AIRNAV

Azlan Naeem, Arjun Menon, Hadi Naqvi, Rohan Regi

Introduction:

Air travel is an essential part of modern life, and millions of people travel by air every day. The efficient management of air traffic is critical to ensuring the safety and convenience of air travel, and one important aspect of this is the calculation of the shortest flight paths between airports. Calculating flight paths is a complex task that involves a range of factors, including the distance between airports, weather conditions, fuel efficiency, and air traffic control regulations. Despite the importance of this problem, it remains a challenging task for airlines and air traffic controllers. The shortest distance between airports is the direct distance between the two airports. This distance is calculated based on the coordinates of the two airports on the Earth's surface using mathematical formulas such as the Haversine formula [5]. However, this distance may not necessarily be the shortest flight path between the two airports, as factors such as weather, air traffic congestion, regulatory restrictions, airport capabilities (whether one airport is connected to another without layovers) can affect flight paths. In our case, we are considering the factor of layovers where a direct flight from one airport to another is not always possible. As a result, we have implemented distance-weighting using the Haversine Formula combined with our shortest path algorithm which takes into account these calculated weights/distances by repeatedly going through the graph and keeping track of distances to create a final shortest path. Therefore, optimizing flight paths involves considering this factor and finding the most efficient route that minimizes travel time and fuel consumption while also ensuring safety and compliance with regulations. We have chosen this project question/goal because we are interested in the similarities and connectivity between the travel and technological industries, and we believe that optimizing flight paths can have a significant impact on the efficiency and sustainability of air travel. By developing new approaches to calculating flight paths, we can reduce travel time, minimize fuel consumption, and improve the overall efficiency of air travel, while also enhancing the safety and convenience of air travel for passengers. **The goal of our project is to utilize graphs, algorithms and mathematical computations to calculate the shortest path between two airport locations.**

Dataset Description:

For this project we used the datasets from these two kaggle datasets:

- https://www.kaggle.com/datasets/open-flights/airports-train-stations-and-ferry-terminals
- https://www.kaggle.com/datasets/open-flights/flight-route-database

The names for the two datasets we used are airports.csv and flight_routes.csv. Our airports dataset is a dataset which has information on real airports from around the world. The columns that came with this dataset are:

- Airport ID - Unique OpenFlights identifier for this airport.

- Name - Name of airport. May or may not contain the City name.
- City - Main city served by airport. May be spelled differently from Name.
- Country - Country or territory where the airport is located. See countries.dat to cross-reference to ISO 3166-1 codes.
- IATA 3-letter IATA code. Null if not assigned/unknown.
- ICAO 4-letter ICAO code.
    - Null if not assigned.
- Latitude - Decimal degrees, usually to six significant digits. Negative is South, positive is North.
- Longitude - Decimal degrees, usually to six significant digits. Negative is West, positive is East.
- Altitude - In feet.
- Timezone Hours offset from UTC. Fractional hours are expressed as decimals, eg. India is 5.5.
- DST Daylight savings time. One of E (Europe), A (US/Canada), S (South America), O (Australia), Z (New Zealand), N (None) or U (Unknown). See also: Help: Time
- Tz - time zone Timezone in "tz" (Olson) format, eg. "America/Los_Angeles".
- Type - Type of the airport. Value "airport" for air terminals, "station" for train stations, "port" for ferry terminals and "unknown" if not known.
- Source Source of this data. "OurAirports" for data sourced from OurAirports, "Legacy" for old data not matched to OurAirports (mostly DAFIF), "User" for unverified user contributions. In airports.csv, only source=OurAirports is included

However, we did not need all of this information for airport data. We only need the airport name, city, country, and the 3 character code such as ‘LAX’. In our code, we include these 4 columns mentioned when using the data.

Our flight_routes.csv file came with the columns:

- Airline 2-letter (IATA) or 3-letter (ICAO) code of the airline.
- Airline ID Unique OpenFlights identifier for airline (see Airline).
- Source airport 3-letter (IATA) or 4-letter (ICAO) code of the source airport.
- Source airport ID Unique OpenFlights identifier for source airport (see Airport)
- Destination airport 3-letter (IATA) or 4-letter (ICAO) code of the destination airport.
- Destination airport ID Unique OpenFlights identifier for destination airport (see Airport)
- Codeshare "Y" if this flight is a codeshare (that is, not operated by Airline, but another carrier), empty otherwise.
- Stops Number of stops on this flight ("0" for direct)
- Equipment 3-letter codes for plane type(s) generally used on this flight, separated by spaces

In the same way, we don’t need every column for our algorithms/computations. So, in our code we only include the Source airport three character code and the Destination airport three character code when using the data.

Computational Plan:

Our flight routes were represented by a weighted directed graph in which each vertex is an object of an airport class called “Airport.” This data class had various instance attributes that included information about the airports such as its 3 character code, the airport name, and the city and country it is located in. Each edge for a corresponding vertex will represent the path or route between two airports with a weight associated with each edge. We chose our weights to be distance-based for this project although there were other options such as cost-based weights. Notice that the 3 character code replaces the “item” instance attribute in the _Vertex class from lecture; the other instance attributes were necessary for our GUI and displaying various information about the airports. The main Graph class was represented by the “FlightNetwork” class which has one private instance attribute storing all the vertices (airports) in the network/graph. The initializer of this class adds each airport to its instance attribute, and then also mutates each airport so that its “routes” instance attribute which represents edges/connections is modified. Note: Not all connections/edges are two-way, since this is a weight directed graph, and thus each airport has its own separate edges containing the airport object it’s connected to as well as the weight (distance between them).

For this project we we used these two datasets:

- https://www.kaggle.com/datasets/open-flights/airports-train-stations-and-ferry-terminals
- https://www.kaggle.com/datasets/open-flights/flight-route-database

We chose to use these two datasets because they both contain necessary information that the other does not include. For example, the first dataset contains information such as the airport name, city, and country, whereas the second dataset includes information like the flight routes from one airport to another—this is needed in order to find the distance between flights.

The data stored in the first dataset/csv file was converted to a dataframe using pandas so it was easier to work with. We then used this dataframe to get the location, that is the latitude and longitude, of each airport in the first dataset using <get_location()>. We also used this dataframe to obtain the general information for each airport, that means we obtained the 3 character code, airport name, city , and country using <get_airports()> which loops through the dataframe. Afterwards, the data stored in the second dataset/csv file was also converted to a dataframe using pandas as it makes our data much easier to work with. This dataframe was used to get destination airports for each source airport using <get_destination()> which takes an airport as a parameter. Contrary to get_airports(), get_destinations() does not just access certain rows of the dataset. The get_destinations() function actually uses both dataframes to find source airports with the same code as the input airport and then looping through those rows to then create a list of tuples with the destination’s code and its respective latitude and longitude. These functions

were used in the network.py module in order to initialize the flight network, as well as the main.py module in order to retrieve the list of airports for the user to select in the GUI.

The dataframe for flight routes is accessed in order to find new flight paths/connections which are then added as edges to the airport vertices, by adding each vertex as a neighbour to the other.

Next, we assigned distance-based weights to the edges or routes of our graph by using the longitude and latitude values from our dataset and then computing distance using the haversine formula in the network.calculate_weight function. This is the formula used to calculate distance in kilometers between two points on a sphere or the Earth in this case. Note: This function takes in the latitude and longitude as floats since precise values of the latitude and longitude are necessary in order to compute the distances.

Then, using our newly built graph we applied Dijkstra's algorithm to find the shortest path between two vertices (airports) based on the user's selected origin and destination airports. This algorithm keeps track of visited nodes and tentative distances to each airport from the origin airport as it traverses through the graph by repeatedly selecting the neighbouring airport with the smallest tentative distance using a priority queue/min heap (This heap is a binary tree where each node has a priority value, and the priority of any child node is less than or equal to the priority of its parent node). The heap/queue starts with the origin airport, and then its neighbours are appended to it with their accumulated weights/distance as the priority measure. This repeats for all neighbours/airports, until it reaches the destination airport. Then it returns the path. This algorithm is one of the most efficient methods for finding the shortest path between two vertices in a graph and utilizes similar concepts learned in lecture such as a set of visited/unvisited vertices.

Finally, we used Tkinter to obtain and display our data/computed results. Our GUI will consist of two text entry boxes with drop-down search results used to retrieve the airport codes of the origin and destination airport, and a text label box that will display the shortest possible flight path along with its corresponding total distance. We chose Tkinter for our Python Library because it has high customizability and interactiveness allowing us to create a GUI specific to our project.

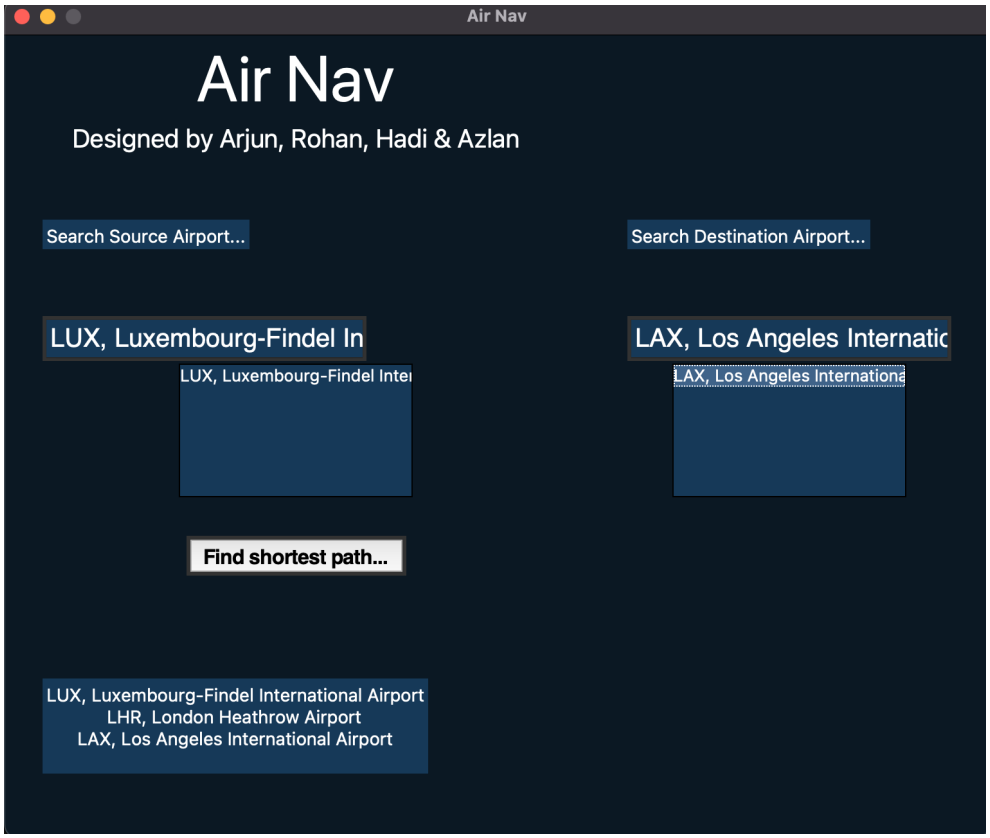
Pandas and Tkinter were two crucial libraries that were utilized in this project. Since we had very large datasets that included unnecessary information, converting our csv files to pandas dataframes allowed us to clean and transform our datasets to be functional. For example, we used pandas <.replace()> and <.dropna()> built-in functions to remove empty columns and rows that we couldn't use. Additionally, pandas dataframe are more efficient compared to using nested lists which would have been the other option for reading in a csv file. When working with much larger datasets, running time is crucial which is why we chose to use pandas. Furthermore, Tkinter allowed us to display our program in an interactive manner. We used text boxes, search boxes, and list boxes to allow the user to input two airports to travel between. In this way, the user can easily find the quickest path between the airports they travel from and to.

Instructions to Run Program:

To acquire the datasets required for our program to run, simply view the ZIP file submitted on Markus which contains a requirements.txt file, our various csv files and our main python code scripts.

To run this program the user should simply run our main.py file. The main block in this file includes most of our GUI widgets like the two entry boxes which have been previously mentioned in this report. **Users should ensure they select two source and destination airport codes from the drop-down menu before clicking our compute button to calculate the shortest distance.**

Here is a sample of our GUI which should be displayed after running main.py:



The user is able to input their source airport (on the left) as well as the destination airport (on the right). Once they have inputted at least one airport, the user can then find the shortest path between the two airports by clicking the “Find shortest path…” button. This path is then displayed near the bottom of the screen. To understand the shortest path displayed, the source airport will be the first airport listed (at the top), connecting airports will be listed in the middle,

and the destination airport will be listed last (at the bottom). In this case, LUX is our source airport, LHR is a connecting airport, and LAX is the destination airport.

Changes:

Since our proposal, we made numerous changes to our program. Firstly, we decided to use the pandas library in order to parse through our datasets and transform them into dataframes which are easier to work with. We decided to do this over using a csv reader and reading through our datasets row by row because our datasets were very large and thus the runtime would be too slow. Additionally, we decided not to combine the two different datasets we were working with, and to also not use them in their original form. Instead, we pruned our datasets to remove the most redundant, unnecessary, and least likely to be used data in order for the runtime of our program to be faster. We also added a header row to the first dataset as it originally didn't have one and was useful to understand what each column corresponds to. Additionally, we had to filter the data to obtain useful columns only as mentioned above. After these transformations, the data was much easier to work with. Moreover, another significant change that we made was to implement a separate search bar in our GUI to allow the user to select airports more easily when trying to find the shortest route. Lastly, one small, yet important change we implemented was an edge-case in our network.find_shortest_route() method to handle the case where there is a direct connection between two airports, since that would always be the shortest path. This was necessary in order to prevent our algorithm from unnecessarily searching through the graph.

Discussion:

We have been able to successfully utilize graphs, algorithms, and mathematical computations to calculate the shortest path between any two airport locations. We are able to display the shortest route between two airports using a graphical user interface, which makes it easier for users to understand the calculated route and make informed decisions about their travel plans.

In addition, our project is able to handle large datasets efficiently, allowing for the analysis of complex travel routes and scenarios. This leads to more optimized travel plans and significant cost savings for both airlines and passengers.

Our shortest path algorithm works very well to choose the shortest path even in tricky situations. For example, when testing fin_shortest_route() we would sometimes get a path with multiple stops even though there would be a path with only one or two stops. In these cases, we found that by using our calculate_weight() method we could see that the overall distance was less when taking more stops. This proved that our algorithm is truly influenced by our weights and not just what you would think is the "shortest" path.

However, there were many issues and limitations when trying to implement this project. For example, when trying to convert the two csv files into data frames there were many issues with certain rows having \\N values in the airport data. So, we filtered this out. However, this raised even more problems as it significantly shortened the dataset and took out airports that are used in the flight routes dataset. After several attempts, we managed to successfully filter out irrelevant columns in our dataset so that the ones that remained still had an adequate number of connecting flights associated with them in our routes file. In addition, had our datasets after filtering been much larger, it may have taken far too long to compute the shortest path for every flight route in the dataset due to the computational complexity of the many functions used. Therefore, for our program, a larger dataset results in a large running time proportional to the size of the dataset. Additionally, another limitation of our program is that only distance is used to calculate the weights of the routes while other factors like fuel consumption, travel cost and time could've been taken into account.

Looking towards the future of AirNav, there are a few steps we could take to further this application. First, we would like to add an abundance of new features. For example, we could expand from just airports to various modes of transportation. That is, users could choose between different ways to travel such as plane, train, ferry, etc. Or, we could allow users to specify preferences such as budget or time. In relation to this, incorporating real time data would be an excellent addition to our program. Instead of calculating weights just based on distance, we could include cost, flight schedules, delays, and more. Another important step we could take to improve AirNav is to improve our user interface. This could be done by using a programming language such as JavaScript that allows for immersive and interactive features such as a map or world view. Finally, integrating AirNav with other applications would be the ultimate goal for this project moving forward. Integrating this program with other travel-related applications such as hotel booking or travel services, to create a more seamless and comprehensive travel planning experience for users would be extraordinary.

Overall, this project has become a valuable tool for optimizing air travel routes and improving the travel experience for everyone.

References:

[1] Amos, David. "Python GUI Programming with Tkinter – Real Python." Realpython.com, 30 Mar. 2022, realpython.com/python-gui-tkinter/.

[2] avitexavitex 2. "Haversine Formula in Python (Bearing and Distance between Two GPS Points)." *Stack Overflow*, 1 Dec. 1957, https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points.

[3] Connors, Liam. “Creating a Simple Map with Folium and Python.” *Medium*, Towards Data Science, 2 Aug. 2022, https://towardsdatascience.com/creating-a-simple-map-with-folium-and-python-4c083abfff94.

[4]“Dijkstra's Algorithm.” *Wikipedia*, Wikimedia Foundation, 3 Mar. 2023, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[5] “Haversine Formula.” *Wikipedia*, Wikimedia Foundation, 1 Mar. 2023, https://en.wikipedia.org/wiki/Haversine_formula.

[6] OpenFlights. “Airports, Train Stations, and Ferry Terminals.” *Kaggle*, 28 Aug. 2017, https://www.kaggle.com/datasets/open-flights/airports-train-stations-and-ferry-terminals.

[7] OpenFlights. “Flight Route Database.” *Kaggle*, 29 Aug. 2017, https://www.kaggle.com/datasets/open-flights/flight-route-database.