GROUP PROJECT REPORT

Sara Haman, Conor Howlett, and Adam Lashley

Table of Contents.

1. INTRODUCTION **2**

2. DATA PRE-PROCESSING **2**

2.1 INITIAL CLEANING 4

2.2 IDENTIFYING CAPACITIES 7

2.3 FINAL REVIEW 8

3. DESIGN **10**

4. ALGORITHMS USED **10**

5. IMPLEMENTATION **11**

6. RESULTS **12**

7. VALIDATION **13**

8. DISCUSSION **14**

8.1 FUTURE DIRECTIONS 15

RESOURCES **16**

1. INTRODUCTION

OpenFlights.org is a website which allows users to map historical flights from around the world. It is an innovative tool for those curious about flight patterns, which allude to both the connectivity and physical flow of humanity. OpenFlights.org is also an open source tool, which means that their years of data about flight patterns is free to download directly from the website. This ease of access made OpenFlights.org an ideal source of data that can be manipulated into a form relevant to our coursework. The movement of planes between airports can be modeled as a network. The flight path becomes a directed edge, and the airports at which planes converge become the nodes. The maximum capacity of each edge is designated by the amount of people each plane can seat. Passengers are passed from node to node until they reach their destination. The manifestation of airline transit as a network is conceptually strong. However, transforming real airline data into a functional network is, predictably, more difficult. This project uses two data sets from OpenFlights in order to construct a network of flights from New York to San Francisco. These data are combined with supplementary data on plane capacity to calculate the maximum number of passengers who can be moved from New York to San Francisco. We also investigate which airline has the capability to move the most people between these two destinations.

The following document explains our work-flow; expanding on our design decisions, the algorithms we chose to use, and our eventual solution to these problems.

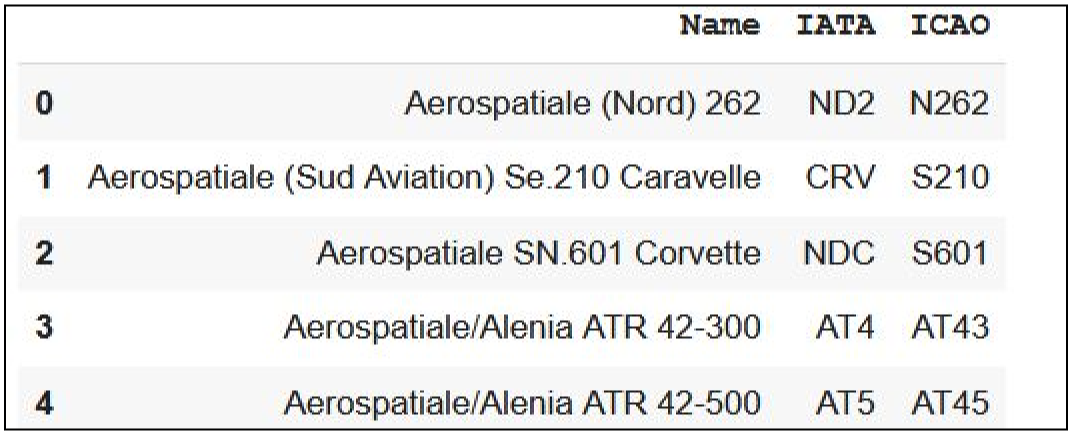
2. DATA PRE-PROCESSING

Each team was provided with two initial datasets sourced from OpenFlights.org. The first was a data set containing information on individual planes. It contained three variables:

Name: The full name of a specific aircraft

ICAO: The four-letter code used by the International Civil Aviation Organization (ICAO), which is an appendant body of the United Nations, to identify specific airline equipment (in this case, the plane). Used for technical and governmental purposes.

IATA: The three-digit code used by the non-governmental International Air Transport Association (IATA) to uniquely identify airline equipment (in this case, the plane). These codes are typically used in the context of commercial flights and safety audits.



	Name	IATA	ICAO
0	Aerospatiale (Nord) 262	ND2	N262
1	Aerospatiale (Sud Aviation) Se.210 Caravelle	CRV	S210
2	Aerospatiale SN.601 Corvette	NDC	S601
3	Aerospatiale/Alenia ATR 42-300	AT4	AT43
4	Aerospatiale/Alenia ATR 42-500	AT5	AT45

The second data set contains information about specific flight routes. Each row in the document was initially organized to be a specific flight between a source airport and a destination airport by a specific airline company. This dataset had nine variables pertaining to information about these flights. These variables were:

Airline: A two-digit code that uniquely identifies each airline

AirlineID: A three-number code that uniquely identifies each airline

Source: A three-letter code that uniquely identifies the airport that the flight began at

SourceID: A four-number code that uniquely identifies the airport the flight began at; corresponds to the Source attribute

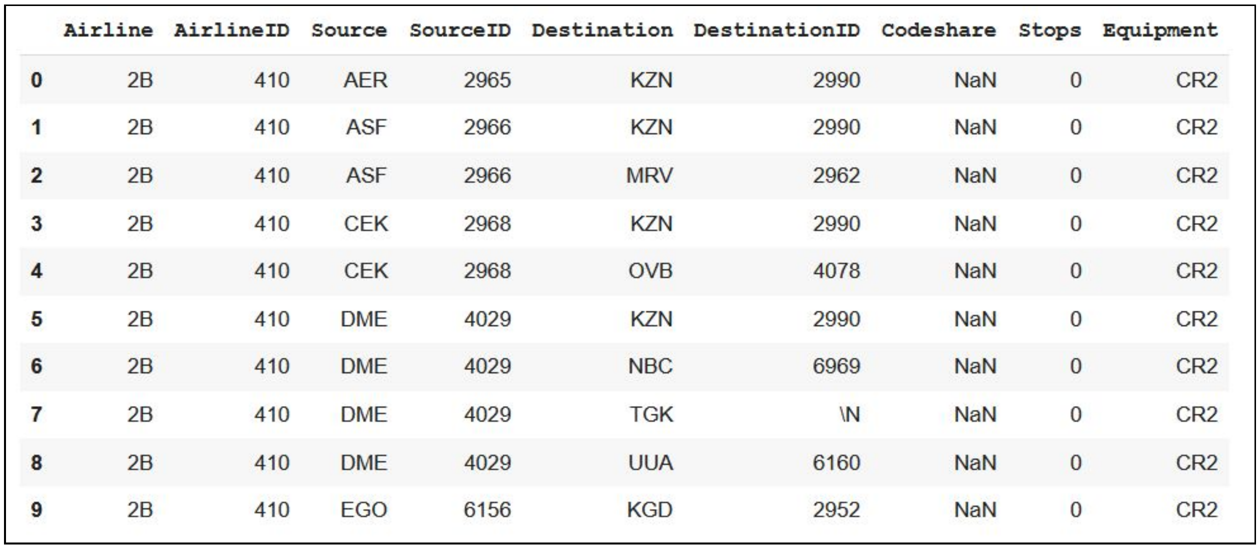
Destination: A three-letter code that uniquely identifies the airport that the flight ended at

DestinationID: A four-number code that uniquely identifies the airport the flight ended at; corresponds to the Destination attribute

Codeshare: A boolean value which indicates whether or not the airlines had a codeshare agreement

Stops: An alphanumeric value between 1 and 0 which indicates how many layovers were on this flight

Equipment: A vector of three-digit IATA codes which indicate which planes flew this specific route

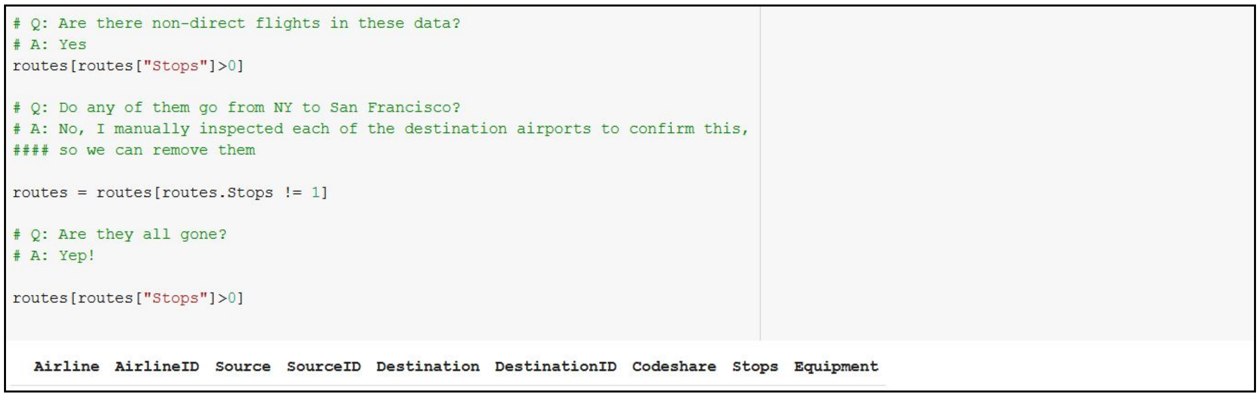


	Airline	AirlineID	Source	SourceID	Destination	DestinationID	Codeshare	Stops	Equipment
0	2B	410	AER	2965	KZN	2990	NaN	0	CR2
1	2B	410	ASF	2966	KZN	2990	NaN	0	CR2
2	2B	410	ASF	2966	MRV	2962	NaN	0	CR2
3	2B	410	CEK	2968	KZN	2990	NaN	0	CR2
4	2B	410	CEK	2968	OVB	4078	NaN	0	CR2
5	2B	410	DME	4029	KZN	2990	NaN	0	CR2
6	2B	410	DME	4029	NBC	6969	NaN	0	CR2
7	2B	410	DME	4029	TGK	\N	NaN	0	CR2
8	2B	410	DME	4029	UUA	6160	NaN	0	CR2
9	2B	410	EGO	6156	KGD	2952	NaN	0	CR2

The documents were stored in a shared Google Drive, which was mounted in our preprocessing code so that the path to the files would not have to be altered depending on who was accessing the document. Both documents were read into a Python notebook as pandas dataframes.

2.1 INITIAL CLEANING

Before we could manipulate these data into the graph object needed to construct a network, we had to process the data into useful format. We aimed for an edgelist which contained the source and destination airports on a route flown by a specific plane. The planes dataframe did not provide us with any particularly useful information, other than as a resource for finding the name of planes. As such, we focused on the routes dataset in order to construct the edgelist. Our first step was to normalize the data frame. We investigated information about the dataframe to decide which attributes to normalize the table on. During this process, we discovered that there were almost no flights in the dataset containing a layover. In fact, there were only 11 flights with at least one layover. Manual investigation of the eleven flights showed that none of them traveled from a New York City airport to a San Francisco airport. Because the prompt for this project stipulated that flights could have at maximum one layover between the source and the destination, we removed these flights from the data set.



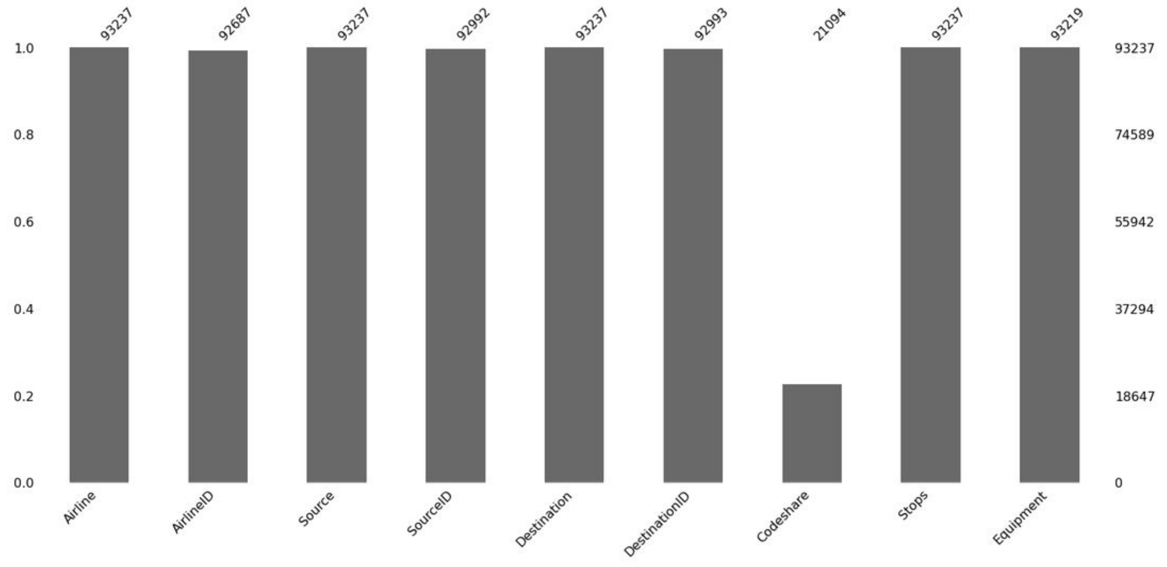
```
# Q: Are there non-direct flights in these data?
# A: Yes
routes[routes["Stops"]>0]

# Q: Do any of them go from NY to San Francisco?
# A: No, I manually inspected each of the destination airports to confirm this,
#### so we can remove them

routes = routes[routes.Stops != 1]

# Q: Are they all gone?
# A: Yep!

routes[routes["Stops"]>0]
```

```
  Airline  AirlineID  Source  SourceID  Destination  DestinationID  Codeshare  Stops  Equipment
```

Once these flights were removed, we moved on to missing data. A personal investigation of the data revealed that missing data was encoded in two ways: through string NaNs and through the value '/N'. Both of these values were converted into numpy NaN values. In order to decide which variables to normalize on, we thought it important to know how much missing data we were working with. For instance, more missing data in SourceID than Source, this would indicate that Source would work better as a 'source airport' attribute than SourceID. This was indeed the case. We found that consistently the numeric ID columns contained missing data whereas the letter ID columns had none. As such, we decided to select Source and Destination to represent the source and destination airports and Airline over Airline ID to represent the airline. . IATA codes are also the codes used to identify commercial airports, and as such are more easily interpretable by a general audience.

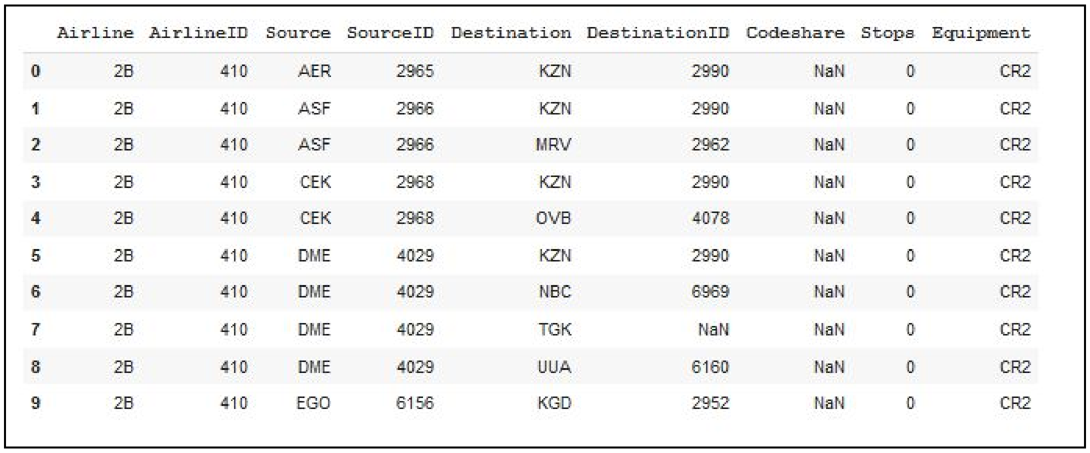
```
# CHECKING TO SEE THE NULL CONTENT OF THE ROUTES
msno.bar(routes)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7eff75a872b0>
```



We also noticed that the equipment column was missing data. There were eighteen rows containing null equipment values. Before dropping these rows, we performed due diligence to confirm that these data could not be manually input. The OpenFlights.org datasets were last updated in 2014. As such, many of the routes in these data are different than the routes flown now. Especially given the pandemic and the decreased air traffic because of this. We searched for flights between the source and destination airports by the named airlines in order to attempt to identify what planes were being flown on these routes. Unfortunately, our investigation turned up no information and a search for historical flight data matching the content of the OpenFlights.org was also futile. Knowing the capacity of the planes was integral to our project. If there is no plane, there can be no capacity. We cannot simply throw people across the world. Given that it was nearly impossible, if not truly impossible, to find the planes that flew these routes, rows with null equipment values were dropped.

Now that we had the columns selected, it came time to handle the equipment. The equipment, which indicated what airplanes were used on specific routes by specific airline companies, were encoded in strings containing multiple aircraft. In order to normalize the data, we wanted to transform it so that each row was a specific route flown by a specific airline company by a specific aircraft so that we could assign that row a capacity. Assigning single capacities to each row would allow us to determine which edges to keep when designing our network later on in the process. Because the equipment were encoded uniformly, meaning that rows with multiple equipment listed were written in the form "CODE1 CODE2 CODE3" with a space in between, were were able to create a new temporary data frame of values by splitting the equipment strings at the spaces. The new equipment data frame was rejoined with the original dataframe using retained index values; the size of the data frame increased by approximately 50% of the original dataframe indicating that the split had been successful.



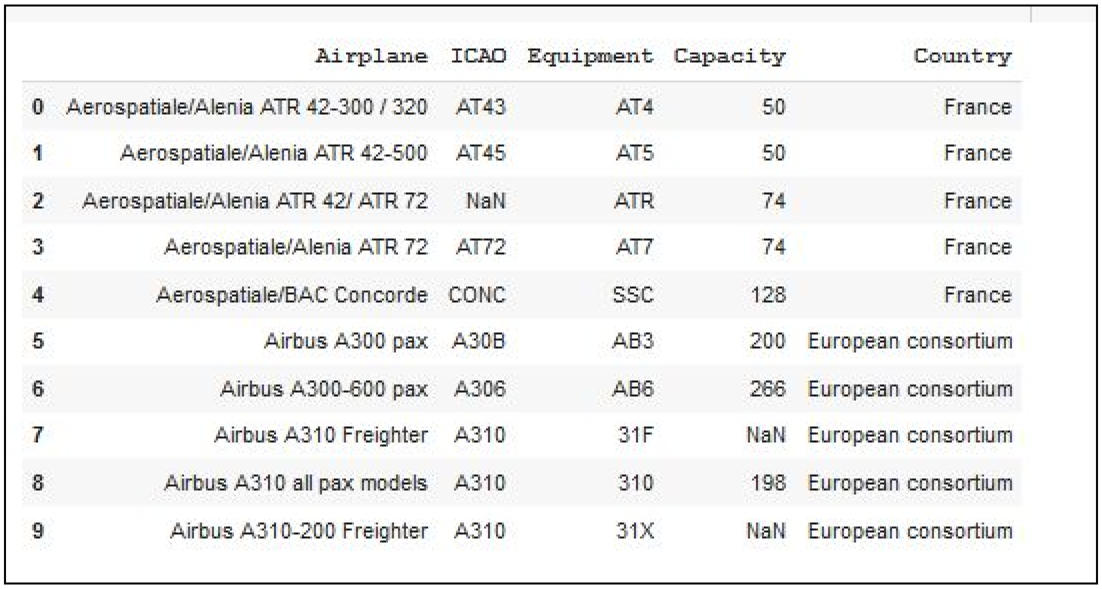
	Airline	AirlineID	Source	SourceID	Destination	DestinationID	Codeshare	Stops	Equipment
0	2B	410	AER	2965	KZN	2990	NaN	0	CR2
1	2B	410	ASF	2966	KZN	2990	NaN	0	CR2
2	2B	410	ASF	2966	MRV	2962	NaN	0	CR2
3	2B	410	CEK	2968	KZN	2990	NaN	0	CR2
4	2B	410	CEK	2968	OVB	4078	NaN	0	CR2
5	2B	410	DME	4029	KZN	2990	NaN	0	CR2
6	2B	410	DME	4029	NBC	6969	NaN	0	CR2
7	2B	410	DME	4029	TGK	NaN	NaN	0	CR2
8	2B	410	DME	4029	UUA	6160	NaN	0	CR2
9	2B	410	EGO	6156	KGD	2952	NaN	0	CR2

We then moved on to identifying the capacities for each plane.

2.2 IDENTIFYING CAPACITIES

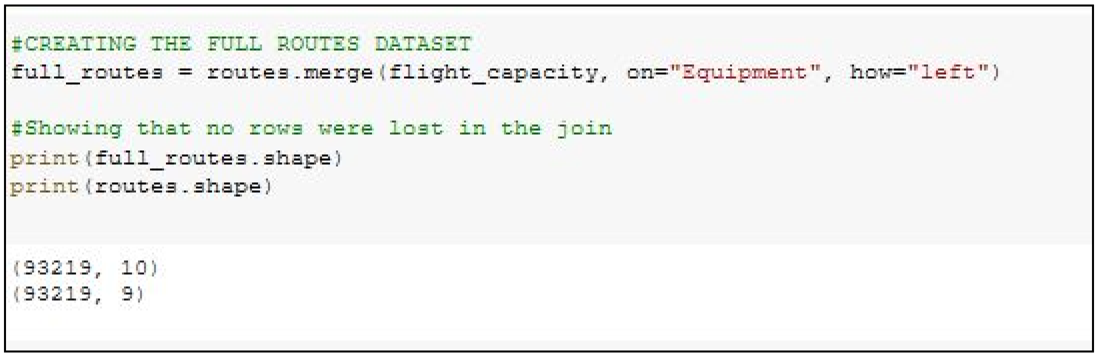
We considered multiple routes for identifying the capacities for each plane, including manually inputting the data, which would have been incredibly time intensive, and scraping a webpage to collect data which we could match with ours on either the name of the plane or their IATA code. However, we instead launched a collaborative campaign with another group to scour the internet for an existing dataset which contained either the IATA code for many planes as well as their capacities. Such a dataset did exist. Laboratoire Spécification et Vérification, a French flight academy, put together a training dataset containing data on hundreds of internationally used aircraft.

The data were in CSV format and had no unintentional missing data (within the data, null values represented planes that did not have a capacity for passengers typically because they were cargo planes).



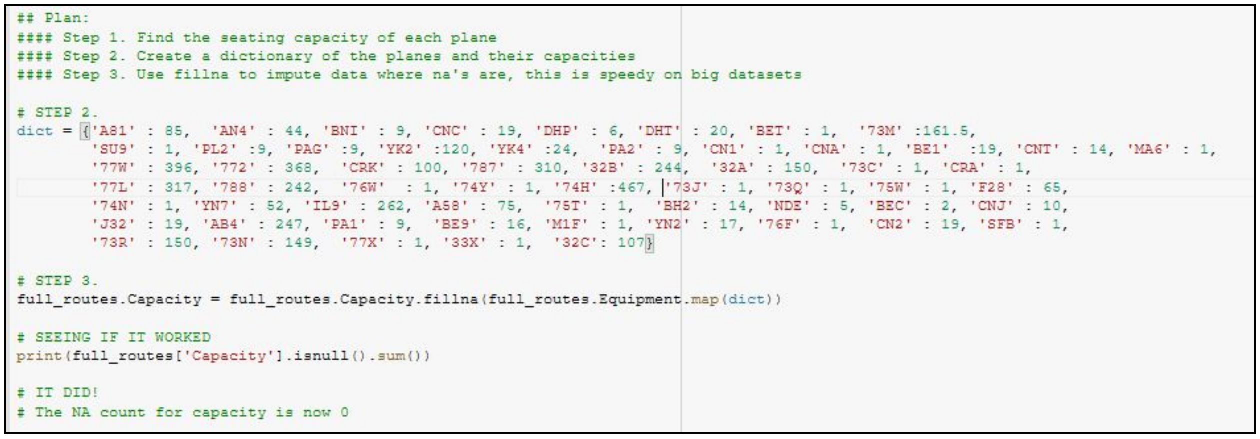
	Airplane	ICAO	Equipment	Capacity	Country
0	Aerospatiale/Alenia ATR 42-300 / 320	AT43	AT4	50	France
1	Aerospatiale/Alenia ATR 42-500	AT45	AT5	50	France
2	Aerospatiale/Alenia ATR 42/ ATR 72	NaN	ATR	74	France
3	Aerospatiale/Alenia ATR 72	AT72	AT7	74	France
4	Aerospatiale/BAC Concorde	CONC	SSC	128	France
5	Airbus A300 pax	A30B	AB3	200	European consortium
6	Airbus A300-600 pax	A306	AB6	266	European consortium
7	Airbus A310 Freighter	A310	31F	NaN	European consortium
8	Airbus A310 all pax models	A310	310	198	European consortium
9	Airbus A310-200 Freighter	A310	31X	NaN	European consortium

We read in and performed a preliminary cleaning routine on the LSeV dataset to investigate potential problematic encoding. We then pruned the dataset to only the relevant columns, which were the IATA code of the plane (to join on) and the capacity of the plane. The join between the LSeV data and the Routes data went near seamlessly; the number of rows remained identical indicating no lost or duplicated rows.



```
#CREATING THE FULL ROUTES DATASET
full_routes = routes.merge(flight_capacity, on="Equipment", how="left")

#Showing that no rows were lost in the join
print(full_routes.shape)
print(routes.shape)


(93219, 10)
(93219, 9)
```

However, after examining capacities for missing data, we found that there were approximately 50 IATA codes in the Routes data that were not represented in the LSeV data. We were not able to find another clean dataset containing a more thorough dataset containing both IATA codes and plane capacities. As such, we decided to input values for these planes by hand. One member of our team located two thorough lists of planes and IATA codes - more thorough documents than even in the original Planes data. We cross-referenced these lists to find the names of the planes corresponding to the IATA codes of the planes with missing values. We found that most of these planes were either primarily cargo planes, personal aircraft, helicopters, or planes made by the Russian company Yakolev. We created a dictionary from the IATA codes of the missing planes set as the key and the capacities as the values. We then used the fillna function to map values from the dictionary onto rows with null capacities based on the IATA value. This method worked and a subsequent count of the null capacity values returned 0.



```
## Plan:
#### Step 1. Find the seating capacity of each plane
#### Step 2. Create a dictionary of the planes and their capacities
#### Step 3. Use fillna to impute data where na's are, this is speedy on big datasets

# STEP 2.
dict = {'A81' : 85,  'AN4' : 44, 'BNI' : 9, 'CNC' : 19, 'DHP' : 6, 'DHT' : 20, 'BET' : 1,  '73M' :161.5,
        'SU9' : 1, 'PL2' :9, 'PAG' :9, 'YK2' :120, 'YK4' :24,  'PA2' : 9, 'CN1' : 1, 'CNA' : 1, 'BE1'  :19, 'CNT' : 14, 'MA6' : 1,
        '77W' : 396, '772' : 368,  'CRK' : 100, '787' : 310, '32B' : 244,  '32A' : 150,  '73C' : 1, 'CRA' : 1,
        '77L' : 317, '788' : 242,  '76W'  : 1, '74Y' : 1, '74H' :467, '73J' : 1, '73Q' : 1, '75W' : 1, 'F28' : 65,
        '74N' : 1, 'YN7' : 52, 'IL9' : 262, 'A58' : 75,  '75T' : 1,  'BH2' : 14, 'NDE' : 5, 'BEC' : 2, 'CNJ' : 10,
        'J32' : 19, 'AB4' : 247, 'PA1' : 9,  'BE9' : 16, 'M1F' : 1, 'YN2' : 17, '76F' : 1,  'CN2' : 19, 'SFB' : 1,
        '73R' : 150, '73N' : 149,  '77X' : 1, '33X' : 1,  '32C': 107}

# STEP 3.
full_routes.Capacity = full_routes.Capacity.fillna(full_routes.Equipment.map(dict))

# SEEING IF IT WORKED
print(full_routes['Capacity'].isnull().sum())

# IT DID!
# The NA count for capacity is now 0
```

2.3 FINAL REVIEW

Now that we have the merged dataframe, it was time to prepare the preliminary edgelist. This edgelist will go on to be further refined when it is transformed into a graph object later down the pipeline.

A final investigation of the data revealed that there were no missing data in the present dataset. We created the edge list by selecting the four columns of interest. To reiterate, these were:

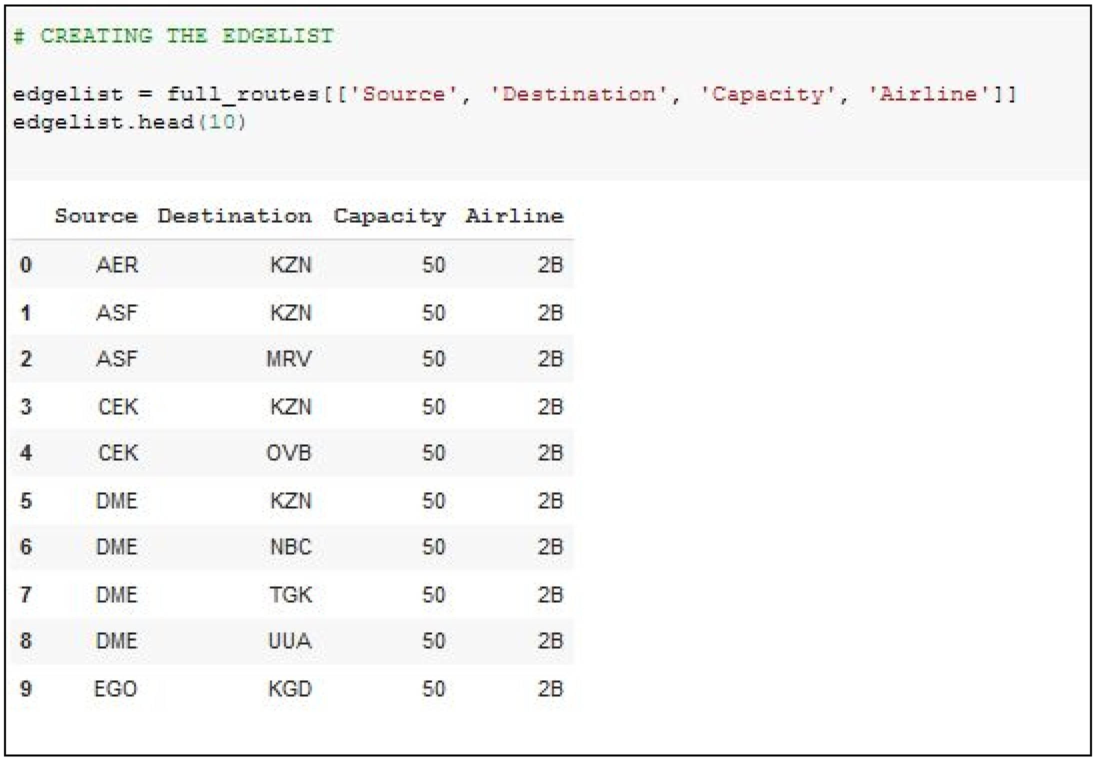
Source: A three-letter IATA code unique to each airport. This variable identifies the airport a flight begins at.

Destination: A three-letter IATA code unique to each airport. This variable identifies the airport a flight ends at.

Capacity: Numeric value indicating the maximum number of passengers that can be seated on a flight

Airline: A unique two-digit code representing the airline company

This edgelist was saved as a CSV titled "flights_edgelist.csv" and passed down the pipeline to be turned into the network.



```
# CREATING THE EDGELIST

edgelist = full_routes[['Source', 'Destination', 'Capacity', 'Airline']]
edgelist.head(10)
```

	Source	Destination	Capacity	Airline
0	AER	KZN	50	2B
1	ASF	KZN	50	2B
2	ASF	MRV	50	2B
3	CEK	KZN	50	2B
4	CEK	OVB	50	2B
5	DME	KZN	50	2B
6	DME	NBC	50	2B
7	DME	TGK	50	2B
8	DME	UUA	50	2B
9	EGO	KGD	50	2B

3. DESIGN

For this project we wanted our program execution to be as optimized as the algorithms within it. In order to accomplish this goal we limited our global variables and laid out our functions in such a way that we could utilize the inherent optimization strengths of object oriented programming. In addition to these design choices we chose to begin our program by pruning our network down to the bare elements required in order to accomplish the task at hand. We limited our use of pandas dataframe operations because we quickly realized that their inefficiencies quickly offset any time saving that we had achieved through other changes. These design choices culminated into a program that is able to execute in 21 seconds on average and return all of the results required for the project.

4. ALGORITHMS USED

For this project we utilized Breadth First Search (BFS) and the Edmonds-Karp (EK) methodology of the Ford Fulkerson algorithm in tandem. While reviewing the requirements for the assignment we quickly realized that we would be dealing with a maximum flow problem that would require augmenting paths. Edmonds Karp is a special implementation of the Ford Fulkerson algorithm that utilizes BFS in order to efficiently process the maximum flow from a 'source' to a 'sink' in a flow network. BFS is utilized in this implementation in order to increase computational efficiency by ensuring that the Ford Fulkerson implementation is always operating on the path with the shortest number of edges.

BFS is a graph traversal algorithm that prioritizes visiting the nodes closest to a given source node before moving to nodes that are farther or 'deeper' in the graph. BFS utilizes a boolean marker to 'mark' a node once visited, ensuring that the algorithm does not visit the same node twice. We chose to use BFS because it is integral to implementing the Edmonds-Karp methodology of the Ford Fulkerson algorithm. There are alternative implementations of BFS that utilize maximum depth but because the maximum length of our paths are managed by our preprocessing functions so that we were able to use a traditional BFS implementation.

5. IMPLEMENTATION

Our implementation was executed with a few key areas of importance: functionalized code, accurate code, and efficient code. The bulk of the processing time in our program is due to Pandas' aggregate and conditional dataframe grouping speed. Since Pandas is Pythonic standard for reading CSVs, storing them as dataframes and performing vectorized operations on the data, we opted to utilize this tool. When tasked with aggregating pairwise flights from starting and ending airports, we found Pandas' groupby() and agg() functions suitable. Once we had the aggregated pairwise airport to airport capacity numbers, we iterated through the rows of the dataframe and appended to an edge list the source, destination and capacity for each aggregated pairwise relationship. This is carried out in our "groupPaths" function. Since we were exclusively tasked with exploring flights that only have a maximum of one layover, we found that a lot of processing time could be saved by trimming the edge list so that it only contained flights either leaving a New York airport or entering a San Francisco airport. This would remove the possibility of cyclic patterns, would limit the edge list to one layover at most, and would ensure flow could only travel in one direction. The function "trimEdgelist" solved this task and trimmed based off of the inserted sources and sinks.

In order to reduce this problem to one simple max flow problem, we needed to create one source and one sink. These nodes would be made up, fictitious airports in our edge list but would be crucial for solving the problem, effectively. The single source and sink nodes were created and given an infinite capacity. The infinite capacity was possible through numpy's "inf" function. This infinite flow would ensure that we do not accidentally create a bottleneck that doesn't exist in the data. The super source would feed into every source node and the super sink would receive flow from each of the sink nodes. These edges were added to our edge list using our "superSourceSink" function. Our next task was to create an adjacency matrix for our edges, but this would not be possible with nodes with string values as their identifiers. We opted for a dictionary so that each airport code could have a unique integer value for use in a matrix. This dictionary was created in our "createDict" function and created a list of unique airports in our edgelist, and a ranged list of integers of equal length to the airports list, then zipped them together into the dictionary. With this dictionary key for our airports, we could create an identical edgelist, but using integers instead of strings for the nodes. This was carried out using the "newEL" function. The edge list was now ready to be converted into an adjacency matrix. The "display" function first instantiated a numpy zeros matrix with dimensions of the length of the total unique airport nodes. The edge capacities were then appended to the matrix in a simple for-loop, iterating through the edge list.

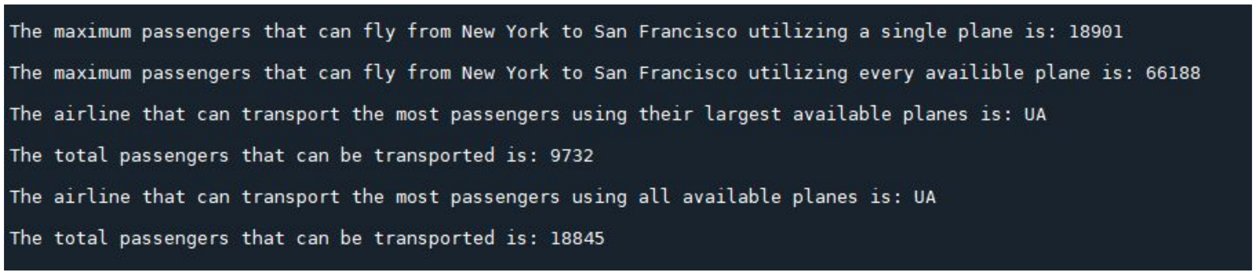
Once the adjacency matrix was completed, a simple call to our “Graph” class would allow us to implement BFS and Edmonds Karp.

These same functions were recycled to answer the question of best airline, with the addition of a couple of extra helper functions. The “carriersList” function takes the Airline carriers from the relevant data and appends them to a list. The carriers list works in tandem with the “trimCarriers” function, which trims the dataframe to only include flight information for the specified carrier input. To solve for the different airlines, we simply iterate through the carriers list and trim the edge list by carrier, then create a graph and run our max flow algorithm on it. The highest value for max flow is stored and updated whenever another airline carrier can transport more passengers in the loop. We then simply print the name of the best carrier and the amount of passengers they can move from New York to San Francisco with no more than one layover flight.

6. RESULTS

Our code was designed to solve for a few constraints, and return a satisfactory solution for multiple semantic differences in the question. When looking at the airline data, one could interpret the equipment for each flight as all being available, or as one single unit being available. For example, if only one piece of the available equipment were used per flight, then we would want the plane with the largest passenger capacity. If each piece of listed equipment were available to be used simultaneously, then we would want to sum the plane’s capacities together. We solved both of these possibilities for the first question. If the largest plane was utilized for each flight, then 18,901 passengers could be flown from New York City to San Francisco. This includes flights from any of the three NYC airports to any of the four San Francisco airports, with no more than one layover. If each listed piece of equipment were to be utilized, then the amount of passengers that can be flown is substantially increased, since many airlines reported two or three planes being available to fly the route. This totaled 66,188 total passengers.

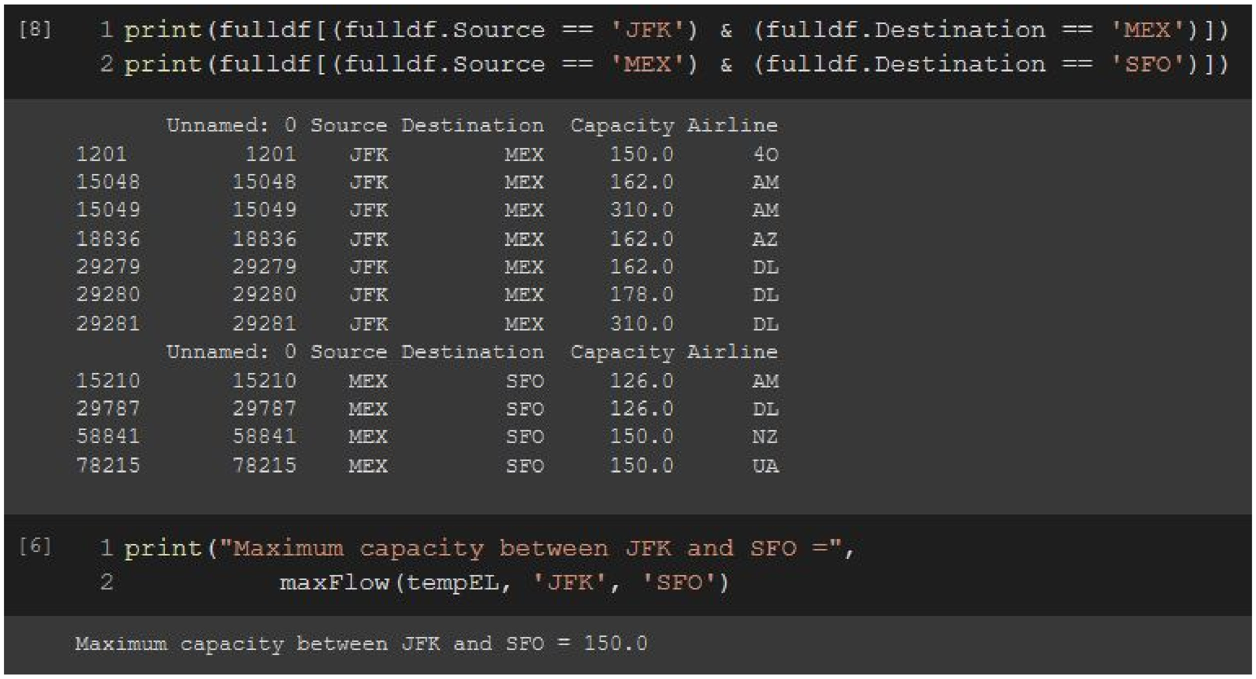
The solution for question two, or the airline that could transport the most passengers from NYC to San Francisco, was interestingly the same when solving for the largest available piece of equipment and for summed available equipment. United Airlines won out with 9,732 passengers flown utilizing the largest capacity plane per flight, and 18,845 flown utilizing all available planes simultaneously. The output of our code shows these results below.



```
The maximum passengers that can fly from New York to San Francisco utilizing a single plane is: 18901
The maximum passengers that can fly from New York to San Francisco utilizing every availible plane is: 66188
The airline that can transport the most passengers using their largest available planes is: UA
The total passengers that can be transported is: 9732
The airline that can transport the most passengers using all available planes is: UA
The total passengers that can be transported is: 18845
```

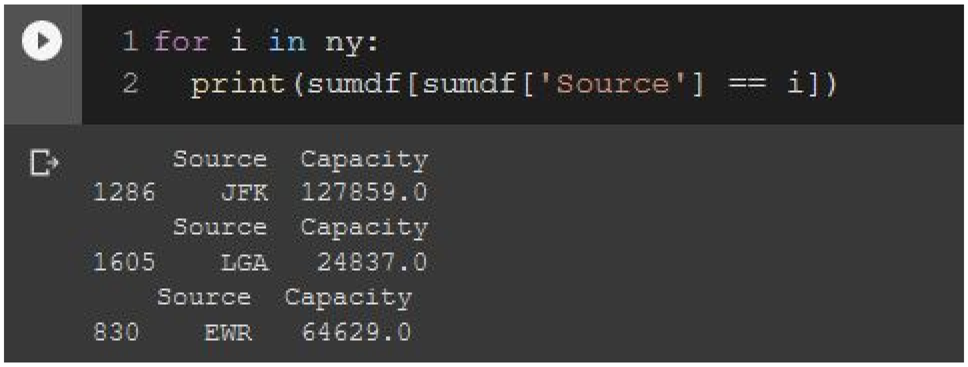
7. VALIDATION

In order to validate our BFS and EK implementations we employed the help of the python package networkx. It was not feasible for us to review and follow each augmenting path in the full network so we decided to break the network into smaller individual networks and sanity check that our implementation of EK was behaving as expected. The following code demonstrates that our EK implementation accurately finds the maximum flow value from ‘JFK’ to ‘MEX’ to ‘SFO’. While this is by no means a validation of the entire network we feel safe to infer that our EK implementation is behaving as expected. As we can see our implementation accurately finds that the maximum capacity of the JFK -> MEX -> SFO path is 150.



```
[8]  print(fulldf[(fulldf.Source == 'JFK') & (fulldf.Destination == 'MEX')])
     print(fulldf[(fulldf.Source == 'MEX') & (fulldf.Destination == 'SFO')])

       Unnamed: 0 Source Destination  Capacity Airline
1201         1201    JFK         MEX     150.0      4O
15048       15048    JFK         MEX     162.0      AM
15049       15049    JFK         MEX     310.0      AM
18836       18836    JFK         MEX     162.0      AZ
29279       29279    JFK         MEX     162.0      DL
29280       29280    JFK         MEX     178.0      DL
29281       29281    JFK         MEX     310.0      DL
       Unnamed: 0 Source Destination  Capacity Airline
15210       15210    MEX         SFO     126.0      AM
29787       29787    MEX         SFO     126.0      DL
58841       58841    MEX         SFO     150.0      NZ
78215       78215    MEX         SFO     150.0      UA

[6]  print("Maximum capacity between JFK and SFO =",
               maxFlow(tempEL, 'JFK', 'SFO'))

Maximum capacity between JFK and SFO = 150.0
```

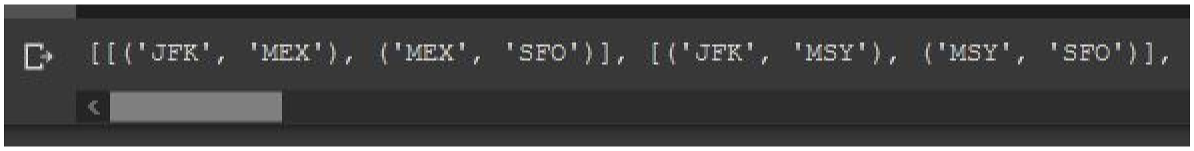
In order to further validate our EK implementation we checked the net capacity of our source nodes to ensure that the achieved maximum capacities received at our sinks were within presumably acceptable limits. The following code output indicates that the net outflow from our source nodes is well above the discovered capacities at the sinks. We can see that the observed output capacity was 217,325 while our maximum utilization was 66,188.



```
for i in ny:
  print(sumdf[sumdf['Source'] == i])
```

```
     Source  Capacity
1286    JFK  127859.0
     Source  Capacity
1605    LGA   24837.0
    Source  Capacity
830    EWR   64629.0
```

Validating our BFS implementation involved ensuring that the edge list generated by the BFS maintained a proper depth (3 in our case, represented by two pairs of tuples [(NY, layover), (layover, SF)]. Based on the way in which we pruned the network before running our algorithms, the depth of the BFS was ensured to be limited to the expected behavior without the addition of 'maximum depth' functionality to the existing BFS solution. Our validation of our implementation of BFS is illustrated by the following edgelist output.



```
[[('JFK', 'MEX'), ('MEX', 'SFO')], [('JFK', 'MSY'), ('MSY', 'SFO')],
```

8. DISCUSSION

The results of our max flow algorithm did not vary significantly from what we expected. There are hundreds of individual routes from New York Airports to San Francisco airports. Given that most commercial flights can seat upwards of one hundred people, we can expect that the theoretical maximum number of individuals who can be moved across the country numbers in the thousands.

Furthermore, United Airlines being able to move the most people was reasonable when contextualized by the OpenFlights.org data. According to the OpenFlights.org website (2), United Airlines was the most prevalent airline in their data. If United Airlines has more routes than any other airline, they will be able to transport more people in the context of the flow network. In reality, United Airlines has the third largest fleet of any U.S. airline. They also come third in the number of passengers they transport nation-wide each year. The reason they are the most represented airline in the OpenFlights flight data may be simply because their planes fly more routes than other carries. As such, they would appear in the data more often. However, it is possible that they fly these routes less often than some other air carriers, which is why they only transport the third highest number of passengers (as of data from 2016).

8.1 FUTURE DIRECTIONS

Of course, the answers we provided are the theoretical maximums. Only under ideal circumstances would all the routes to San Francisco from New York be operating at once. Even more rare is the occasion that all of the planes would be in the air together. In reality, this would be an air traffic disaster. However, the prompt for this project implied that we were aiming for the theoretical maximum more than the practical number of people who could be transported to San Francisco. We were given no timeline, nor the frequency at which planes fly on these routes. One future direction for this project, and one which would be practically useful, would be to combine the flights data with more detailed information about the flights themselves. If more constraints could be put on the network, we could potentially create a model wherein a user could input attributes, such as a particular weather pattern or day of the week, along with the source and destination cities, and receive a maximum value geared to their target circumstance.

A more realistic future direction, and one that we could implement if we had more time for this project, would be clustering the airports by city so that a user simply had to input target source and destination cities instead of a list of the IATA codes for the airports. Making the application more user friendly - something that could be formatted into an app, for instance - would be a long-term end goal if we were ever to revisit this project.

RESOURCES

1. (http://www.lsv.fr/~sirangel/teaching/dataset/).
2. See header “Top Ten Airlines”: (https://openflights.org/about)