The network flow problem requires consideration of trips between airports that involve (at most) one layover. A layover is defined as a trip that includes an origin airport, and one intermediate stop, before reaching a destination airport. For each origin and destination airport that has a route, we have examined the following:

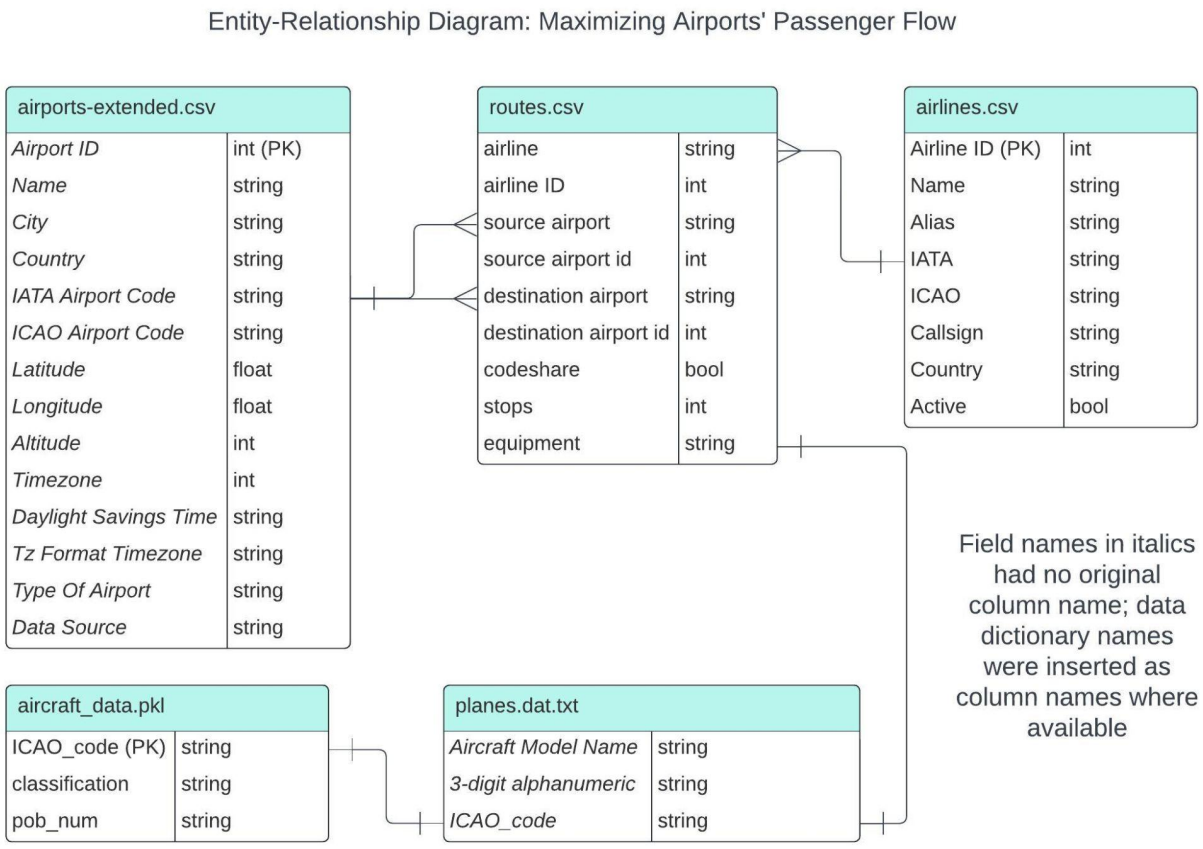
- Given your passenger capacity available on each flight, what is the maximal number of people that can be moved from an origin airport to a destination?
- Which carrier can transport the greatest number of individuals on this route?

1. Data Pre-Processing & Transformation

Initial data provided in the Kaggle dataset included four CSV files. This data included aircraft codes, routes, and airlines, but no specific columns regarding passenger capacity per plane. This situation required research on sources for passenger capacity which led us to doc8643.com. This website has a simple HTML layout that includes data on each plane per ICAO code and classification, along with passenger capacity labeled "Persons On Board". It was selected not only for this simplicity which allows easier scraping, but also for its comprehensive dataset. The site includes over 2,500 aircraft derived from ICAO Document 8643, an aircraft type designator document which includes those aircraft requiring air traffic service. For these reasons, the document appeared to be an excellent fit for the approximately 250 aircrafts referenced in our passenger plane data.

The scraping tool was written in Python using VSCode, using lxml and the requests library. This strategy is preferable to other scraping options like BeautifulSoup and Selenium due to its speedier runtimes. The tool was built by requesting a web address from doc8643.com which was parsed to provide a table of contents style list of the site's offerings per plane. Each plane in the list located at the original address has its own website address, which is where the passenger capacity data is located. Once the planes' address list was compiled and scraped, a request was made per plane to retrieve the "Persons On Board" data, which was then appended to the matching ICAO code and classification.

Once the scraped data was added, we can derive the following entity-relationship model:



With this information, the various csv files have been explored for data quality, reduced to pertinent columns, merged with joins on key columns, and renamed into a single dataset. Full names of airports and carriers were added for readability and context. While preparing this processing, the need to contemplate several design decisions became apparent, as follows.

2. Design Choices

As the data was examined, several questions regarding the terms of the problem became apparent. First, while passenger capacities are often integers, occasionally we found values like “246+2” or “150+3”, which we determined refer to the paying passengers (the larger number) alongside the pilots. Because the problem is a maximization of passengers, we decided to retain only the passenger values here, discarding the pilot-related numbers.

Next, we considered scenarios in which an airline had multiple aircraft models listed for a specific route. This situation provides the potential to select different planes per single flight from the same origin to the same destination. Because different models will have varying passenger capacities, this would

affect our final numbers. Our group consulted the original requirements of the project to provide direction on this issue. We concluded that, because we are seeking the maximum number of passengers a single carrier can convey, we should sum the total passenger capacities in cases where an airline held more than one model of aircraft at its location.

Note that this dataset does not tally an inventory of aircraft per airline, so one aircraft model might be available in quantity for an airline, but we do not have this information. By summing the capacities available, we would have sought to reach the most accurate picture of maximum capacity using the given data.

The final dataset we were able to produce does not provide this sum capacity, however. Instead, it utilizes the maximum capacity available from the largest aircraft of those listed per route per carrier.

We also discussed the issue of null values which did appear in three columns:

492 missing values for airline_ID
220 missing for source_airport_ID
221 missing destination_airport_ID:

In total, if these missing values are found in unique rows, we would lose a maximum of 1.3% of the routes. Depending on the imperative of the project requirements, in a real-world scenario we might or might not be concerned with this percentage of loss. Because the data is randomly missing with no way to resolve, and because its percentage of the total dataset is relatively small (933/67766), we have decided to omit these rows.

While some team members explored preprocessing issues of passengers, aircraft capacities, and missing data, others considered the definition of our data structure. Knowing that we must provide an origin and destination (i.e. source and sink) in a scenario with zero or more passengers (edge weights), the most suitable framework is a flow network, defined as:

A directed graph with two distinguished nodes, source and sink, and no anti-parallel edges, for which each edge has a non-negative weight called the edge capacity. Each weight is described as a cost c where c(u,v)); in this case, passenger capacity.

Before the data structure could be fully represented, we also needed to refine the definition of source and sink, given project constraints. For our purposes, the source and sink comprise the route, and the layover is a possible node incorporated within the route. A route is defined as:

- a traversal from node u to node v, in which
- u does not equal v, and
- some positive edge exists from nodes u to v,
- with zero or one intermediate nodes between u and v

A layover is defined as the single intermediate node that may exist between u and v, as per the requirements of the project.

3: Description of Algorithm

For flow network algorithms, flow is constrained by capacity. For all u, v $\in$ V,

$$0 \leq f(u, v) \leq c(u, v)$$

For this flow network, we operate under the same principle. At maximum, we can supply the number of passenger seats based on aircraft available, within the available network of routes. The code runs in $O(V^2)$ time, between the for-loop required to shell the first set of nodes adjacent to the source, and the potential for a run through the adjacency list in the case of a layover. A more detailed description of runtime would be $V^2 + V^2$, because the adjacency_list method has a relatively heavy computational time requirement, as it utilizes a nested for-loop, though this resolves to $O(V^2)$.

To explain runtime in greater detail, we can describe the two-part operation that occurs. First, the graph is created as a two-dimensional matrix, with each [u,v] matrix location containing an edge weight. For any route that has more than one flight between the two vertices, those multiple edge weights are summed and located in the corresponding matrix field, so that only a single matrix is necessary. Along with the graph, a dictionary is also created that includes both a list of the vertices as keys, and a counter as values. The counter values correspond to each source node's matrix location.

When an airport ID is entered, we begin the process of finding the maximum capacity. First, a list called vert_capacity is created in which every value is zero, except the source node which is designated infinity. Then, an adjacency list is created which holds edge weight values only when they exist, and only for the routes which exist. Lastly, the algorithm locates the capacity value for each route by checking the matrix against the adjacency list for that particular source and sink combination.

This algorithm operates similarly to Ford-Fulkerson, in that both begin with breadth-first search - an initial traversal through all vertices adjacent, in which the algorithm explores all nodes directly connected to a source by an edge. The code varies from Ford-Fulkerson, however, because the residual graph required in Ford-Fulkerson is not a requirement of this network flow. This is due to the single or dual pass needed to provide a correct solution, when the adjacency list is compared in a linear fashion. This contrasts with the iterative requirement in the Ford-Fulkerson scenario where more than two nodes may be passed through, and the network flow requires a greater detail of edge weight information. For this reason, our algorithm does not require a residual network graph, as its shelling is limited to one pass, and once that series of nodes is identified, only one node may be needed afterward, in the case of a layover.

4: Technical Description

Understanding the code requires a walk-through of its functionality, and a technical description of how the algorithm operates. As an overview, the file class_graph.py contains the code which creates a graph, imports the flights data, and provides a network in which maximum flow is determined. At a more detailed level, we can familiarize ourselves with the code's mechanism object by object:

DATA SET:

df_routes, the csv file with the flights data

DATA STRUCTURES OVERVIEW:

This code is comprised of the class flight_graph, and seven methods within it. Local variables to each method (and their use) are noted within the method description:

CLASS DATA STRUCTURES:

flight_graph: the class that acts as an umbrella over all the functions inside it

self.graph, an array which contains the matrix of source (u_vert), sink (v_vert), and edge weights (weight)

self.vert_dict, a dictionary which maintains keys of each airport code in the network, and values that correspond to matrix locations

self.vertcount, an integer that serves as a counter for the number of vertices in the graph

METHODS, in order of execution:

1. importGraphData: a method that takes origin airport (source as uCol), destination airport (sink as vCol), and capacities (edge weights as weightCol) and creates a matrix that represents the graph. The function iterrows() iterates over each row, using that row as a series of individual inputs. In this way, the entire graph is created.

2. addEdge: adds an edge weight to vert_d, called only from importGraphData. It creates all the edges for the graph. Its variables and their corresponding status in the network are below:

u_vert, an integer, source airport code
v_vert, an integer, destination airport code
weight, an integer, passenger capacity

3. addVert: adds a vertice as a key to vert_dict, called only from addEdge. Its single argument, vert, accepts both u_vert and v_vert values from the addEdge function. Each addition receives a counter increment integer as its value. If the vertice is the first one to populate vert_dict, it builds an empty graph. If the vertice is not the first, it appends to the existing graph.

4. displayGraph: a one-line method that displays the matrix that was just created

5. find_max: the bulk of the work gets done in this method. At this point, the network graph has been built, and maximizing within the network begins. The method find_max accepts a source and destination airport code, and has a boolean value for layover as its last input argument.

Within find_max, input_startVert is the origin airport code, which gets converted (by referencing the vertice dictionary vert_dict) into startVert, the vertice label. The same process repeats for input_startVert, resulting in the value endVert (the origin airport node label).

Then initialize_max() and adjacency_list() are called (see below). The returned list vert_capacity provides a list of all zeros, its size equal to V, with the source vertex assigned the value infinity. The returned adj_list provides a list of all possible route segments, recorded as a list of edge weight locations in the matrix.

After this, the variable first_shell derives exactly which routes are possible from the source node, and first_shell assigns the edge weight of passenger capacities to all adjacent nodes. Based on whether there's a layover or not, an if-else conditional statement then executes. With a layover, the method repeats this assignment process once more, from the layover-designated source node to the destination node. Without a layover, this step is skipped. In both cases, the final accumulation of passenger counts is assigned to the variable max_people, and returned.

6. initialize_max: A method called only from find_max, which creates a list of zeros that equals the length of the graph, then makes only one vertice (the origin airport) equal to infinity.

7. adjacency_list: This method ends with a list of lists. It begins with an empty list, which is populated with a list per row, recording which of the row elements has an edge weight. Its purpose is to provide all the possible routes available in the network, and their weights, in a single variable. For example, for a matrix like this:

```
[[ 0 10  5  0]
 [ 0  0  1  3]
 [ 0  0  0  1]
 [ 0  0  0  0]]
```

The adjacency_list output will be:

[[1, 2], [2, 3], [3], []]

Each list in adjacency_list represents each row in the matrix, with each row's elements recorded only when their value is greater than zero. Note that because it records edge weight locations, the output will remain the same regardless of the source node.

5: Validation & Testing

Team members were able to test the code on their local machines for basic validation. One team member had access to the dashboard and was able to test within that environment.

Test I: We began with a test of “ground truth”, with a goal of determining whether the code performed correctly at a basic level. For this level of testing we employed the code’s original input graph, which had four vertices, five edges, and a maximum of three connections to one vertex. We ran the code with the following variations added individually, and in combinations:

- vertices with more than three connecting edges: PASS
- vertices without edges: PASS
- designated source node as something other than one: PASS

- input where more than one layover would be required to make the traversal: FAIL

Errors/Faults:
→ For more than one layover, the code does output a number, but should be stopped from doing this... should instead give a message like "more than one layover required for this trip". This is eliminated when the layer of the dashboard interface is in place.

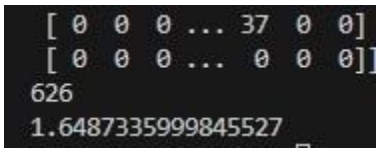
→ If a source and destination combination are entered that do not have a route between them, we simply receive an error, both in the Python environment and in the dashboard.

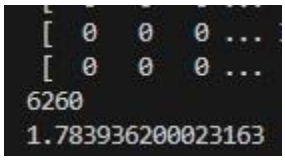
Test II: Once we received the final dataset of over 67,000 rows, and the dashboard was in place, we were able to test for correctness with more varied data. For this round of testing, we investigated accuracy for cases when:

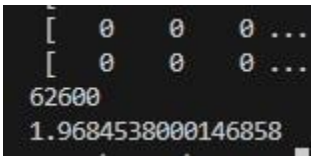
- A layover was entered: PASS
- A route has multiple layover routes available at multiple intermediate-node airports: PASS
- A route's capacity output is only including the correct routes at the destination airport: PASS
- A route's capacity output is only including the correct routes at the origin airport: PASS

Test III: Runtime was explored by taking the dataset of 67,343 routes, and feeding it multiple times into the algorithm using the glob library in Python. Essentially the same data was copied into a specific file folder a certain number of times, and then the code requested all .csv files in that folder to be fed into the find_max() method.

The resulting runtimes from original dataset, followed by a 10x dataset, and then a 100x dataset, are shown here:







The bottom number shows the runtime, and the number just above (626, 6260, and 62600) represent the edge capacity returned as output for each timed round.

We can see the runtimes increase for each exponential input, and notably, they do not increase as high as a worst-case runtime $O(V^2)$. This is due to the real-world case that not every airport origin has a flight available to every destination. If we had a strongly connected graph, we would expect an exponential time increase to match. For the purposes of testing, however, these runtimes are reasonable and expected.

6: Possible Project Extensions and Future Builds

The various tests performed on the dataset highlighted some basic additional features we could add, such as error handling and more complex trips, perhap round-trips as well. But from a practical perspective, while working on this data, the question arose: Why are we interested to know maximum capacities from city to city? Two possible answers seem reasonable.

First, perhaps a pending disaster is known to be imminent, such as a hurricane or tsunami, and emergency management deems evacuation necessary. In a situation like this, the question of how many people can leave one city to another becomes of high importance. We might alter the design in this case to include all seats on every aircraft, not just the passenger values, so that pilots and crew are counted as safe from harm. We would also edit the code so that transport from a source node to a range of destinations is examined, rather than a single destination. Additionally, it may be necessary to temporarily omit some edge weights, if those edges represented nodes in perilous locations due to nearby weather conditions. In other words, plan to offer flights that go away from the disastrous weather, instead of toward it.

Secondly, a real-world scenario might require maximum capacity due to a planned event in a certain city (like the Olympic Games) and airlines for a certain timeframe may have no seats available. City planning, the Olympics committees, or the airlines themselves may want this information in advance. For a situation like this, we would expect to maintain the passenger capacity definition as before, alter the code so that we could determine maximum passengers into the destination node from any source, and perhaps modify functionality to accommodate varying numbers of layovers.

In terms of realistic use cases, the most notable omission of data in this project's current form is the absence of arrival and departure times, as well as dates. Thus, the most practical extension of this project would be to gain real-time information about flights, the fleet of aircraft each airline maintains, their current locations, and which (if any) may be grounded for maintenance or repair. Additionally, international flights could be included. On the passenger side, each airline's available seats status would also be essential for practical use cases.

This would entail purchased data in the form of an ongoing subscription from a resource like FlightAware.com, or perhaps FAA or government-managed versions of the same type of data. With an extended version of the project accessible on the proper hardware, the project could feasibly be used for businesses in need of knowledge about commercial airlines' upcoming seats available, consumers who plan to purchase airline tickets, or enterprises like SeatGuru.com which cater to savvy travelers.

Files for this project can be found at https://github.com/MatthewLeeWilcox/FlightsGraph

A readme style guide to the Github files:

Original data:	airlines.csv airports-extended.csv planes.dat.txt routes.csv
Scraped data:	aircraft-data.pkl
Scraping code:	scraper-tool.py
Preprocessing code:	preprocessing_routes_w_capacities.ipynb
Data set:	FINAL_ALL_DATA_DEC_11.csv
Network flow code:	graph_class.py
User interface:	final dashboard.py

Sources Cited:

Data Acquisition: Web Scraping
https://www.datacamp.com/

https://www.scrapingdog.com/blog/web-scraping-with-xpath-and-python/

https://www.geeksforgeeks.org/web-scraping-using-lxml-and-xpath-in-python/

Algorithms:
https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

https://ncf.instructure.com/courses/7499/files/folder/Lecture%20slides?preview=727189

Dashboard:
https://www.youtube.com/playlist?list=PLCC34OHNcOtoC6GglhF3ncJ5rLwQrLGnV

https://docs.python.org/3/library/tkinter.html