Using Graph Theory Algorithms to Optimize Domestic American Airline Routes

Completed for the Certificate in Scientific Computation
Fall 2018

Nikhil Bhargava
Civil Engineering
Civil, Architectural and Environmental Engineering
Cockrell School of Engineering

Howard Liljestrand
Professor
Civil, Architectural and Environmental Engineering

Table of Contents

Abstract

The purpose of this paper is to see how graph theory algorithms, such as Dijkstra’s shortest path algorithm, can be applied to Domestic American airline routes in order to optimize them in a linear fashion when an airport is shutdown. The paper will take an in-depth look at how airline employ hub-and-spoke versus point-to-point models and how they can be combined to create a linear model. Using Neo4j and Python, publicly available data on American Airline routes will be cleaned, filtered and joined as well as extracted for new features. Using a combination of Neo4j and Python, it is possible to turn this table into graph database. Applying Dijkstra’s algorithm to the graph, new routes can be created based on an optimal path in terms of the distance the flight travels as well as adding new stops, the basis of a linear route model.

Introduction

Airline companies all over the world provide services spanning numerous cities across many different countries. These airlines carefully organize which airplanes will operate when, and exactly where each plane will fly. The three dominant route methodologies, when it comes to flight planning, are the hub-and-spoke model (H&S), the point-to-point model (P2P) and a combination of both, the linear model. Each model provides these airline companies with different tradeoffs that need to be systematically evaluated in order to generate the most revenue possible. Figure 1 below is a visualization on which the first two models operate. The circles, or nodes, represent airports, while the arrows, relationships, indicate possible routes that could connect those airports together.

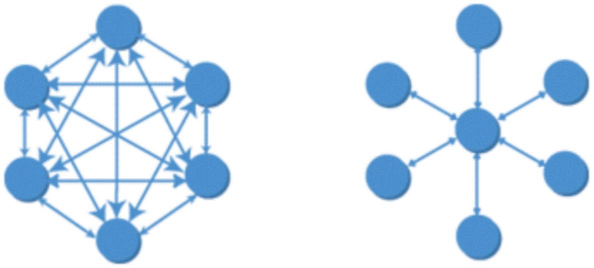


Figure 1: Point-to-Point (left) vs. Hub-and-Spoke (right). Figure from McDermott, John. "The Airline Economics of the Bicycle Wheel: Point-to-Point vs Hub-and-Spoke Flying." *Aeronautics Online*, 3 July 2017, aeronauticsonline.com/the-airline-economics-of-the-bicycle-wheel-point-to-point-vs-hub-and-spoke-flying/.

Hub-and-Spoke Model

In the US and around the world, the hub-and-spoke model has been adopted by a majority of the major airlines. According to Cook and Goodwin, the main advantage to a H&S model comes from "consolidating the travel demand of each spoke city to most or all of the destinations in the network". (Cook and Goodwin) This is done by connecting spoke cities to major hubs and operating through the hub itself. In this way, it is possible for these airlines to reach more parts of the world, "increasing passenger density, network growth and positively affecting both supply and demand." (Cook and Goodwin) H&S models also allow for airlines to match airplane size with demand, as well as expand to other regions in a more efficient manner.

One major flaw in the H&S model is the high operating costs. According to Cook and Goodwin, about 40% of network activity involves a passenger having a hub as either their origin or destination, with the rest using the hub to make another outbound connection, also known as a layover or a stopover. Operating costs, labor costs as well as scheduled and unscheduled delays are just a few additional disadvantages of operating through an H&S model.

Point-to-Point Model

The point-to-point model was at one time extremely popular in the United States before air travel was deregulated in 1978. At the time, flights connecting exclusively small cities were often relatively empty, causing ticket prices to go up and a lack of profit. For big cities, it is sometimes very difficult for P2P models to "match capacity with demand" (Cook and Goodwin) depending on the season, day or time of day.

Point-to-point models, however, are generally advantageous when connecting big cities, as those flights are easier to sell on a regular basis. These flights also have reduced traveled time compared to H&S as they don't require any stoppages during the duration of the flight.

Linear Models

Some of the most interesting point-to-point models seen today are actually a hybrid of both H&S and P2P models known as Linear Systems. These models operate similarly to a bus or train, making several stops between an origin and final destination. Two of the most famous airlines known for operating in such a manner are Southwest Airlines and Jet Blue. Instead of hubs, these airlines design routes based on "focus cities" (Cook and Godwin) which are either passed through or the final destination of a route. This allows for these airlines to provide service to many passengers as well as many different cities.

For comparisons sake, a day's (7/18/18) worth of air traffic for both American Airlines' hub-to-spoke model and Southwest's linear model can be seen in Figures 2 and 3 below. Hugo Larcher's flight data visualization code was adapted to the data collected below (see Data Collection) to create these figures. The regions with "whiter routes" indicate areas where there is a dense collection of routes while the "pinker routes" the opposite. The adapted code (Larcher) used for these visualizations can be found in Appendix 1E.

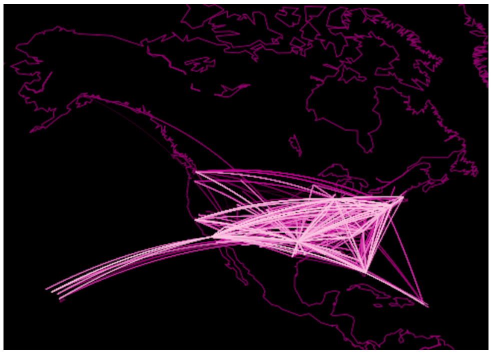


Figure 2: American Airlines Hub-to-Spoke Model

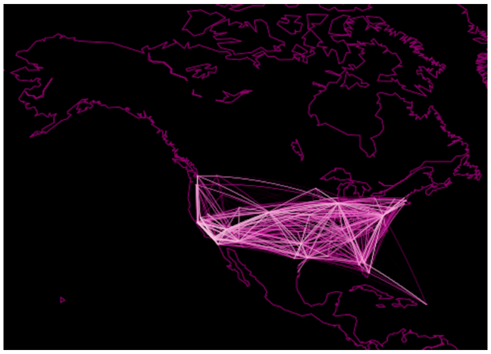


Figure 3: Southwest Airlines Linear Model

Problem Statement

The problem statement this paper will attempt to address is: If an airport is shutdown, due to inclement weather, safety threats, etc., is it possible to add additional stops to an existing route, such as that seen in a linear model? Additionally, would this result in an optimized route for the airline to go from a hub-and-spoke to a linear one?

The standard protocol for when a flight cannot reach its destination is to book passengers from the cancelled flight onto another flight that will then take them to a different hub that has a connecting flight to their original final location. By repurposing these aircrafts to fly a route that is taken by other flights of the airline, a point-to-point style approach can extend a flight path when its original one was disturbed

Based on initial research, it seems possible that a linear network could increase profits for an airline by increasing the number of trips in-between hubs. However, it is also important to take into consideration the current infrastructure of an airline company when coming to that conclusion. To answer the problem statement at hand, it is important to filter data specifically to a domestic airline. For the purpose of this paper, domestic American Airline flight data will be used, as they are the largest domestic airline in the US in regards to the number of annual passengers, according to Sawe, and also implement the hub-and-spoke network system.

Materials and Methods

To address the problem statement, an ensemble of tools was used to approach a conclusion. The two primary tools used in this paper are Python and Neo4j. Python "is an interpreted, object-oriented, high-level programming language with dynamic semantics" (Python) and Neo4j "is an open-source, NoSQL, native graph database." (Neo4j, *"What Is a Graph Database and Property Graph | Neo4j")* In Neo4j, the Cypher Querying language was used to write code. Cypher "is a declarative, SQL-inspired language for describing patterns in graphs visually using an ascii-art syntax." (Neo4j, *Neo4j's Graph Query Language: An Introduction to Cypher*") Although Neo4j can also be run as a package on python, the desktop version was used for memory purposes.

Python code was written for:

- The collection, cleaning, filtering and joining of data
- Generating features and the identification of hubs
- Adapted to visualize flight route data
- Determining flight routes needed to be optimized by Neo4j

Cypher code was written for:

- Creating a graph database
- Creating relationships to connect nodes together
- Querying the graph database
- Optimizing flight routes using Dijkstra's Shortest Path Algorithm

All of the code used for this paper can be seen in Appendix 1 and Appendix 2.

Data Collection

To effectively answer the problem statement posed above, the collection of accurate data was crucial to create an informed conclusion. The data used for this paper was obtained from The Bureau of Transportation Statistics (BTS). The BTS releases data on all domestic United States flights monthly, as well as Airport Location data.

For the month of July 2018, the following airline features were downloaded into a csv: tail_num, op_unique_carrier, fl_date, origin_airport_id, origin_airport_seq_id, origin_city_name, dest_airport_id, dest_airport_seq_id, dest_city_name, distance, actual_elapsed_time, distance, dep_time, arr_time.

Airport information with the following features were downloaded into a csv: airport_seq_id, airport_id, display_airport_name, display_airport_city_name_full, longitude, latitude.

These CSV files, renamed to routes.csv and location.csv respectively, were uploaded into a Jupyter Notebook and transformed into a dataframe. The code to setup the notebook can be seen in Appendix 1A.

Data Cleaning, Filtering, & Joining

Like most data downloaded online, the CSV files downloaded from the BTS needed to be cleaned, filtered and joined together to create a cohesive dataset for analysis. Relevant features filtered from the location dataframe are: airport_seq_id, airport_id, display_airport_name, display_airport_city_name_full, longitude, latitude. From the routes dataframe, the relevant features filtered from the dataframe are: tail_num, op_unique_carrier, fl_date, origin_airport_id, origin_airport_seq_id, origin_city_name, dest_airport_id, dest_airport_seq_id, dest_city_name, distance, actual_elapsed_time, distance, dep_time, arr_time. Most of these feature names are self-explanatory and easily understood as to what each column contained. This data was subsequently filtered to only contain the relevant features necessary for the paper as not to run into memory issues that could be possibly associated with large amounts of data.

To join the two dataframes, a primary key (a unique identifier for the dataframe) and a foreign key (a value to be referenced by the primary key of a parent table). The primary key for the location dataframe, the parent table in this join, was airport_seq_id, while origin_airport_seq_id and dest_sirport_seq_id were the foreign keys for the routes dataframe. These IDs were important to use as they uniquely identify each airport in the US. If done incorrectly, this join could possibly create duplicate, inaccurate rows of flight data. The airport location data was joined to the route data to create one dataframe that contained all the features necessary for each flight route.

Once the data was consolidated into a single dataframe, the size of the dataframe was further reduced to a single days' worth of data, July 18th 2018 and filtered to contain only American Airlines flights. July 18th 2018 was selected as it was in the middle of the week, and not on a holiday, as not to create an accidentally biased data set. The names of the features were also renamed to be more intuitive to the problem. Lastly, a quick query was run to ensure no duplicate rows were created in this process, which confirmed the joining and filtering process was successful.

The code for the cleaning, filtering and joining of the dataset can be seen below in Appendix 1B.

Feature Generation

Once the dataframe was created from the downloaded BTS data, 3 more features were created. A column to double check for duplicate rows named duplicates, flight_group, and num_routes. These columns were created either to confirm the overall quality of the data or as a property to easily associate the same airplane together.

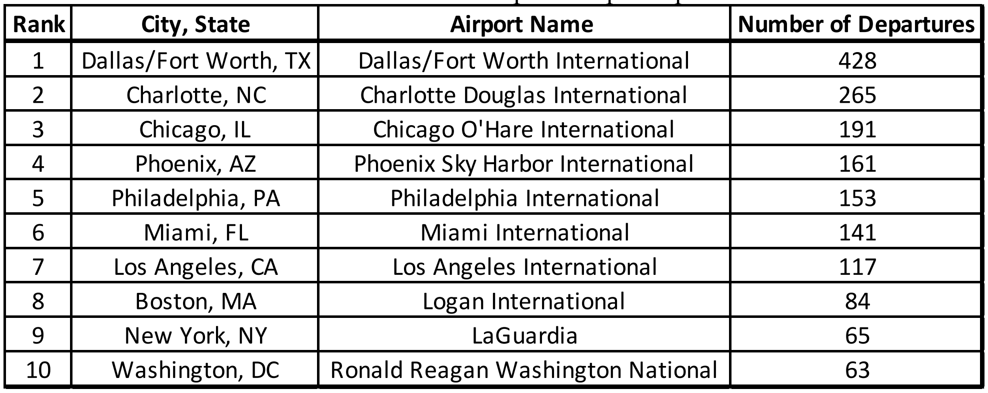
The duplicates feature was added when checking if there were any duplicates. If this column is populated with a '1' that means there are no other rows identical to that one. If that column has a value greater than '1' that means there are more than one of those rows and the join was performed incorrectly. The feature, flight_group, was created to assign a value to all of the routes one particular airplane embarked on June 18th, 2018. The feature, num_routes, was created to count the number of unique routes one particular airplane embarked on June 18th, 2018.

The code to generate these features can be found below in Appendix 1C.

Hub Identification

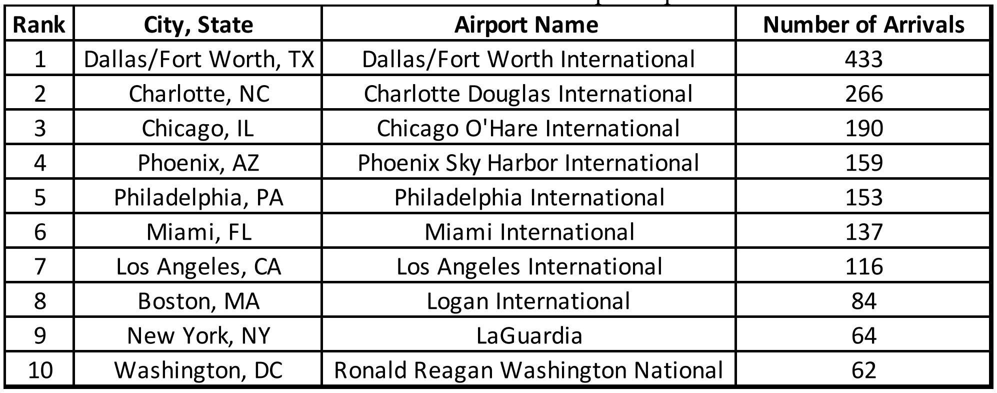
According to American Airlines, it operates through nine hubs located in the following cities: Dallas/Fort Worth, Charlotte, Washington DC, Philadelphia, New York, Chicago, Miami and Los Angeles. To double check if the data collected from BTS roughly projects the same cities and their respective airports as hubs, the top ten cities with the highest number of flights departing and arriving from them were determined and are displayed in Tables 1 and 2 below.

Table 1: Number of departures per airport



Rank	City, State	Airport Name	Number of Departures
1	Dallas/Fort Worth, TX	Dallas/Fort Worth International	428
2	Charlotte, NC	Charlotte Douglas International	265
3	Chicago, IL	Chicago O'Hare International	191
4	Phoenix, AZ	Phoenix Sky Harbor International	161
5	Philadelphia, PA	Philadelphia International	153
6	Miami, FL	Miami International	141
7	Los Angeles, CA	Los Angeles International	117
8	Boston, MA	Logan International	84
9	New York, NY	LaGuardia	65
10	Washington, DC	Ronald Reagan Washington National	63

Table 2: Number of arrivals per airport



Rank	City, State	Airport Name	Number of Arrivals
1	Dallas/Fort Worth, TX	Dallas/Fort Worth International	433
2	Charlotte, NC	Charlotte Douglas International	266
3	Chicago, IL	Chicago O'Hare International	190
4	Phoenix, AZ	Phoenix Sky Harbor International	159
5	Philadelphia, PA	Philadelphia International	153
6	Miami, FL	Miami International	137
7	Los Angeles, CA	Los Angeles International	116
8	Boston, MA	Logan International	84
9	New York, NY	LaGuardia	64
10	Washington, DC	Ronald Reagan Washington National	62

To put these numbers in perspective, roughly 62% of all air traffic for American Airlines touched at least one of these ten airports on July 18th, 2018. This percentage backs the hub-to-spoke model that American Airlines is known to use and is even higher than the estimated 40% Cook and Goodwin proposed.

One interesting point to note is that Logan International in Boston, MA ranked 8th highest in both the number of arrivals and departures, yet was not listed as a hub on American Airlines' website. This is probably due to a high demand for flights from and to Boston and may be an example of American Airlines using the P2P methodology for a special case. The other nine airports however, are hubs identified by American Airlines. Many of the departures and arrival totals are similar in number due to the rotating traffic that is received through these hubs.

This data was also the data used to visualize Figure 2 above, where the routes are most dense around the location of the hubs. For contrast, once again, Figure 3 is Southwest's linear model of operation, which creates less dense hub regions and rather focus cities. This can be seen by how many fewer "whiter routes" (less dense) Southwest Airlines has. The exact same method to creating the American Airlines data was used to create the Southwest Airline visualization.

The code to create the data for these tables can be found below in Appendix 1D.

Graph Database Design

As discussed earlier, an ensemble of tools was used to derive value from the data collected. Up to this point, the data collected was transformed from csv files into dataframes and used much like tables in a relational database would. Now, to visualize the data in a more meaningful way and to use Dijkstra's Shortest Path Algorithm for the optimization of routes, it is important to create an effect graph database.

A graph database contains data that is "organized as nodes, relationships, and properties (data stored on the nodes or relationships)." (Neo4j, *"What Is a Graph Database and Property Graph | Neo4j")* Nodes "are the entities in the graph" (Neo4j, *"What Is a Graph Database and Property Graph | Neo4j")* that can contain properties. Relationships direct "relevant connections between two node entities" and "always [have] a direction, a type, a start node, and an end node". (Neo4j, *"What Is a Graph Database and Property Graph | Neo4j")*

For the cleaned flight data, this data was modeled through a graph database using Neo4j. To do so, the cleaned dataframe named "clean.csv" was downloaded and then uploaded into Neo4j as well as the Airport Location csv data, "location.csv". Every airport was then created into a node with the following properties, id: toInteger(line.airport_seq_id), name: line.display_airport_name, city: line.display_airport_city_name_full, latitude: line.latitude, longitude: line.longitude. Every route that was mapped to a flight number in "clean.csv" was created into a node with the following properties, flight_num: line.flight_num, orig_air_id: line.orig_air_id, dest_air_id: line.dest_air_id, dep_time: line.dep_time, arr_time: line.arr_time, elapsed_time: line.elapsed_time, distance: line.distance, flight_group:line.flight_group, num_routes: line.num_routes. Once again, relationships were formed with these nodes using unique airport id and contained the distance each flight traveled. The general visualization of the database design discussed above can be seen in Figure 4. This database design was adapted from a concept Max De Marzi wrote on.

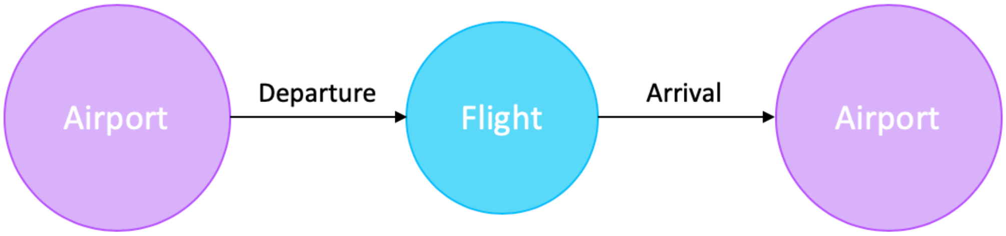


Figure 4: Graph Database Design to Visualize Flight Data

An example of the data that was cleaned earlier and then uploaded into Neo4j can be seen in Figure 5 below. Using the num_routes feature generated earlier, the flight (N1544AA) with the greatest number of routes (8) in the day was found. All of these nodes and relationships associated with this flight are in the figure as well.

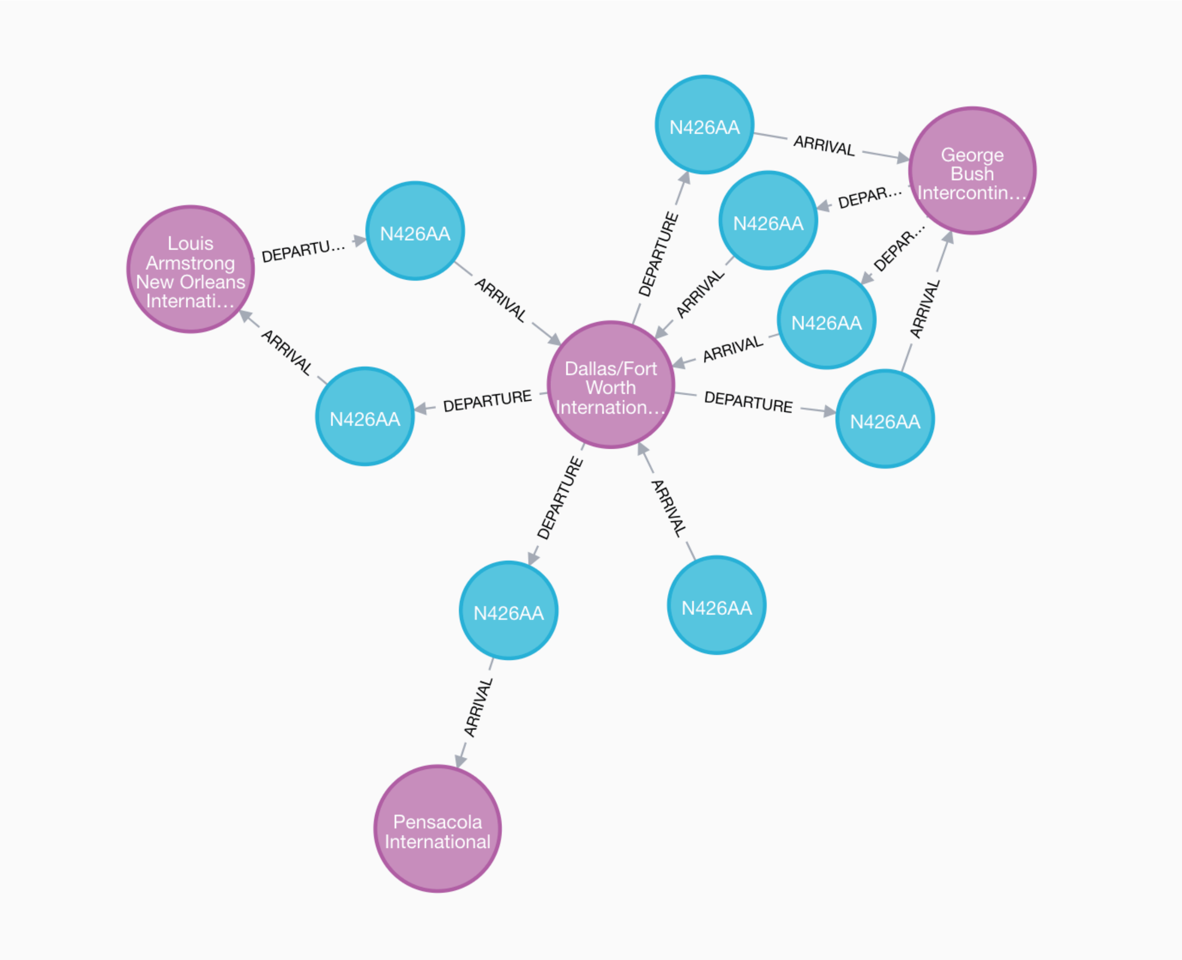


Figure 5: All nodes and relationships connected to Flight N426AA using Neo4j

The code for the creation of the graph database can be found in Appendix 2A. The code for determining all the nodes and relationships that are associated with flight N1544AA can be found in Appendix 2C.

Optimization: Dijkstra's Shortest Path Algorithm

In graph theory, Dijkstra's Shortest Path Algorithm "calculates the shortest (weighted) path between a pair of nodes". (Neo4j, *"Neo4j Graph Database Platform* According to Chen, the algorithm is based on the following three principles:

1. "All edge costs are non-negative."
2. "The number of vertices is finite."
3. "The source is a single vertex, but the target may be all other vertices."

Although Chen refers to relationships as edges, and nodes as vertices, the underlying principles remain the same. For the first principle, the graph database design contains two relationships per flight node. The first being the departure and the second being the arrival. The cost of these relationships is non-negative as they are the distance that each flight needs to travel to get from the flight's origin to its destination. The second principle is satisfied as there is only a finite number of flights that fly to a finite number of airports. The third principle is also satisfied as the source node is the origin airport and the target would just be one node, the destination airport.

Since the prerequisite principles for the algorithm are satisfied, the generic implementation of the algorithm remains the same for all graphs. This section of the paper will provide insight into the implementation of the algorithm as well as how inputs and outputs relevant to the data will fit into the algorithm. The algorithm notation and implementation discussed below is adapted from concepts introduced by both Chen and Abiy et al.

Notation

The following are important notation used in the implementation of Djikstra's algorithm:

- The Dijkstra Shortest Path Algorithm denotes all nodes as either *v* or *u*.
- A relationship is denoted as *(u, v)* and its weight is denoted as *w(u, v)*.
- The notation of the source node will be *s*.
- *N* is set of visited nodes. The set starts as empty when the algorithm starts and contains all nodes once the algorithm is finished.
- *Q* is a queue of all nodes in the graph. The queue contains all nodes when the algorithm starts and is empty once the algorithm is finished.
- *D( )* is list of distance values.
    - *D(u)* is variable that keeps track of shortest distance.
    - *D(v)* is the distance from node being examined to next node *v*. Should be initialized as ∞.
    - *D(s)* is the distance of the source node. Should be initialized as 0.

Implementation: Pseudo-Code

INPUTS: Initial source node *s*, final source node (end of queue, *Q*), weight, *w(u, v)*, is distance flight takes from one airport to another (property of relationship)
OUTPUTS: NodeID, Distance

```
WHILE len(Q) != 0:
        #if first run through loop, s will be popped because D(s) is smallest
        Pop v or s, with smallest D(v or s) from Q, if not in N
        #keeps track of v
        Add node v to N
        #update D( ) of current D(v) values based on adjacent u nodes
        IF $D(v) + w(u, v) < D(u)$:
                $D(u) = D(v) + w(u, v)$
        ELSE:
                pass
```

In Neo4j, Dijkstra's Shortest Path Algorithm is contained in a package called "Graph Algorithms". This package will allow for use of the algorithm by specifying the same inputs stated above. This package will work in a similar way to the pseudo-code implementation above and apply it to the graph database designed for this paper.

When implementing the shortest path algorithm to the problem statement, the time in which flights are run are ignored and instead distances and possible connections from one airport to another are used.

Experiment: Removal of LaGuardia (Hub) Node

Based on the initial design of the graph database, every flight has a relationship connecting to the departure and arrival airport. To determine a solution to the problem statement posed, the temporary shutdown of an airport is necessary (since all the data used in this problem is for one day, the removal of an airport node is the equivalent of the airport being shut down for a day). Considering the hub-and-spoke nature of American Airlines, shutting down a hub could be detrimental to their operations. For this experiment, the LaGuardia node will be removed to see if a possible combination of P2P with H&S may be able to benefit American Airlines.

To determining which airplanes have routes that go through LaGuardia, Python was once again used. Since there are many constraints to whether or not an airplane route can be optimized or not, this process could more accurately be determined using a Python script. It was determined that 55 unique airplanes went through LaGuardia at some point in the day. Although that many airplanes went through LaGuardia in one day, not all 55 routes can be optimized. Several factors deterred the possibility of a route being optimized, including:

1. If the start location or end location of the flight is LaGuardia, the flight will either have to be cancelled because it can't take off or will end its route prematurely.
2. If a flight only goes between LaGuardia and another airport, the flight will need to be cancelled.
3. If the flight starts at a location and then ends it route going back and forth between LaGuardia and another airport, the route will just end prematurely.

After excluding those possible scenarios, it was only possible for 16 flights routes to be optimized. And within those 16 scenarios, only part of each route needed to be optimized (the part where the plane was supposed to go to LaGuardia). The Python code used to determine the routes that could possibly optimized can be found in Appendix 1F.

Many of the 16 routes that possibly needed to be optimized didn't need to change from the original flight path. If, for example, a flight (Flight Number N110UW) was going from Charlotte Douglas International to LaGuardia, and then to Theodore Francis Green State in Rhode Island, the flight could just skip LaGuardia in its flight plan and go straight to Theodore Francis Green State as that is a service (Flight Number N732US) offered by American Airlines. This sort of scenario often occurred when the previous flight before the plane was supposed to go to LaGuardia was at hub such as Charlotte, Dallas/Fort Worth or Chicago. This accounted for 10 of the 16 routes that could be optimized through a shortest distance algorithm and become partially linear. This is because the shortest distance from one hub to another is just a direct flight from that hub to the next.

After eliminating all routes that could not be fully or partially optimized, 6 of the 55 unique flight routes could be partially converted to a linear modeled route. This means that roughly 11% of the airline routes that used LaGuardia as a hub could be modified if necessary to avoid a flight route from ending early. Using Dijkstra's Shortest Path Algorithm, flights which take place on that day serve as a blueprint for how an airplane can reroute its day to add stops, while also ending up in the final location.

The change in routes for the six flights can be seen in Appendix 3. The code to analyze the shortest possible route using the Dijkstra method can be found in Appendix 2B.

Results and Discussion

From the six flights that were able to add an additional stop to its route like a linear model, only one decreased in the total distance flow. The other five incurred an increase in the number of flights flown on the day, however, the increase in miles was usually minimal and relatively similar to the original flight. This was the result of the way in which American Airlines is already structured. American Airlines is currently set up through a H&S. When one hub went down (LaGuardia), the next best option to complete the flight route became another hub. This usually made the optimal shortest distance route to be the hub closest to where the destination airport was located, which makes sense when hubs account for over 60% of the traffic American Airlines operates through. The use of existing airline routes to optimize a route made going through a hub the next best option. This also displayed the robustness of such a network. Due to the closure of LaGuardia, the network was able to adapt by using a different airport.

Although there was usually an increase in the number of miles a flight needed to take to reach its destination, the original flight path could be salvaged with the addition of a replacement stop. This would amount to a potential increase in revenue for American Airlines as they could operate through more airports and serve more people. Since normally the route would get cancelled, adding a stop and keeping the original destination stop intact is a huge advantage and can allow them to increase passenger density. This becomes even more important if an airline would know in advance that an airport is being shut down. This would allow them to adjust other flights as well to increase their range of service.

When analyzing the results, one of the biggest points of interest was once again Logan International Airport in Boston. Since Boston is not a hub used by American Airlines, many flights that come in and out of Boston are currently due to an actual hub.

Since numerous flight fly into and out of Boston, many flights need to go through a nearby hub before it can go somewhere else. Since it is not a hub and doesn't connect to as many locations, rerouting a flight often takes going to other locations a longer process. Due to the shear capacity of flights handled in Boston, it may make sense for American Airlines to build infrastructure in that area to consider Logan International Airport a hub.

From these results, it seems as though it would be very difficult for a H&S airline to optimize their routes through a linear or a P2P approach. This is largely in part due to the way their current infrastructure is setup. Using American Airlines as an example, it may be beneficial to linearize routes when an airline closure occurs based on the situation at hand. Even changing 10% of airline routes to an airport that is experiencing a closure is a huge deal as it allows companies to salvage that may be lost and optimize their network as a whole based on those changes.

Acknowledgements

I would like to acknowledge professor Liljestrand for his advice and feedback throughout the research process. It was extremely useful and helped guide decisions made throughout the research.

Literature Cited

Abiy, Thaddeus, et al. "Dijkstra's Shortest Path Algorithm." *Brilliant Math & Science Wiki*, 2016, brilliant.org/wiki/dijkstras-short-path-finder/.

American Airlines, 2018, aa.fltmaps.com/en.

Bureau of Transportation Statistics. *Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances | Bureau of Transportation Statistics*, 28 Nov. 2018, www.bts.gov/.

Chen, Jing-Chao. "Dijkstra's Shortest Path Algorithm." *JOURNAL OF FORMALIZED MATHEMATICS*, vol. 15, 2003.

Churcher, Clare. *Beginning SQL Queries; from Novice to Professional.* Apress, 2008.

Cook, Gerald, and Jeremy Goodwin. *Airline Networks: A Comparison of Hub-AndSpoke and Point-to-Point Systems*. Journal of Aviation/Aerospace Education & Research, 2008, commons.erau.edu/cgi/viewcontent.cgi?article=1443&context=jaaer.

De Marzi, Max. "Modeling Airline Flights in Neo4j." *Graphs with Neo4j*, 26 Aug. 2015, maxdemarzi.com/2015/08/26/modeling-airline-flights-in-neo4j/.

Hock-Chuan, Chua. "A Quick-Start Tutorial on Relational Database Design." *Relational Database Design*, 2010, www.ntu.edu.sg/home/ehchua/programming/sql/relational_database_design.html.

Larcher, Hugo. "Flight Data Visualisation with Pandas and Matplotlib." *Coding Entropy*, Medium, 19 Oct. 2015, blog.hugo-larcher.com/flight-data-visualisation-with-pandas-and-matplotlib-ebbd13038647.

McDermott, John. "The Airline Economics of the Bicycle Wheel: Point-to-Point vs Hub-and-Spoke Flying." *Aeronautics Online*, 3 July 2017, aeronauticsonline.com/the-airline-economics-of-the-bicycle-wheel-point-to-point-vs-hub-and-spoke-flying/.

Neo4j. *Neo4j Graph Database Platform*, 2018, neo4j.com/docs/graph-algorithms/current/algorithms/shortest-path/.

Neo4j. "Neo4j's Graph Query Language: An Introduction to Cypher." *Neo4j Graph Database Platform*, 2018, neo4j.com/developer/cypher-query-language/#_about_cypher.

Neo4j. "What Is a Graph Database and Property Graph | Neo4j." *Neo4j Graph Database Platform*, 2018, neo4j.com/developer/graph-database/.

Python. “What Is Python? Executive Summary.” *Python*, 2018, www.python.org/doc/essays/blurb/.

Sawe, Benjamin. “The Largest Airlines in the US.” *World Atlas*, Worldatlas, 31 Jan. 2018, www.worldatlas.com/articles/the-largest-airlines-in-the-united-states.html.

Appendix 1: Python Code

Appendix 1A: Data Collection

```
#import packages needed
import numpy as np
import pandas as pd
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize, LinearSegmentedColormap, PowerNorm

#import routes.csv data into dataframe named routes
routes = pd.read_csv("routes.csv")

#import location.csv data into dataframe named location
location = pd.read_csv("location.csv")
```

Appendix 1B: Data Cleaning, Filtering, & Joining

```
#import routes.csv data into dataframe named routes
routes = pd.read_csv("routes.csv")

#import location.csv data into dataframe named location
location = pd.read_csv("location.csv")

#trim down dataframe size by selecting data needed for the research paper
location =
location[['AIRPORT_SEQ_ID','AIRPORT_ID','DISPLAY_AIRPORT_NAME','DISPLA
Y_AIRPORT_CITY_NAME_FULL','LONGITUDE','LATITUDE']]
routes =
routes[['TAIL_NUM','OP_UNIQUE_CARRIER','FL_DATE','ORIGIN_AIRPORT_ID','
ORIGIN_AIRPORT_SEQ_ID','ORIGIN_CITY_NAME',
'DEST_AIRPORT_ID','DEST_AIRPORT_SEQ_ID','DEST_CITY_NAME',
'DISTANCE', 'ACTUAL_ELAPSED_TIME', 'DISTANCE', 'DEP_TIME',
'ARR_TIME']]

#joining location data with route data to create one single dataframe with all route and
location information needed
origin_merge = pd.merge(location, routes, left_on = 'AIRPORT_SEQ_ID', right_on =
'ORIGIN_AIRPORT_SEQ_ID')
dest_merge = pd.merge(origin_merge, location, left_on = 'DEST_AIRPORT_SEQ_ID',
right_on = 'AIRPORT_SEQ_ID')

#filter data to view only data for one day: July 18th 2018 and only American Airline
flights
dest_merge = dest_merge[dest_merge['FL_DATE'].str.contains("2018-07-18")]
dest_merge = dest_merge[dest_merge['OP_UNIQUE_CARRIER'].str.contains("AA")]

#select relevant data
clean =
dest_merge[['OP_UNIQUE_CARRIER','TAIL_NUM','FL_DATE','AIRPORT_SEQ_ID_
x', 'DISPLAY_AIRPORT_NAME_x','ORIGIN_CITY_NAME','LONGITUDE_x',
'LATITUDE_x','DEST_AIRPORT_SEQ_ID','DISPLAY_AIRPORT_NAME_y','DEST_
CITY_NAME', 'LONGITUDE_y','LATITUDE_y',
'DEP_TIME','ARR_TIME','ACTUAL_ELAPSED_TIME', 'DISTANCE']]
clean = clean.reset_index()
clean = clean.drop(columns='index')
clean.columns = ['airline_id','flight_num','date','orig_air_id', 'orig_air_name', 'orig_city',
'orig_long', 'orig_lat', 'dest_air_id', 'dest_air_name', 'dest_city', 'dest_long', 'dest_lat',
'dep_time','arr_time','elapsed_time', 'distance', 'distance_2']
clean = clean.drop(columns='distance_2')

#check for duplicate data, if the number 1 is printed, there are no duplicates
clean = clean.groupby(['airline_id', 'flight_num', 'date', 'orig_air_id', 'orig_air_name',
       'orig_city', 'orig_long', 'orig_lat', 'dest_air_id', 'dest_air_name',
       'dest_city', 'dest_long', 'dest_lat', 'dep_time', 'arr_time',
       'elapsed_time', 'distance'])['flight_num'].count().reset_index(name='duplicates')
print('If the number 1 is printed, there are no duplicates:', len(clean.duplicates.unique()))
```

Appendix 1C: Feature Generation

```
#check for duplicate data
clean = clean.groupby(['airline_id', 'flight_num', 'date', 'orig_air_id', 'orig_air_name',
       'orig_city', 'orig_long', 'orig_lat', 'dest_air_id', 'dest_air_name',
       'dest_city', 'dest_long', 'dest_lat', 'dep_time', 'arr_time',
       'elapsed_time', 'distance'])['flight_num'].count().reset_index(name='duplicates')
print('If the number 1 is printed, there are no duplicates:', len(clean.duplicates.unique()))

#order each flight number by their departure times
clean.sort_values(['flight_num', 'dep_time'], ascending=[True, True], inplace=True)

#create duplicates, flight_group, num_routes
flight_groups =
clean.groupby('flight_num')['flight_num'].count().reset_index(name='num_routes')
flight_groups.insert(0, 'flight_group', range(0, 0 + len(flight_groups)))
final = pd.merge(clean, flight_groups, left_on = 'flight_num', right_on = 'flight_num')
clean = final

# convert clean (final dataframe created) to a csv
clean.to_csv('clean.csv')
```

Appendix 1D: Hub Identification

```
#Top 10 airport departures
hub = clean.groupby(['orig_city','orig_air_name'])['orig_air_name'].count().reset_index(name='nb_flights')
hub = hub.sort_values(by='nb_flights',ascending = False).head(10)
hub.to_csv('hub_departures.csv')
hub.head(10)

#Top 10 airport arrivals
hub = clean.groupby(['dest_city','dest_air_name'])['dest_city'].count().reset_index(name='nb_flights')
hub = hub.sort_values(by='nb_flights',ascending = False).head(10)
hub.to_csv('hub_arrivals.csv')
hub.head(10)
```

Appendix 1E: Route Optimization

```
# Code Citation
""" Title: Flight data visualisation with Pandas and Matplotlib
Author: Hugo Larcher
Date: 10/19/15
Availability: https://blog.hugo-larcher.com/flight-data-visualisation-with-pandas-and-matplotlib-ebbd13038647

Larcher, Hugo. “Flight Data Visualisation with Pandas and Matplotlib.” Coding Entropy, Medium, 19 Oct. 2015, blog.hugo-larcher.com/flight-data-visualisation-with-pandas-and-matplotlib-ebbd13038647.
"""

# Visualize airline route data
def plot_map(in_filename, color_mode='screen',
             out_filename='airline_viz.png', absolute=False):
    """Plots the given CSV data files use matplotlib basemap and saves it to
    a PNG file.
    Args:
        in_filename: Filename of the CSV containing the data points.
        out_filename: Output image filename
        color_mode: Use 'screen' if you intend to use the visualisation for
                    on screen display. Use 'print' to save the visualisation
                    with printer-friendly colors.
        absolute: set to True if you want coloring to depend on your dataset
                  parameter value (ie for comparison).
                  When set to false, each coordinate pair gets a different
                  color.
    """

    if color_mode == 'screen':
        bg_color = (0.0, 0.0, 0, 1.0)
        coast_color = (204/255.0, 0, 153/255.0, 0.7)
        color_list = [(0.0, 0.0, 0.0, 0.0),
                      (204/255.0, 0, 153/255.0, 0.6),
                      (255/255.0, 204/255.0, 230/255.0, 1.0)]
    else:
        bg_color = (1.0, 1.0, 1.0, 1.0)
        coast_color = (10.0/255.0, 10.0/255.0, 10/255.0, 0.8)
        color_list = [(1.0, 1.0, 1.0, 0.0),
                      (255/255.0, 204/255.0, 230/255.0, 1.0),
                      (204/255.0, 0, 153/255.0, 0.6)
                      ]

    # define the expected CSV columns -- THIS IS THE ADAPTED PART OF THE
CODE TO WORK WITH RESEARCH DATA
    routes = clean[['orig_lat', 'orig_long', 'dest_lat','dest_long']]
    routes = routes.groupby(['orig_lat', 'orig_long',
'dest_lat','dest_long'])['dest_long'].count().reset_index(name='nb_flights')
    routes.insert(5, 'CO2', 1)
    routes.columns = ['dep_lat', 'dep_lon', 'arr_lat', 'arr_lon','nb_flights', 'CO2']

    num_routes = len(routes.index)

    # normalize the dataset for color scale
    norm = PowerNorm(0.3, routes['nb_flights'].min(),
                     routes['nb_flights'].max())
    # norm = Normalize(routes['nb_flights'].min(), routes['nb_flights'].max())

    # create a linear color scale with enough colors
    if absolute:
        n = routes['nb_flights'].max()
    else:
        n = num_routes
    cmap = LinearSegmentedColormap.from_list('cmap_flights', color_list,
                                             N=n)
    # create the map and draw country boundaries
    plt.figure(figsize=(27, 20))
    m = Basemap(projection='mill', lon_0=0)
    m.drawcoastlines(color=coast_color, linewidth=1.0)
    m.fillcontinents(color=bg_color, lake_color=bg_color)
    m.drawmapboundary(fill_color=bg_color)

    # plot each route with its color depending on the number of flights
    for i, route in enumerate(routes.sort_values(by='nb_flights',
                              ascending=True).iterrows()):
        route = route[1]
        if absolute:
            color = cmap(norm(int(route['nb_flights'])))
        else:
            color = cmap(i * 1.0 / num_routes)

        line, = m.drawgreatcircle(route['dep_lon'], route['dep_lat'],
                                  route['arr_lon'], route['arr_lat'],
                                  linewidth=0.5, color=color)
        # if the path wraps the image, basemap plots a nasty line connecting
        # the points at the opposite border of the map.
        # we thus detect path that are bigger than 30km and split them
        # by adding a NaN
        path = line.get_path()
        cut_point, = np.where(np.abs(np.diff(path.vertices[:, 0])) > 30000e3)
        if len(cut_point) > 0:
            cut_point = cut_point[0]
            vertices = np.concatenate([path.vertices[:cut_point, :],
                                      [[np.nan, np.nan]],
                                      path.vertices[cut_point+1:, :]])
            path.codes = None  # treat vertices as a serie of line segments
            path.vertices = vertices

    # save the map
    plt.savefig(out_filename, format='png', bbox_inches='tight')

if __name__ == '__main__':
    # use 'screen' color mode for on-screen display. Use 'print' if you intend
    # to print the map
    plot_map('data.csv', 'screen', absolute=False)
```

Appendix 1F: Flight Determination

```
# Determine all flights departing from LaGuardia
laguardia_flights = clean[clean['orig_air_name'].str.contains("LaGuardia")]

# Determine all unique flights departing from LaGuardia
laguardia_flight_num = laguardia_flights['flight_num'].tolist()

# Determine routes of all unique flights taking off from LaGuardia
laguardia_flights = clean[clean['flight_num'].isin(laguardia_flight_num)]
laguardia_flights

#Determine number of unique LaGuardia flights
```

```
laguardia_flights['flight_num'].nunique()

#keep track of which flights can or cannot be optimized
cancelled_flights = []
same_flights = []
other_flights = []
optimization = []

#eliminate flights that can not be further optimized
for i in range (0, laguardia_flights['flight_num'].nunique()):
    ignore = False
    #Determine which flights would or wouldn't be cancelled
    current_flight = laguardia_flight_num[i]
    lg_current_flight = clean[clean['flight_num'].str.contains(current_flight)]
    start_location = lg_current_flight['orig_air_name'].iloc[0]
    end = len(lg_current_flight) - 1
    end_location = lg_current_flight['dest_air_name'].iloc[end]
    if start_location == 'LaGuardia' or end_location == 'LaGuardia':
        cancelled_flights.append(current_flight)
        ignore = True
    elif len(lg_current_flight) == 2:
        cancelled_flights.append(current_flight)
        ignore = True
    elif lg_current_flight['dest_air_name'].iloc[end-1] == 'LaGuardia':
        same_flights.append(current_flight)
        ignore = True
    else:
        pass

    #Determine start of route optimization
    if ignore == False:
        for j in range(0, len(lg_current_flight)):
            num = j
            if lg_current_flight['dest_air_name'].iloc[num] == 'LaGuardia':
                start_o = lg_current_flight['orig_air_name'].iloc[num]
                start_fn = lg_current_flight['flight_num'].iloc[num]
                start_time_o = lg_current_flight['dep_time'].iloc[num]
                end_num = num+2
                end_o = lg_current_flight['dest_air_name'].iloc[end_num]
                end_time_o = lg_current_flight['dep_time'].iloc[end_num]
                break
        if start_o == end_o:
            same_flights.append(current_flight)
        else:
            other_flights.append(current_flight)
            optimize = [start_o, start_fn, start_time_o, end_o, end_time_o]
        optimization.append(optimize)

print('Total flights usually associated with LaGuardia:',
laguardia_flights['flight_num'].nunique())
print('Number of flights to be cancelled:', len(cancelled_flights))
print('Number of flights that will remain the same:', len(other_flights))
print('Number of flights possible to be optimized:', len(optimization))
print('')

#print part of route that needs to be optimized for a flight
print(optimization)
```

Appendix 2: Cypher (Neo4j) Code

Appendix 2A: Graph Database Design

```
//Setting up graph data

//Create location nodes

LOAD CSV WITH HEADERS FROM 'file:///location.csv' AS line
CREATE (a:Airport { id: toInteger(line.AIRPORT_SEQ_ID), name:
line.DISPLAY_AIRPORT_NAME, city:
line.DISPLAY_AIRPORT_CITY_NAME_FULL, latitude: line.LATITUDE, longitude:
line.LONGITUDE})

//Create flight nodes

LOAD CSV WITH HEADERS FROM 'file:///clean.csv' AS line
CREATE (f:Flight { flight_num: line.flight_num, orig_air_id: line.orig_air_id,
dest_air_id: line.dest_air_id, dep_time: line.dep_time, arr_time: line.arr_time,
elapsed_time: line.elapsed_time, distance: line.distance, flight_group:line.flight_group,
num_routes: line.num_routes})

//Create departure-arrival relationship
MATCH (f:Flight), (one:Airport), (two:Airport)
WHERE toInteger(one.id) = toInteger(f.orig_air_id) AND toInteger(two.id) =
toInteger(f.dest_air_id)
CREATE (one)-[d:DEPARTURE { toInteger(f.distance)/2}]->(f)-
[a:ARRIVAL{distance: toInteger(f.distance)/2}]->(two)
RETURN d,a;
```

Appendix 2B: Experiment: Removal of LaGuardia (Hub) Node

```
//Creating the experiment

//Delete LaGuardia Node
MATCH (a:Airport { name: 'LaGuardia' })
DETACH DELETE a

// Possible Route Optimization Scripts (Only 6 flights had no direct flight to solve the
issue)

//Route 1: Flight N110UW @ Time: 1307
MATCH (start:Airport{name:'Charlotte Douglas International'}),
(end:Airport{name:'Theodore Francis Green State'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
```

```
RETURN nodeId AS name, cost

//Route 2: Flight N154AA @ Time: 1138
MATCH (start:Airport{name:'Los Angeles International'}), (end:Airport{name:'Logan
International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 3: Flight N567UW@ Time: 1138
MATCH (start:Airport{name:'Charlotte Douglas International'}),
(end:Airport{name:'McCarran International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 4: Flight N572UW @ Time: 1435
MATCH (start:Airport{name:'Charlotte Douglas International'}),
(end:Airport{name:'Kansas City International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 5: Flight N754UW @ Time: 558
MATCH (start:Airport{name:'Charlotte Douglas International'}),
(end:Airport{name:'Pittsburgh International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 6: Flight N812NN @ Time: 931
MATCH (start:Airport{name:"Chicago O'Hare International"}), (end:Airport{name:'San
Diego International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 7 & 8: Flight N905AU @ Time: 723, Flight N909AM @ Time: 1220
MATCH (start:Airport{name:"Dallas/Fort Worth International"}),
(end:Airport{name:'San Diego International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 9: Flight N909AN @ Time: 1236
```

```
MATCH (start:Airport{name:"Chicago O'Hare International"}),
(end:Airport{name:'Miami International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 10: Flight N920AN @ Time: 556
MATCH (start:Airport{name:"Chicago O'Hare International"}),
(end:Airport{name:'George Bush Intercontinental/Houston'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 11: Flight N926AN @ Time: 826
MATCH (start:Airport{name:"Chicago O'Hare International"}),
(end:Airport{name:'Ronald Reagan Washington National'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 12: Flight N935NN @ Time: 1342
MATCH (start:Airport{name:"Chicago O'Hare International"}), (end:Airport{name:'San Antonio International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 13, 15 & 16: Flight N945UW @ Time: 953, Flight N955UW @ Time: 953, Flight N960UW @ Time: 1355
MATCH (start:Airport{name:"Logan International"}), (end:Airport{name:'Ronald Reagan Washington National'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

//Route 14: Flight N949UW @ Time: 554
MATCH (start:Airport{name:"Logan International"}), (end:Airport{name:'Philadelphia International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost
```

Appendix 2C: General Helpful Queries

```
MATCH (f:Flight { flight_num: 'N110UW', dep_time: '1307.0'})-[r]-()
RETURN f;
```

```
MATCH (s)
WHERE ID(s) = 7666
RETURN s

MATCH (start:Flight{flight_num:'N154AA', dep_time: '1138.0'}),
(end:Airport{name:'Los Angeles International'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
RETURN nodeId AS name, cost

MATCH (f:Flight { flight_num: 'N154AA' })-[r]->(b)
RETURN *;

//Delete Relationships
MATCH ()-[d:DEPARTURE]-()
DELETE d
```

Appendix 3: Optimized Linear Route Results

Table 3: Optimized Linear Route for Flight N154AA

Flight Number	City_OR	Original Route (OR)	OR_Distance (miles)	City_LOR	Linear Optimized Route (LOR)	LOR_Distance (miles)
N154AA	Chicago	Chicago O'Hare International	-	Chicago	Chicago O'Hare International	-
N154AA	Dallas	Dallas/Fort Worth International	802	Dallas	Dallas/Fort Worth International	802
N154AA	Los Angeles	Los Angeles International	1235	Los Angeles	Los Angeles International	1235
N154AA	~~New York~~	~~LaGuardia~~	2475	Philadelphia	Philadelphia International	2402
N154AA	Boston	Logan International	184	Boston	Logan International	280
TOTAL DISTANCE:			4696	TOTAL DISTANCE:		4719

Table 4: Optimized Linear Route for Flight N567UW

Flight Number	City_OR	Original Route (OR)	OR_Distance (miles)	City_LOR	Linear Optimized Route (LOR)	LOR_Distance (miles)
N567UW	Tampa Bay	Tampa International	-	Tampa Bay	Tampa International	-
N567UW	Charlotte	Charlotte Douglas International	507	Charlotte	Charlotte Douglas International	507
N567UW	~~New York~~	~~LaGuardia~~	544	Philadelphia	Philadelphia International	449
N567UW	Las Vegas	McCarran International	1389	Las Vegas	McCarran International	2176
TOTAL DISTANCE:			2440	TOTAL DISTANCE:		3132

Table 5: Optimized Linear Route for Flight N572UW

Flight Number	City_OR	Original Route (OR)	OR_Distance (miles)	City_LOR	Linear Optimized Route (LOR)	LOR_Distance (miles)
N572UW	Philadelphi	Philadelphia International	-	Philadelphi	Philadelphia International	-
N572UW	Chicago	Chicago O'Hare International	678	Chicago	Chicago O'Hare International	678
N572UW	Charlotte	Charlotte Douglas International	599	Charlotte	Charlotte Douglas International	599
N572UW	New York	~~LaGuardia~~	544	New York	Chicago O'Hare International	678
N572UW	Kansas City	Kansas City International	1112	Kansas City	Kansas City International	403
TOTAL DISTANCE:			2933	TOTAL DISTANCE:		2358

Table 6: Optimized Linear Route for Flight N945UW

Flight Number	City_OR	Original Route (OR)	OR_Distance (miles)	City_LOR	Linear Optimized Route (LOR)	LOR_Distance (miles)
N945UW	Manchester	Manchester-Boston Regional	-	Manchester	Manchester-Boston Regional	-
N945UW	Philadelphia	Philadelphia International	289	Philadelphia	Philadelphia International	289
N945UW	Boston	Logan international	280	Boston	Logan international	280
N945UW	~~New York~~	~~LaGuardia~~	184	Philadelphia	Philadelphia International	280
N945UW	Washington DC	Ronald Reagan Washington National	198	Washington DC	Ronald Reagan Washington Nati	399
TOTAL DISTANCE:			951	TOTAL DISTANCE:		1248

Table 7: Optimized Linear Route for Flight N955UW

Flight Number	City_OR	Original Route (OR)	OR_Distance (miles)	City_LOR	Linear Optimized Route (LOR)	LOR_Distance (miles)
N955UW	Manchester	Manchester-Boston Regional	-	Manchester	Manchester-Boston Regional	-
N955UW	Philadelphia	Philadelphia International	289	Philadelphia	Philadelphia International	289
N955UW	Boston	Logan international	280	Boston	Logan international	280
N955UW	~~New York~~	~~LaGuardia~~	184	Philadelphia	Philadelphia International	280
N955UW	Washington DC	Ronald Reagan Washington National	198	Washington DC	Ronald Reagan Washington Nati	399
TOTAL DISTANCE:			951	TOTAL DISTANCE:		1248

Table 8: Optimized Linear Route for Flight N960UW

Flight Number	City_OR	Original Route (OR)	OR_Distance (miles)	City_LOR	Linear Optimized Route (LOR)	LOR_Distance (miles)
N960UW	Boston	Logan international	-	Boston	Logan international	-
N960UW	Philadelphia	Philadelphia International	280	Philadelphia	Philadelphia International	280
N960UW	Boston	Logan international	280	Boston	Logan international	280
N960UW	~~New York~~	~~LaGuardia~~	184	Philadelphia	Philadelphia International	280
N960UW	Washington DC	Ronald Reagan Washington Nati	198	Washington	Ronald Reagan Washington Nati	399
TOTAL DISTANCE:			942	TOTAL DISTANCE:		1239