Saf Flatters

2024

*This project delivers a fully interactive airline route management system, built from scratch in Python using custom graphs, heaps, hash tables, and BFS-based search algorithms—designed for speed, scalability, and a real-world user experience.*

AIRLINE MANAGEMENT SYSTEM

**Design & Implementation Report**

**Contents**

Introduction

Implementation Strategy

* Interactive Menus
  + Login Menu
  + Admin Menu
  + Customer Menu

Modules / Classes

* Graphs: Airports and Flight Paths
  + Vertices
  + Edges
  + Adjacency List
* HashTable: Airport Lookup
  + Hash
  + Double Hash
  + Good Hash Function Properties
* Route Search
  + Flowchart
  + Process
* BFS Search
  + DestBFSbyLay and DestBFSbyDest
  + BreadthFirstSearch
  + Export to filteredroutes.csv
* Heapsort
  + HeapSort
  + Heapify
  + Export to heaproutes.csv

Challenges

* Weighted Edges,
* BFS Infinite Loops

Efficiency

* Data Structures
* Sorting Algorithms (ADDITIONAL MARKS)
  + Algorithm Features
  + To Test
  + Output
  + Results

Future Improvements/Alternatives

* Double Hashing Options
* Simplify Breadth First Search
* Advanced Sorting

**Introduction**

This project presents a custom-built airline route management system developed in Python, simulating the operations of a travel agency, *AnywhereButThere*. It features interactive menus for both administrative and customer tasks, with key functionalities supported by self-implemented data structures including graphs, hash tables, heaps, stacks, queues, and linked lists.

The system enables route searches by distance or layovers, dynamic airport and flight path management, and efficient data retrieval. This report details the design approach, key modules, challenges encountered, performance considerations, and areas for future development.

**Implementation Strategy**

To divide this large project up into workable sections, I created a list of tasks I wanted the User to be able to do matching them with the assignment requirements. This list ultimately became Interactive Menu items, split between the Admin Menu and the Customer Menu.

**Interactive Menus**

Login Menu:

To ensure usability, users have the option to select from menus. In the Login menu, users can choose whether they want to perform Admin style tasks or Customer style tasks. This login also provides 2 AUTOINSERT options (one with a simple 5 airports and 7 flightpaths, the other with 36 airports and 114 flightpaths).

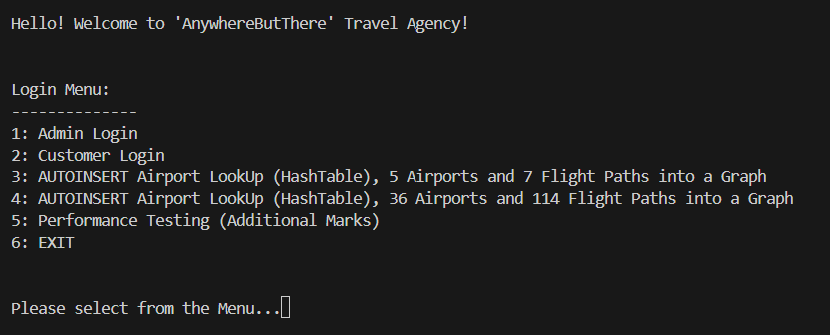


Figure 1: Login Menu

Admin Menu:

Inside the Admin menu, users can choose AUTOINSERT options again including being able to create a Hashtable separate to creating the graph. Other options include displaying the created hashtable (“Display Airports”), displaying an adjacency list (of created vertices), adding an airport into the graph and hashtable, adding a flight path (edge) to the graph, searching the hashtable with a key (3 letter airport code), deleting an airport (the vertex and associated hashtable item) or deleting a flightpath (an edge in the graph).

A computer screen with white text

Description automatically generated

Figure 2: Admin menu

Customer Menu:

Inside the Customer menu, a user has the option to search for a route by maximum distance or by maximum layovers. Please note that before the route options are displayed, the user will be prompted to select whether they want the results sorted by layovers or distance.

A black screen with white text

Description automatically generated

Figure 3: Customer Menu

The customer also has the options before choosing a departure and destination airport to display available locations. This matches each graph label (airport code) to a hashtable key to retrieve it’s hashtable value.

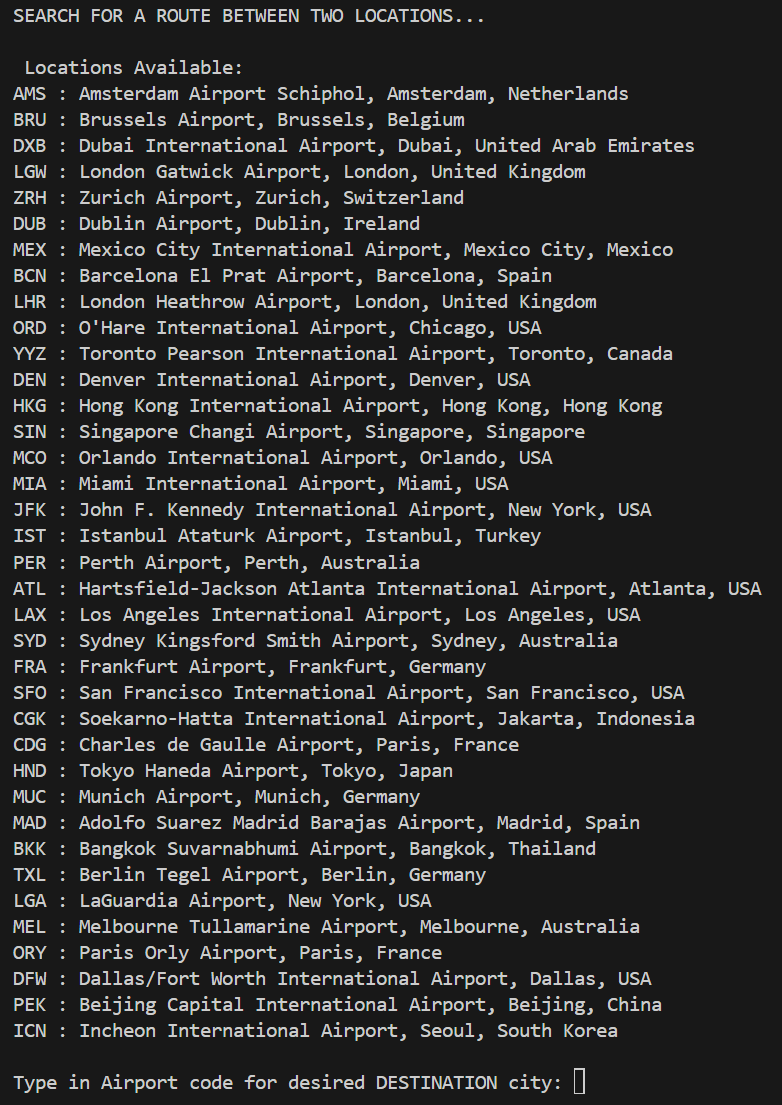


Figure 4: Available Locations provided to user during route search

**Modules / Classes**

**Graphs: Airports & Flight Paths**

This program comes with two pre-made airport networks that can be imported into a graph by selecting the option in the Login Menu. These can be auto inserted by the user (they can also be made manually). “SIMPLEAirportTestData.csv” contains five airports and seven flight paths. For visualisation, this report will use this one as an example. “AirportDataTest.csv” contains 37 airports and 114 flight paths.

Vertices:

A vertex in a graph represents each airport with the label being the airport code. Each vertex had a linked list with other airport codes that were connected to this airport by a flightpath (edge) and the distance of that flightpath.

Edges:

An edge in a graph represents each (bi-directional) flight path between airports. These edges were weighted by a distance amount. Each edge is added by being inserted into both vertices Edgelist (linked list).

A group of white circles with black text

Description automatically generated

Figure 5: Visual representation of graph (with Edge Linked Lists) created with SIMPLEAirportTestData.csv – image made with drawio

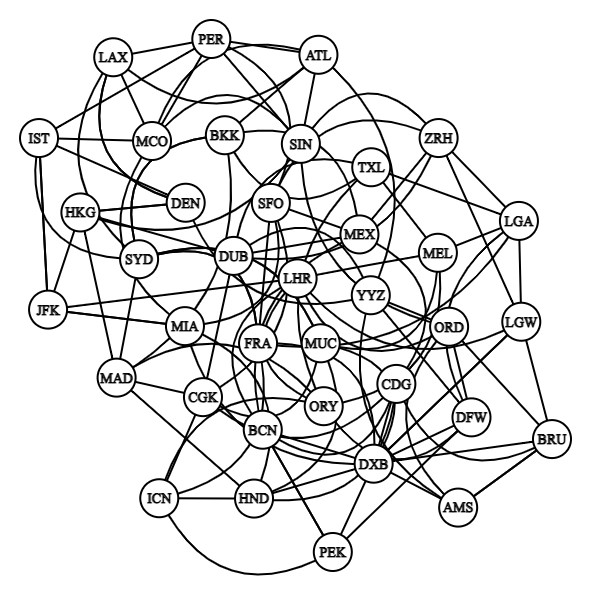


Figure 6: Visual representation of graph created with AirportDataTest.csv – image made with https://csacademy.com/app/graph\_editor/

Adjacency List:

This is an example of the output created by the SIMPLEAirportTestData.csv when User selects DISPLAY: Adjacency List in the Admin Menu.

A screenshot of a computer program

Description automatically generated

Figure 7: Output when AUTOINSERT is used, displaying Adjacency List

**Hashtable: Airport Lookup**

A hashtable was used to create a searchable array of airports in which it’s key is the three-letter airport code, and its value was the associated details of that airport, such as airport name, city and country it resides in. The location details to be hashed is pre-made, and all 37 items are hashed using HashClass.py functions. The key for all items matches the pre-made vertex labels. “AirportLookup.csv” and “AirportLookup2.csv” are read by the HashClass. I made them two separate files because the Hash Table is created by the total number of items in a csv file multiplied by 1.5. By reading in one file at a time, the Hash Table is forced to resize once the load factor reaches 75% and all items must be rehashed before continuing.

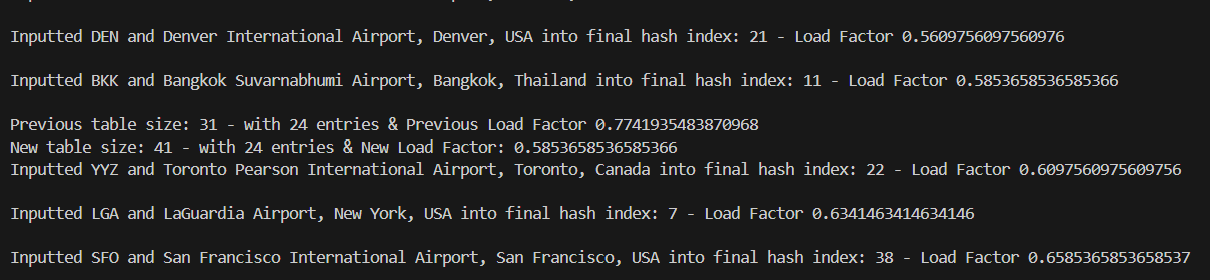


Figure 8: Resizing Hash Table when hashing AirportLookup.csv and AirportLookup2.csv

Hash:

Items are hashed by: the sum of ascii worth of all characters % 1.5\* size of hash table rounded up to nearest prime. Eg.

Ascii of MEL = 222, 222 modulus (size of table = 20, so 31) is 222 % 31 = 5



Figure 9: Example of hashed item - MEL

Double Hash:

If an item is hashed and the hash index is already taken, item is then doublehashed. This uses \_stepHash function and is doublehashed by: ascii worth of final character multiplied by amount of times it has been hashed modulus size of hashtable to nearest prime. Eg.

LGW = Ascii of W = 87, multiplied by 1 (first try), = 87 modulus 31 = 25 (25 is taken)

LGW = Ascii of W = 87, multiplied by 2 (second try), 174 modulus 31 = 19

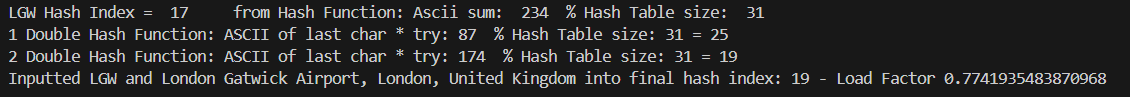


Figure 10: Example of doublehashed item - LGW

*Note: The printouts for these items are commented out in the code to reduce clutter of printout display*

Good Hash Function Properties:

1. Hash indexes must fit table – using the modulo of the nearest prime of the table size ensured that the hash indexing fit within the table.
2. Be fast to compute – through trial and error, I measured my double hashing options by making my code print out the maximum amount of double hashes that were required for all items and then used the technique that required the least.
3. Repeatable – this was tested by making sure get() would retrieve the correct item everytime. This proves the algorithm is repeatable and results in the same index every time.
4. Distributing Keys evenly – this was somewhat successful. I found that I would have some clustering in areas and with further research could spread hashed items more evenly.

A screen shot of a computer program

Description automatically generated

Figure 11: Entire Autoinserted Hashtable displayed to show clustering

The hashtable created is used extensively throughout the program to display airports as a list and also used to retrieve single items as a confirmation of users airport code input and also to display route search results.

**Route Search**

To ensure that all the user functionality specified by the assignment brief when searching for a route between two airports is covered, I created a flow chart. This flow chart was altered many times during the implementation process.

Flow Chart:

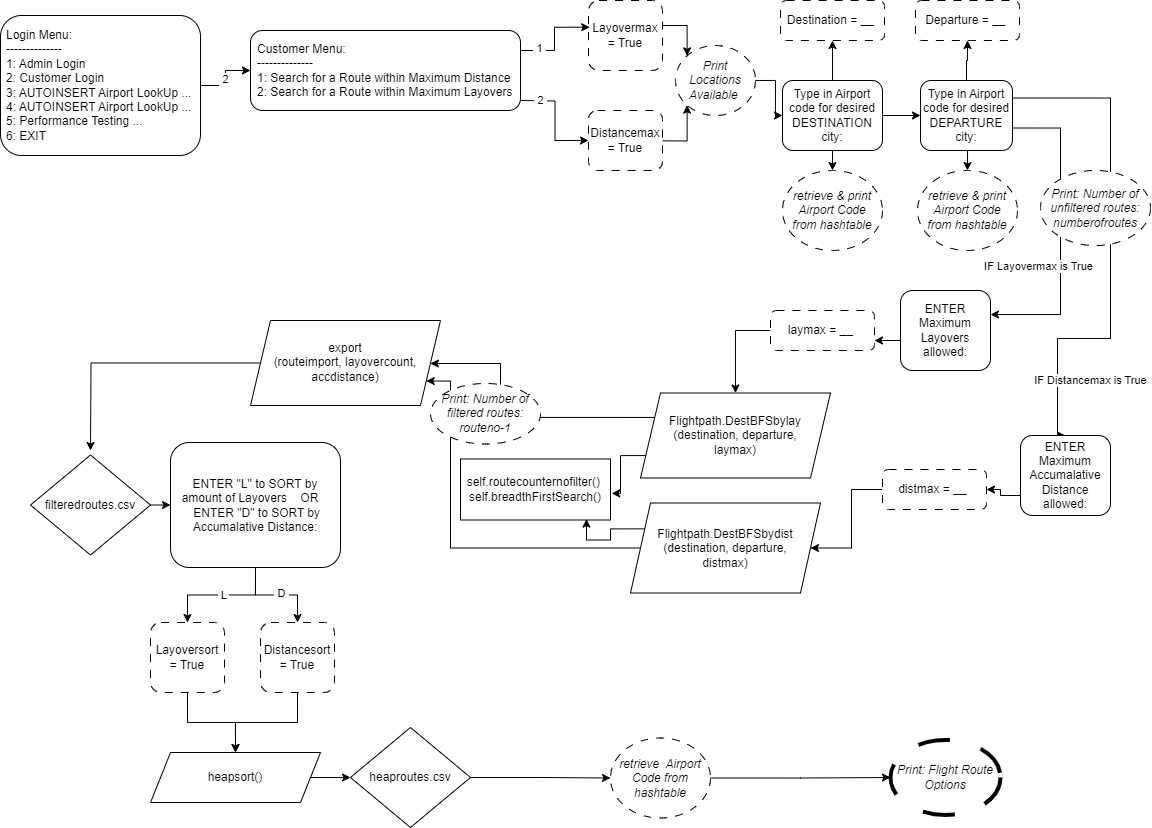


Figure 12: Flow Chart of how route search works

--- See BreadthFirstSearch() Flow chart

Process:

To perform a route search, user selects 2 in Login Menu and then selects 1 or 2 in Customer menu. Code outputs all available locations (mapping graph vertices with hashtable). User then prompted to type in airport code for destination. Code outputs the retrieved value from the hashtable. User then prompted to type in airport code for departure. Code outputs the retrieved value from the hashtable. User is then asked to enter maximum either layovers or distance (depending on what they chose earlier). The code then calls either DestBFSbylay or DestBFSbydist. Within this code it performs a Breadth First Search (see BFS section), outputs the number of unfiltered routes and then outputs the number of filtered routes. The ‘labelsets’ which are flightpaths are exported to filteredroutes.csv. This export contains number of layovers, accumulated distance, tuples of flights within this route for each route option found by breadthfirstsearch.

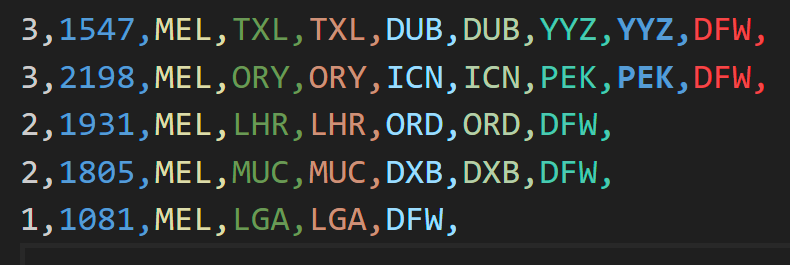


Figure 13: Example of routes exported to filteredroutes.csv

The user is then prompted to enter L or D to sort by accumulated distance or amount of layovers. The function Heapsort is then used to sort the route results into an order which then exports it to heaproutes.csv.

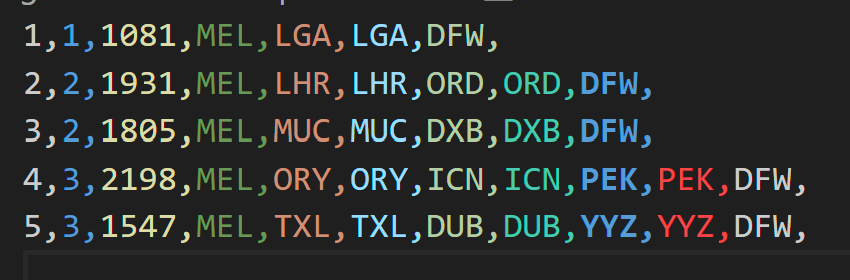


Figure 14: Example of sorted routes (by layover amount) exported to heaproutes.csv

The heaproutes.csv file is then read and printed to the screen with the retrieved hashtable values.

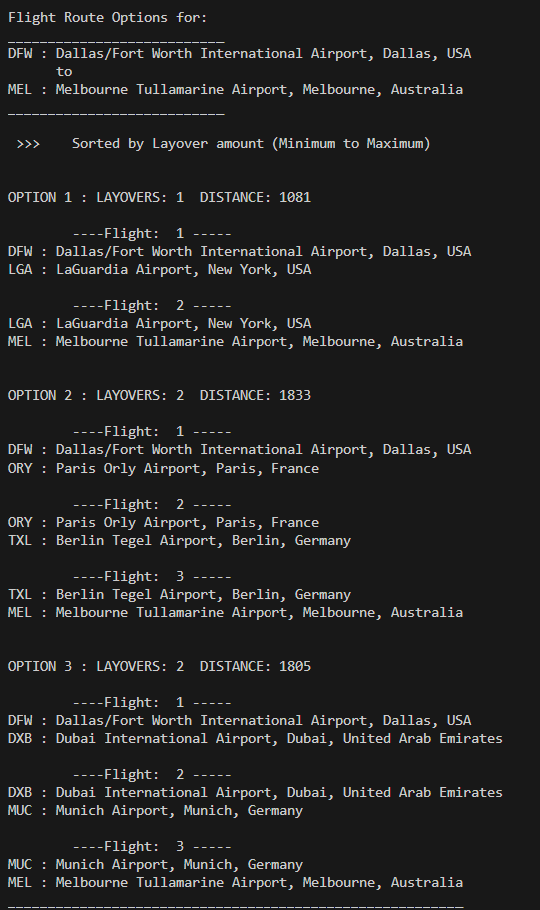


Figure 15: Excerpt of flight route options retrieved from heaproutes.csv and printed with hashtable values

**Breadth First Search**

Breadth First Search(BFS) is an algorithm that allows for search for multiple routes from the departure vertex to the destination vertex in the graph. Inside the graph class, there is five modules that were required for the route search. There were BreadthFirstSearch(), RouteCounterNoFilter(), DestBFSbyLay(), DestBFSbyDest() and Export(). These modules used Circular Linked Lists, Graphs, Queues and Stacks so all associated classes for these are also required.

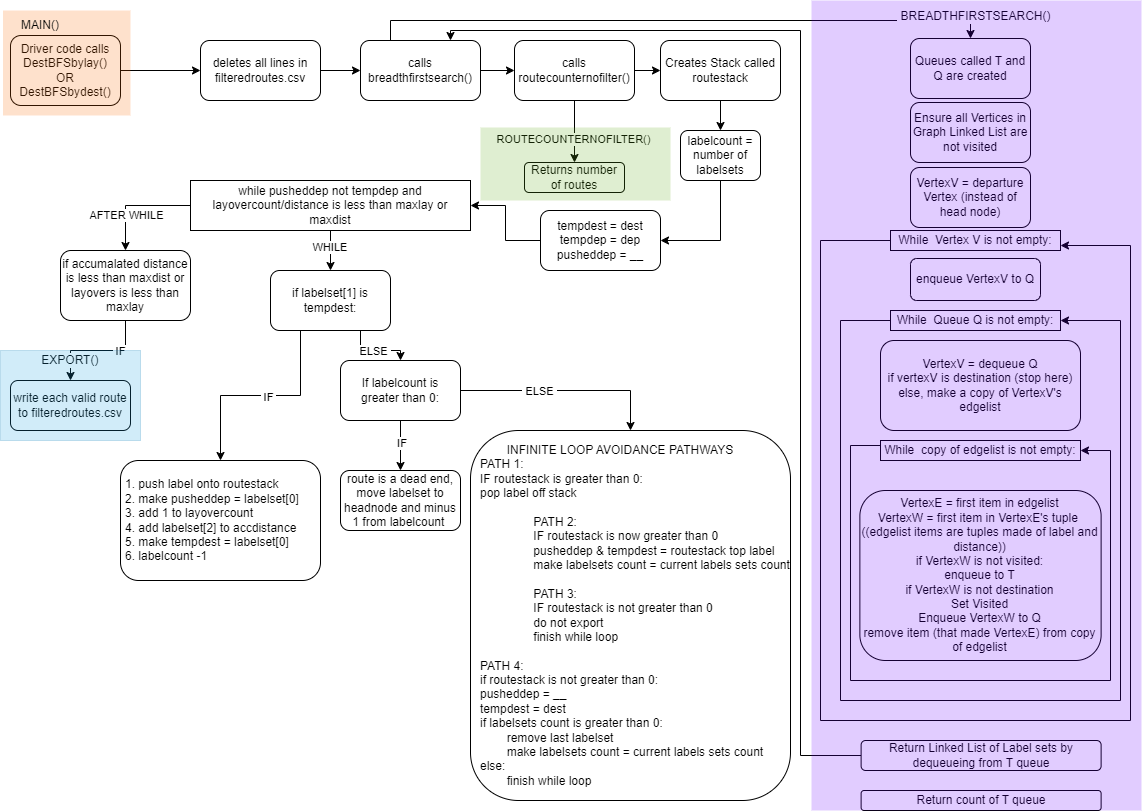


Figure 16: Flow chart of BFS for AnywhereButThere

DestBFSbyLay() and DestBFSbyDest:

These are called by the driver code depending on if the user chooses to limit the route search results by maximum distance or maximum layovers. Both functions call the BreadthFirstSearch Function which returns a Linked List of Graph edges, called a labelset (airport code 1, airport code 2, distance) that possible route options use and the number of edges found (not taking into account the maximum limit specified by the user).

This function prints the amount of routes found (not taking into account the maximum limit specified by the user).



The function then creates a Stack called routestack. For the amount of edges returned by BFS, this function goes through each labelset and builds routes from destination to departure until the maximum layover or accumulated distance is reached. Each labelset that is valid is put into the routestack. Once the routestack has built a route, it is then exported to filteredroutes.csv. Then the process starts again until there are no more labelsets.

This function may have labelsets that do not lead a between the departure and destination (the route edges have been used up by another route found) and therefore are a dead end. This is found when the labelsets have all been worked through and some have been left behind or the stack has not lead to the departure airport. I created a “Infinite Loop Avoidance Strategy” to deal with all possible invalid combinations.

1. If the stack’s top label exists, remove it
2. If the stacks next top label exists – use it to look for the next labelset to continue the route.
3. If no next top label exists on the stack – this is the end of all routes found, do not export this last stack (which is empty anyway)
4. If the stacks top label didn’t exist, start entire process again by removing the latest labelset, looking for the destination label to start and resetting the labelsetcounter. If there are no labelsets, finish process.

For each route processed (stack top labelset[0] is the departure airport), the stack is popped one by one and placed into a queue called routequeue. This routequeue is exported to Export() along with the layovercount and accumulated distance of that route.

Once all routes have been found, this function also prints “Number of filtered routes available”.



BreadthFirstSearch():

This algorithm contains the base code created during the lab practicals with a few changes. Two Circular Queues are still created (called Q and T). It makes VertexV the departure airport vertex instead of the head node. It still enqueues VertexV into Q and then dequeues it from Q. It checks whether this vertex is the destination and if not, makes a copy of the Linked List (edgelist) this vertex contains. It then continues to traverse each of these edges, setting them as visited as it goes. Another change from the base code is that it checks each vertex is not the destination before changing its visited status. The destination vertex doesn’t ever get changed to visited so that multiple routes can be found. A Linked List of each edge used for a route is created by dequeuing from T queue. This is returned to DestBFSbyLay() or DestBFSbyDest (depending which one called it) along with a count of all T queue items.

Export() to filteredroutes.csv:

This function was created specifically for this program. Each valid route is sent from DestBFSbyLay() or DestBFSbyDest, it is then formatted as a csv line and written to filteredroutes.csv. See Figure 13.

**Heaps**

Once filteredroutes.csv has been filled with valid routes, the user has the option to choose whether they want to view the valid route options sorted from min to max layovers or sorted min to max accumulated distance.

To perform either option, a DSAheap is created and called routeheap. Each item from filteredroutes.csv is placed into the routeheap, either mapping its priority to the first element in the line (layover number) or the second element in the line (accumulated distance).

Heapsort:

heapSort() is then called to sort routeheap. This is done by first calling the heapify() method. Then to reuse the same array (in-place), it swaps the root with the last item and changes the count of items to one less, then trickles down the new root and repeats until all items are in the correct place in the array.

Heapify:

Heapify sorts the the unsorted tree by sorting every non-leaf item into the root and then calling trickledown(). Trickledown() then pushes the root element down the tree until it is in the correct position (parent nodes must be greater or equal to its children nodes).

Export() to heaproutes.csv:

Once the array has been sorted by heapsort, the driver code calls export(). This function writes each element along with its value (the route) to heapsorted.csv to be read by the drivercode and mapped to the hashtable.

**Challenges**

The two challenges I experienced in this project came from having to alter my LinkedList class and my Graphs class to allow for weighted edges and where to change the BFS code to avoid infinite looping when a labelset was a dead end.

Weighted Edges:

Once I discovered that weighted edges (tuples) were creating issues when attempting to delete edges, I worked through a solution that allowed to keep as much base code as possible unchanged. DSAListNode() to have a module called getEdgename() which returned the first item in a tuple. DoubleLinked() class to now have a deleteEdgeNode() module that was separate to the DeleteNode() module as edges are tuples and this allowed for successfully finding the node using the first half of the edge tuple.

BFS Infinite Loops:

As mentioned earlier in this report and shown via Figure 16, the complexity of dealing with labelsets that led to deadend routes due to previously found routes using the a labelset that was also used for this route. There were numerous ways of fixing this issue, however I chose to implement the “Infinite Loop Avoidance Strategy” to avoid messing with modules that came as part of my base code from the labs. This was to ensure I didn’t further affect modules that I had tested previously and deemed OK.

**Efficiency**

Data Structures:

A diagram of complexity

Description automatically generated

Figure 17: from bigocheatsheet.com

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Structure** | **Time Complexity** | | | |
|  | **Average/Worst** | | | |
|  | **Access** | **Search** | **Insertion** | **Delete** |
| **Array** | O(1) | O(n) | O(n) | O(n) |
| **Stack** | O(n) | O(n) | O(1) | O(1) |
| **Queue** | O(n) | O(n) | O(1) | O(1) |
| **Doubly-Linked List** | O(n) | O(n) | O(1) | O(1) |
| **Hash Table** |  | O(1) | O(1) | O(1) |

*Created by me, with Lecture Slides*

The data structures used inside this code are more efficient due to using Stacks, Queues, Linked Lists and Hash Tables rather than using Arrays. However, the algorithms could be refined, and functions made more modular to assess efficiency better. This code is unnecessarily complex due to programmer inexperience.

**Sorting Algorithms (ADDITIONAL):**

In addition to this project, I tested three sorting algorithms to determine which one was the most efficient (in time) for both sorting route options by distance (with options ranging between 1 and 10,000km) and sorting route options by layovers (with options ranging between 1 and 10 layovers). The algorithms tested were Merge Sort, Quick Sort and Heap Sort (all written by me).

To get a clear idea of the efficiency of these algorithms, I tested with large numbers of route options (just the distance numbers or layover numbers – not attached to actual route values as it was unnecessary to generate the actual routes for the tests). I tested with 3, 5000, 10000, 15000, 20000 and 25000 routes. It would be very unlikely that route options would ever get this high but it allowed for a better indication of which sorting algorithms were the most time efficient (in seconds).

I made sure I tested the Layover numbers and Distance numbers separately as Layovers being between 1 and 10 would test how well the sorting algorithms dealt with a high number of repetitive numbers due to its small range. This would reflect a real situation.

The algorithm modules are in DSAsorts.py. I used SortingTestHarness.py supplied by COMP1002 Lab 1 to write SortsTest.py which conducts three tests on a given amount of random numbers within a given range and times them with seconds and takes the average. I also wrote a bash shell file to automate the multiple tests, output the results to output.txt and call plotoutput.py to then read this text file and make two line graphs with the results.

**Algorithm Features:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Average Case** | **Worst Case** | **In-Place** | **Stable** | **Recursive** |
| QuickSort | O(N log N) | O(N^2) | Yes | No | Yes |
| MergeSort | O(N log N) | O(N log N) | No | Yes | Yes |
| HeapSort | O(N log N) | O(N log N) | Yes | No | Yes |

**To Test:**

1. Open subfolder “AdditionalMarks”.
2. Open gitbash and type ‘./input.sh’ OR open subfolder with Terminal. Type 'bash input.sh'.
3. Wait for all errors to be completed (this is part of the test).
4. A pyplot will open up (see below) with results. Also a txt file called output.txt is generated with the results of each test.

**Output:**

A graph of different types of graphs

Description automatically generated with medium confidence

Figure 18: Plotted output of Sorting Algorithms

A screenshot of a computer

Description automatically generated

Figure 19: Table of Results (made from output.txt)

**Results:**

This simulation shows that for Distance where the range of numbers is large and therefore not too repetitive, Quick Sort was the most efficient in time and would be O(N log N). This was expected as Quick Sort is known to be on of the fastest sorting algorithms. It is an In-place sort which means it doesn’t need much more RAM. I also used a 3-way pivot strategy to choose a pivot point which would of helped it’s efficiency.

For Layovers, where the range of numbers is small and therefore very repetitive, Quick Sort would fault on Recursion Errors and would not complete the tests over 5000 route options. This was even with a good pivot strategy. Merge Sort was a better option, O(N log N) even though it was not an in-place sort (used more RAM). This is because QuickSort is an Unstable Sort which meant it would continue to sort like items rather than leave them in place. It’s recursive nature lead to stack overflows (Recursion errors) and was O(N^2) which is worst case.

**Future Improvements/Alternatives**

1. Double Hashing Options:

If I had more time I would research optimal double hashing algorithms that would result in less collisions and less clustering when double hashing. I would like to test a few options and plot them on a graph for easy decision making.

1. Simplify Breadth First Search:

If I had more time, I would simplify all functions associated with my Breadth First Search algorithm. This includes working on it to make it more modular and reduce complexity. This may include introducing code to skip looking for BFS routes once it hits the user specified maximum or layovers/accumulated distance. This could be an alternative to my current set up which is returning all possible combinations and then filtering out the routes that comply.

1. Advanced Sorting:

I would also remove my heap sorting algorithm and use a Quick Sort algorithm in it’s place if the User chose to sort by Distance. I would implement a Merge Sort algorithm instead of a Heap Sort if the User chose to sort by Layovers.

**References**

HashClass: https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html#:~:text=Generally%2C%20the%20default%20load%20factor,%2C%20including%20get%20and%20put

LOAD FACTOR THRESHOLD CHART - https://realpython.com/python-hash-table

AirportLookup: https://www.world-airport-codes.com/world-top-30-airports.html

<https://en.wikipedia.org/wiki/List_of_international_airports_by_country>

Big O graph: bigocheatsheet.com

Representation of my Airport Network Graph: <https://csacademy.com/app/graph_editor/>

Additional Marks Section References:

SortsTestHarness by Valerie Maxwell - used some of this code to time my sorting algorithms for Additional Marks section

https://medium.com/@tuvo1106/merge-sort-in-python-5d9617fb9ee1

https://visualgo.net/en/sorting

https://www.geeksforgeeks.org/python-program-for-quicksort/