts PASSED
testAircraft.py::testGetRange PASSED

========================== 2 passed in 0.60 seconds ==========================
obyrned1@obyrned1-Aspire-V5-122P:~/eclipse-workspace-comp30670/DS/Term_Project$ py.test --verbose testMain.py
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34, pluggy-0.4.0 -- /home/obyrned1/anaconda3/bin/python
cachedir: .cache
rootdir: /home/obyrned1/eclipse-workspace-comp30670/DS/Term_Project, inifile:
collected 1 item

testMain.py::testMain PASSED

========================== 1 passed in 2.79 seconds ==========================
```