Comp 1002 – Data Structures and Algorithms

Final Assignment Report

By: Robert Lewis

Student ID: 21494561

Tutorial Class: Monday 2pm

Semester 1 2024

Summary of Work:

The code is designed to import data from a file containing information about a list of airports, and a file containing a list of flights between airports. It will then allow a user to search for a sequence of flights to get them from one airport to another. The results of this search will be sorted, either by distance or by layover count, then output to the terminal.

The files containing the list of airports and the list of flights was downloaded from <https://openflights.org/data.php>, they were then parsed to convert from Unicode to ascii.

Implementation Strategy

Initialisation:

First all the airports are loaded in, each line is parsed, turned into an airport object, then added to the hash table of all airports. Since the IATA codes are all unique short strings, they were used as the key.

Then the flights are read in and added to a graph. To limit the size of the graph, only routes operated by Qantas (id = “qf”), British Airways (id = “ba) and Spirit (id = “nk”) are considered. If a route is operated by one of those three airlines, the chode checks if if the source and destination airport are already in the graph, and adds them if they aren’t. Then an edge between the airports is added.

Since many airports do not have a route operated by one of the three airlines, they are unreachable, so any airport not in the graph is purged from the hash table of airports.

The code then prints how many airports and flights were imported and is now ready for user input.

Before each user input it prints a help menu, reminding the user of the syntax of the commands. The program accepts two types of inputs, requests for info about an airport with a specific IATA code, or it can find and display routes between two airports.

Airport Info

To get info about a specific airport, input “aprt” then the IATA code of the airport you’re interested in. The program will then output the name of the airport corresponging to that code. To do this, it looks up the IATA code in the hash table of airports, since the IATA code is the key.

Route Finding:

To find the routes between the airports is more complicated. The user inputs “rout” then the code for the airports they want to travel between, and whether they want the results sorted by layovers or distance. The code checks that both airports are part of the graph, and throws an error if they aren’t.

If both airports are part of the network, the code conducts a breadth first search of the airport graph. This is achieved by tracking a linked list ( functioning as a queue ) of routes to search. Each route is represented as a linked list, the first element of which is the length of the route so far, and then the remainder of the list is the ordered list of airports the route passes through. The algorithm repeatedly pulls a route from the front of the queue, and checks if the last airport it visited is the destination. If it is, this route linked list is added to a linked list of valid routes.

If the route is not complete, and the max layover count hasn’t been reached, the route is appended by all the airports the last airport is connected to that haven’t already been visited by this route. The distance of the route is increased correspondingly. These routes are all pushed to the back of the queue to search.

Once the search is completed, there is a linked list that contains every successful route found. This is then fed into one of 4 sorting algorithms, detailed below.

Once the linked list of valid routes is sorted, the code checks if it is longer than 20 routes, if it is, it only displays the top twenty routes, otherwise it displays every route.

Sort by Layover:

This is based on a heap sorting algorithm. It creates an array of heaps, with a length equal to the maximum length of a route, which is known since the max depth to search is specified by the user. Each heap is initialised to have a length equal to the number of valid routes, to avoid any chance of the heaps overflowing.

Each route is added to the heap with it’s index corresponding to the route length, the heap will then use the routes total distance as the priority.

The heaps will then be systematically emptied starting at highest index heap, emptying that heap completely, then moving on to the next highest index heap and repeating until all heaps are empty. As each route is pulled of it’s heap it’s pushed to the front of a linked list. Once this process is complete, this linked list will hold all the routes, sorted by minimum layovers, and then minimum distance.

Heap Sort By Distance:

This pushes all routes onto a heap, then pulls them all off the heap, pushing each to the front of a linked list. The distance the route takes is used as the priority. This results in a linked list that holds all routes sorted by distance.

Merge Sort By Distance:

This converts the linked list of unsorted routes to an array. The array is recursively broken in half, each half is merge sorted by distance, then the two halves are merged back together, repeatedly removing the lowest element ( which is the first un-merged element from one of the half arrays ) and returning that element to the full array.

Once the full array is sorted it is converted back to a linked list, which is also sorted.

Quick Sort By Distance:

This converts the linked list of unsorted routes to an array. The last element of the array is picked as the pivot. Any elements smaller than the pivot are moved ahead of the pivot, and any elements larger than the pivot are moved after the pivot. This means the pivot is now correctly positioned in the array. The sections of the array before and after the pivot are then quick sorted by distance.

Once the full array is sorted it is converted back to a linked list, which is also sorted.

Challenges

While creating this I faced a few challenges. I encountered many minor bugs such as off by one errors. I overcame these by testing regularly, to ensure each subsystem was working properly before further work was done on top of it.

Another challenge was maintaining a large codebase. The final project contains a few thousand lines of code, way too much to keep in my head. I also completed the project over several weeks. To ensure I could keep working on the code I made sure I documented each function, and maintained good modularity principles. I also regularly made minor concessions in performance for enhances in readability.

Efficiency Analysis

Throughout this section some simple representation will be helpful:

a = number of airports

d = depth of search

n = number of flights

v = number of valid routes

Loading Airports

Reading an airport from the file is O(1).

To add an airport to the hash table is nominally O(1) since it should go in the first slot it hashes too. Worst case is O(a) as it may have to probe past every other airport.

Sometimes the hash table will need to resize.

Resizing requires:

* Finding a new prime. Checking if a number is prime is O(a^0.5). Since primes are roughly evenly distributed, finding the next prime after a number is also O(a^0.5)
* Copying all the previous entries into the new hash table which is O(a).

Therefore, resizing is also O(a)

Overall loading each airport is between O(1) and O(a), so loading all airports is between O(a) and O(a^2), though since the bad cases for adding airport to the hash table are relatively uncommon, it is likely much closer to O(a) than O(a^2)

This could be significantly optimised though, since the code later goes through and heavily purges the hash table, so many airports are wasting time being added, more efficient code could avoid this.

Loading Flights

To load a flight requires:

* Reading it in and interpreting it, which is O(1)
* Checking to determine if it should be added to the graph which is O(1)
* Checking if each airport is in the graph
  + Requires iterating through the linked list of all nodes, which is O(a)
* Calculating the length of the flight which is O(1)
* Adding the edge to the graph which requires:
  + Finding the two airports in the graph which is O(a)
    - The graph stores nodes in a linked list so to find each airport requires iterating over the linked list
  + Adding the edge
    - Requires checking the edge doesn’t already exist
      * Requires iterating over the list of nodes this node is connected to already, since that’s a linked list with average length = n / a, this is O(n / a).
      * Since n can’t be larger than a^2 which is when every node is connected to every other node, O (n / a) is less than O(a)

Therefore loading each flight is O(a), so loading in all the flights is O(a^2).

The algorithm could be much more efficient if the nodes in the graph, and the nodes each node is connected to were stored in a hash table, since this would make accessing each node O(1) instead of O(a), and increase the efficiency of loading all the flights to normally O(a).

Airport Lookup

To lookup an airport requires accessing it in the hash table, which is nominally O(1) and at worse O(a). Then it gets the name from the airport object which is O(1).

Route Search

The code:

* Checks both airports exist which is O(a)
* Creates an initial route and adds it to the que to search which is O(1)
* For each route in the que:
  + It checks if it’s done which is O(1)
  + If not done it adds all potential routes, to the end and adds these new routes to the search que which is O(n / a)
* Will end up searching (n/a) ^ d routes, so will take O((n/a) ^ d)

This could potentially be optimised if we only wanted to search for the best route, as if a node is connected to the destination we should only add that route, not any others to be searched. But since we are looking for all routes, we do need to check these alternative but worse routes to see if they work.

Sorting Algorithm Comparison

The three sort by distance algorithms (heap, merge and quick) are all O(v log v) in time complexity.

Quick sort could be optimised in memory usage, by freeing the copied arrays before sorting the sub arrays, which would reduce it’s memory usage from O( v^2 ) to O(v).

A graph with different colored dots

Description automatically generated

After running the three algorithms, and timing their sort times on randomly generated routes, these were the results. The full data is available in results.csv and results.xlsx.

The data is poor quality, measuring an algorithms length by timing it on a computer that is running other things will result in inconsistent readings. What can be seen from this data though is that the time grows non-linearly, but not as fast as an exponential or quadratic. All three algorithms seem to be growing at a similar rate, which is also expected since they are all O( n log n).