Routing Optimisation for Aeronautical Networks

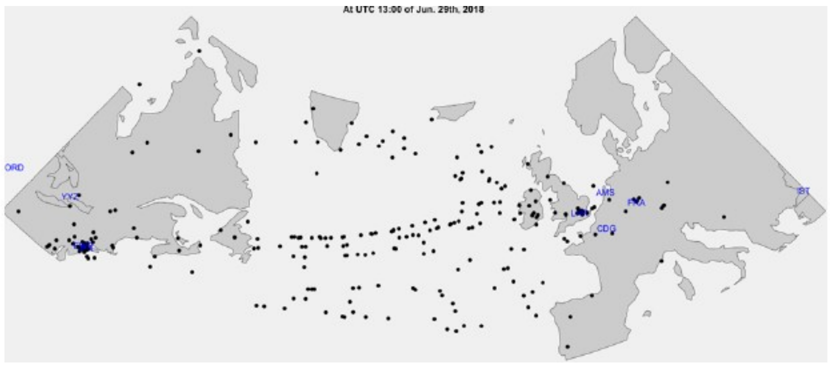
Introduction

Wireless communication and internet access have become essential in human daily life. These networks rely on 4G or 5G wireless mobile systems. However, accessing the internet in the skies is not currently possible. This paper proposes a solution to the North-Atlantic aeroplane dataset, which includes aircraft details and a data transmission table, to provide internet access to passengers on board.

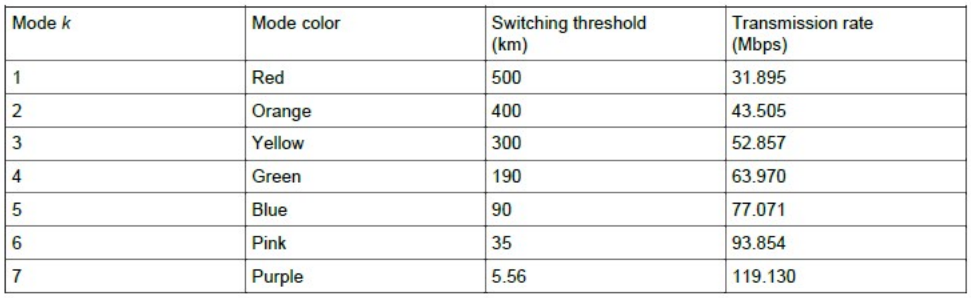
The efficient routing of air traffic is of critical importance for the safe and efficient operation of aeronautical networks. In this paper, I investigate various routing optimization techniques for aeronautical networks. These techniques are designed to improve the overall performance of the network by reducing delays, increasing transmission rate, and improving connectivity.I present 3 types of algorithm that are DIJKSTRA'S ALGORITHM and ANT COLONY OPTIMISATION (ACO) and evaluate their performance using real-world data. The results demonstrate the potential for significant improvements in the efficiency and connectivity of aeronautical networks through the use of advanced routing optimization techniques.

Problem Statement

Numerous planes travel across the North Atlantic daily, as depicted in the image where each black dot symbolizes an aircraft. To furnish internet access to individuals on board, each plane must determine the most efficient path for transmitting data packets to a ground station (GS) based on one or multiple objectives that require optimization.



To identify the optimal path for routing data packets, two key metrics must be considered: the overall speed at which data is transmitted and the total delay incurred by each link in the path. The total delay, known as end-to-end latency, is calculated by summing up the delay caused by each link in the path. For example, in a routing path that goes from Airplane-5 to Airplane-3 and then to GS-1, with each link having a delay of 50 milliseconds, the end-to-end latency would be 100 milliseconds. The end-to-end data transmission rate, on the other hand, is determined by the slowest link in the routing path. For example, in a routing path that goes from Airplane-5 to Airplane-3 and then to GS-1, with a data transmission rate of 52.857 Mbps between Airplane-5 and Airplane-3 and 43.505 Mbps between Airplane-3 and GS-1, the end-to-end data transmission rate would be 43.505 Mbps. The data transmission rate of a link is determined by the distance between the communicating airplanes, as outlined in Table 1. As an example, if the distance between Airplane-5 and Airplane-3 is 350 km, falling in the range of 300 km to 400 km, the data transmission rate of the link between these two airplanes would be 52.875 Mbps.



Mode *k*	Mode color	Switching threshold (km)	Transmission rate (Mbps)
1	Red	500	31.895
2	Orange	400	43.505
3	Yellow	300	52.857
4	Green	190	63.970
5	Blue	90	77.071
6	Pink	35	93.854
7	Purple	5.56	119.130

OPTIMISATION PROBLEMS

In this assesment I have addressed two type of problems for optimizing routing path:

1. Single-objective optimisation:

Routing path having the maximum end-to-end data transmission rate for each airplane that can access any of a GS, either at Heathrow airport (LHR) (Latitude, Longitude, Altitude) = (51.4700° N, 0.4543° W, 81.73 feet) or Newark Liberty International Airport (EWR) (Latitude, Longitude, Altitude) =(40.6895° N, 74.1745° W, 8.72 feet).

2. Multiple-objective optimisation:

Routing path having the maximum end-to-end data transmission rate and minimum end-to-end latency for each airplane that can access any of a GS, either at Heathrow airport (Latitude,Longitude, Altitude) = (51.4700° N, 0.4543° W, 81.73 feet) or Newark Liberty International Airport (Latitude,Longitude, Altitude) = (40.6895° N, 74.1745° W, 8.72 feet).

Literatiure Review

Single objective optimization (SOO) is the process of finding the optimal solution to a problem with a single objective function that needs to be minimized or maximized. This approach is widely used in various fields, such as engineering, finance, and operations research. There are many different algorithms used for SOO (Griffel, 2003), including gradient descent, Newton's method, and evolutionary algorithms such as genetic algorithms. These methods have been shown to be effective in solving a wide range of optimization problems, but they can become impractical or even impossible to use when the problem becomes too complex or has multiple objectives. Multi-objective optimization (MOO) is the process of finding the set of non-dominated solutions that optimally trade-off between multiple objectives. These objectives can be conflicting, meaning it is not possible to find a solution that optimizes all of them at the same time. The approach commonly used to solve MOO is the use of metaheuristics such as evolutionary algorithms (Deb and Jain, 2014) and swarm intelligence algorithms (Coello, Pulido and Lechuga, 2004), and in many cases, combination of both. Recent research has focused on developing hybrid algorithms that combine both SOO and MOO techniques(S Kalyankar and R. V. Dukkipati, 2013), in order to combine the strengths of both approaches. One example of this is the use of evolutionary algorithms to solve multi-objective optimization problems (Zong Woo Geem, Joong Hoon Kim and Loganathan, 2001) by incorporating a scalarization method to convert the multi-objective problem into a single objective problem that can be solved using SOO techniques.

Aeronautical networks (Zhang et al., 2022)are complex systems that are responsible for providing communication, navigation, and surveillance (CNS) services to aircrafts. Routing in these networks is a challenging task due to the dynamic and uncertain nature of the airspace, which is affected by various factors such as weather, traffic, and topography. There are several routing optimization techniques (R. McEliece and J. K. O'Gorman, 2018) that have been proposed for aeronautical networks, including mathematical optimization, heuristic algorithms, and metaheuristics.

Mathematical optimization techniques such as linear programming, mixed-integer programming, and non-linear programming have been used to solve routing problems in aeronautical networks. These techniques have been shown to be effective in finding optimal solutions, but they can become impractical or even impossible to use when the problem becomes too complex or has many constraints.

Heuristic algorithms such as greedy algorithms, simulated annealing, and tabu search have also been proposed for routing optimization in aeronautical networks. These algorithms have the advantage of being relatively simple to implement and can find good solutions quickly, but they can become trapped in local optima(Z. Zou, L. Hanzo and Z. Yang, 2019).

Metaheuristics, such as genetic algorithms (Bhardwaj and El-Ocla, 2020), ant colony optimization, and particle swarm optimization have been proposed as a solution for routing optimization in aeronautical networks. These algorithms are inspired by natural phenomena and have the ability to explore the solution space, making them well suited to solving complex optimization problems with multiple objectives.

Recently, some studies have tried to combine different approaches such as mathematical optimization and metaheuristics to solve routing problems in aeronautical networks.Some recent research in this field also focus on the integration of AI techniques such as Machine learning, Reinforcement learning for dynamic optimization of the routing problem (Liu et al., 2021).

Methodology

1. Data Preprocessing: The first step is to preprocess the data. This includes cleaning, normalizing, and formatting the data to ensure that it can be easily analyzed. Specifically, the data should include information about the location of aircrafts, ground stations and any other relevant information that is needed for the routing optimization.
2. Distance and Transmission Function: Next, define the distance function and transmission function. The distance function is used to calculate the distance between two nodes, and the transmission function is used to calculate the transmission rate between two nodes. The transmission rate is calculated based on the distance between the nodes, the altitude of the aircraft and the type of antenna used at the ground station.
3. Create Dictionary: Create a dictionary with every possible node. This will be used to store the information about each node, including its location, transmission rate and other relevant information.
4. Divide Planes: Divide the aircrafts into two lists, one for aircrafts that are in flight and another for aircrafts that are on the ground. This will allow us to apply the routing algorithm to both lists of aircrafts.
5. Routing Algorithm: Apply the routing algorithm such as Dijkstra or Bellman-Ford algorithm, to both lists of aircrafts and combine the routing path for a single optimization problem. The algorithm should consider the transmission rate, distance, and latency as the parameters to optimize the route.
6. Multiple Optimization: For multiple optimization problems, use the latency and add 50ms latency for each node in the routing path we got in the single optimization. Also, consider the transmission rate, distance and other relevant parameters that affect the routing optimization.
7. Ant Colony Optimization: Use the Ant Colony Optimization algorithm for multiple routing optimization and print the best route with the maximum transmission rate and minimum latency.
8. Evaluation: Finally, evaluate the results by comparing the performance of the proposed methodology with existing methods and analyze the results. This can include metrics such as transmission rate, latency, and routing path length.

In [1]:
```python
# Importing Libariries

from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Data Preprocessing

DATASET

The dataset for this assignment consists of one file:

NA_13_Jun_29_2018_UTC13.csv

In [2]:
```
# Importing dataset

flight_data = pd.read_csv("NA_13_Jun_29_2018_UTC13.CSV") # load csv file
flight_data.tail(10)
```

Out[2]:

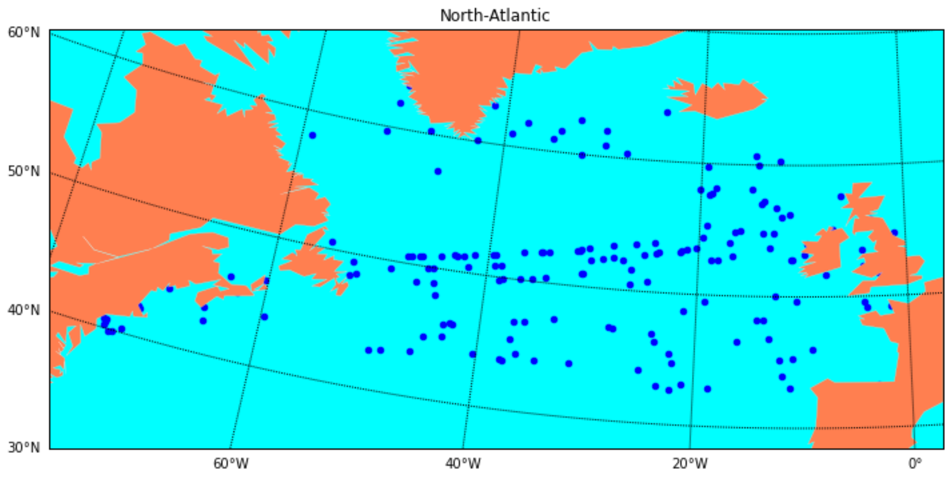
	Flight No.	Timestamp	Altitude	Latitude	Longitude
206	UA951	1530277200	35000.0	52.4	-28.1
207	UA953	1530277200	32000.0	59.8	-19.4
208	UA957	1530277200	34000.0	43.0	-49.8
209	UA961	1530277200	34000.0	52.3	-27.2
210	UA963	1530277200	32000.0	50.7	-46.5
211	UA971	1530277200	32000.0	60.9	-29.9
212	UA973	1530277200	33000.0	61.0	-39.3
213	UA975	1530277200	36000.0	50.5	-26.4
214	UA986	1530277200	36000.0	60.0	-32.2
215	UA988	1530277200	36100.0	52.7	-18.8

In [3]:
```
#renaming column name
flight_data.rename(columns = {'Flight No.':'Flight_No'}, inplace = True)
```

In [4]:
```
#Coverting column values in array

LatitudeData = flight_data.Latitude.copy()
LongitudeData = flight_data.Longitude.copy()
AltitudeData = flight_data.Altitude.copy()
flightnumber = flight_data.Flight_No.copy()
```

Visualising all the airplane location for the given timestamp

In [5]:
```
min_lat = 30
max_lat = 70
min_lon = -75
max_lon = 5
fig = plt.figure(figsize=(12, 9))
ax = plt.axes()
m = Basemap(projection='eqdc',  # Equidistant Conic Projection
            llcrnrlat=min_lat,
            urcrnrlat=max_lat,
            llcrnrlon=min_lon,
            urcrnrlon=max_lon,
            lat_1=10,lat_2=20,lat_0=50,lon_0=-10
           )

m.drawcoastlines(color='white', linewidth=0.2)  # add coastlines
m.shadedrelief(scale=0.5)
m.drawparallels(np.arange(-90, 90, 10), labels=[1, 0, 0, 0], zorder=1)
m.drawmeridians(np.arange(-180, 180, 20), labels=[0, 0, 0, 1], zorder=2)

m.drawmapboundary(fill_color='aqua') # change the background color of the map
m.fillcontinents(color='coral',lake_color='aqua') # change the color of the continents

# Set airport locations
LHR = [51.47, -0.45, 24.91]   # (Latitude, Longitude, Altitude)
EWR = [40.69, -74.17, 2.66]   # (Latitude, Longitude, Altitude)
latitudes = [LHR[0],EWR[0]]
longitudes = [LHR[1],EWR[1]]
altitudes = [LHR[2],EWR[2]]
lats_with_head = LatitudeData
lons_with_head = LongitudeData

lats = LatitudeData
lons = LongitudeData
# compute n# make up some data for scatter plot
x, y = m(lons, lats)  # transform coordinates
plt.scatter(x, y, 20, marker='o', color='Blue')
#text_airport_LHR = ax.text(0, 0, '', c='b', zorder=5, bbox=dict(facecolor='w', alpha=0.5, edgecolor='w'))
#text_airport_EWR = ax.text(0, 0, '', c='b', zorder=5, bbox=dict(facecolor='w', alpha=0.5, edgecolor='w'))
#text_airport_LHR.set_position([x[0]*1.01, y[0]*1.01])
#text_airport_EWR.set_position([x[1]*1.02, y[1]*1.02])
#text_airport_LHR.set_text('LHR')
#text_airport_EWR.set_text('EWR')

plt.title("North-Atlantic")
plt.show()
```



In [6]:
```
# Addding LHR data in fligh data

new_row = pd.DataFrame({'Flight_No':'LHR', 'Timestamp':1530277200, 'Altitude':24.91, 'Latitude':51.47, 'Longitude': -0.45}, inde
flight_data = pd.concat([new_row,flight_data.loc[:]]).reset_index(drop=True)


# EWR data in flight data
new_row = {'Flight_No':'EWR', 'Timestamp':1530277200, 'Altitude':2.66, 'Latitude':40.69, 'Longitude': -74.17}
flight_data = flight_data.append(new_row, ignore_index=True)
print(flight_data)
```

```
    Flight_No   Timestamp  Altitude  Latitude  Longitude
0         LHR  1530277200     24.91     51.47      -0.45
1       AA101  1530277200  39000.00     50.90     -38.70
2       AA109  1530277200  33000.00     60.30     -12.20
3       AA111  1530277200  39000.00     52.70     -18.10
4       AA113  1530277200  37000.00     43.00     -11.10
..        ...         ...       ...       ...        ...
213     UA973  1530277200  33000.00     61.00     -39.30
214     UA975  1530277200  36000.00     50.50     -26.40
215     UA986  1530277200  36000.00     60.00     -32.20
216     UA988  1530277200  36100.00     52.70     -18.80
217       EWR  1530277200      2.66     40.69     -74.17

[218 rows x 5 columns]
```

In [7]:
```
#Priniting in arrays for future reference

LatitudeData = flight_data.Latitude.copy()
LongitudeData = flight_data.Longitude.copy()
AltitudeData = flight_data.Altitude.copy()
flightnumber = flight_data.Flight_No.copy()
print(flightnumber)
print(LatitudeData)
print(LongitudeData)
print(AltitudeData)
```

```
         ...
213   -39.30
214   -26.40
215   -32.20
216   -18.80
217   -74.17
Name: Longitude, Length: 218, dtype: float64
0         24.91
1      39000.00
2      33000.00
3      39000.00
4      37000.00
         ...
213    33000.00
214    36000.00
215    36000.00
216    36100.00
217        2.66
Name: Altitude, Length: 218, dtype: float64
```

In [8]:
```
#Printing list of airplanes

airplane1 = list(flight_data['Flight_No'].values)
airplane=airplane1[1:217]
```

In [9]:
```
# distanve function that takes index values og flighdata

import math
def distance1(a,b):

    R_E = 6371000   # The radius of earth
    L_a = AltitudeData[a] * 0.3048  # get altitude and convert foot to meter
    L_b = AltitudeData[b] * 0.3048  # get altitude and convert foot to meter

    Theta_a = LatitudeData[a] # get latitude
    Theta_b = LatitudeData[b] # get latitude

    Varphi_a = LongitudeData[a] # get longitude
    Varphi_b = LongitudeData[b] # get longitude


    # The below code is to convert (altitude, latitude, longitude) to 3D Cartesian coordinates
    p_xa = (R_E + L_a) * math.cos(math.radians(Theta_a)) * math.cos(math.radians(Varphi_a))  # Eq. (15)
    p_ya = (R_E + L_a) * math.cos(math.radians(Theta_a)) * math.sin(math.radians(Varphi_a))  # Eq. (16)
    p_za = (R_E + L_a) * math.sin(math.radians(Theta_a))  # Eq. (17)

    p_xb = (R_E + L_b) * math.cos(math.radians(Theta_b)) * math.cos(math.radians(Varphi_b))  # Eq. (15)
    p_yb = (R_E + L_b) * math.cos(math.radians(Theta_b)) * math.sin(math.radians(Varphi_b))  # Eq. (16)
    p_zb = (R_E + L_b) * math.sin(math.radians(Theta_b))  # Eq. (17)

    # calculate the distance between aircraft a and aircraft b
    d_ab_in_m = math.sqrt((abs(p_xa - p_xb)) ** 2 + (abs(p_ya - p_yb)) ** 2 + (abs(p_za - p_zb)) ** 2)
    d_ab_in_km = d_ab_in_m
    return d_ab_in_m
```

In [10]:
```
# Mapping index to airplane names

N = [i for i in range(0, 218)]
V =  N
A = [(i, j) for i in V for j in V]
c = {(i, j): distance1(i,j) for i, j in A}
imapping = flight_data.to_dict()['Flight_No']
index_mapping = {v: k for k, v in imapping.items()}

# Reverse mapping from index values to airplanes
reverse_mapping = {index: airplane for airplane, index in index_mapping.items()}

# Dictionary with index values as keys
d_indexed = c

# Convert keys to airplane names using reverse mapping
d = {(reverse_mapping[k1], reverse_mapping[k2]): v for (k1, k2), v in d_indexed.items()}
```

In [11]:
```
## Final dictionary of nodes and their distance as values
try_dict = {key: value for key, value in d.items() if key[0] != key[1]}
try_dict
```

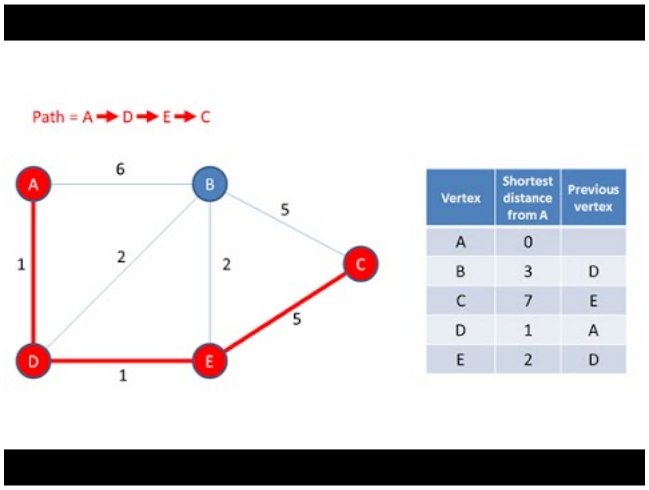
```
 ('LHR', 'BA295'): 2455120.0310311327,
 ('LHR', 'BA49'): 23983.53557319928,
 ('LHR', 'BA53'): 3623154.1035190546,
 ('LHR', 'BA67'): 741295.6669974615,
 ('LHR', 'BA801'): 785190.9114970781,
 ('LHR', 'BA93'): 158478.94550300032,
 ('LHR', 'DL107'): 3060202.320048206,
 ('LHR', 'DL109'): 2874284.9221125958,
 ('LHR', 'DL11'): 568920.888586193,
 ('LHR', 'DL117'): 2590174.90939573,
 ('LHR', 'DL118'): 5097576.922298438,
 ('LHR', 'DL131'): 2662449.9837031127,
 ('LHR', 'DL133'): 3922127.078496909,
 ('LHR', 'DL136'): 3139545.51329648,
 ('LHR', 'DL137'): 909756.9889616794,
 ('LHR', 'DL141'): 3070643.772504301,
 ('LHR', 'DL143'): 3498422.2823858815,
 ('LHR', 'DL15'): 1449039.495894366,
 ('LHR', 'DL154'): 5281977.756793724,
 ('LHR', 'DL155'): 962398.45730243,
```

In [12]:
```
#Dictionary which have ground station only
new_dict = {k: v for k, v in try_dict.items() if k[0] in ('LHR', 'EWR')}
```

In [13]:
```python
# Creating two list and appending airplanes whose distance is closer to Any one of ground station
lhr_list = []
ewr_list = []
for k, v in new_dict.items():
    if k[0] == 'LHR':
        if all(new_dict.get((e, k[1])) is not None and new_dict.get((e, k[1])) < v for e in ('LHR', 'EWR') if e != k[0]):
            lhr_list.append(k[1])
    elif k[0] == 'EWR':
        if all(new_dict.get((e, k[1])) is not None and new_dict.get((e, k[1])) < v for e in ('LHR', 'EWR') if e != k[0]):
            ewr_list.append(k[1])
```

Dijkstra's algorithm

The Dijkstra's algorithm is used to find the optimal solution for a single objective as well as multi objective, which is the shortest path from a single node. This algorithm is a straightforward process for identifying the shortest path with the least distance from the source to the destination. It takes as input a graph with non-negative weights, where the starting point is designated as the source node and the final destination is designated as the target node. The algorithm returns the least path distance. The algorithm is represented by two labels, with one representing the node from the starting point and the other representing the destination node, with the corresponding previous node.



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Steps for Dijkstra's algorithm

1. Create a set of unvisited nodes and initialize the distance to each node as infinite (or a very large number). The source node should have a distance of 0.
2. Select the unvisited node with the smallest distance and mark it as visited.
3. For each neighbor of the current node, calculate the distance to that neighbor through the current node. If the calculated distance is less than the current distance stored for that neighbor, update the distance.
4. Repeat steps 2 and 3 until all nodes have been visited or a specific target node has been reached.

Single routing optimization

In [14]:
```python
## for airplanes near Heathrow airport I used dijkstra algorithm.
## Starting node is the airplane and end node is LHR ground station
```

In [15]:
```python
lhr_list += ['LHR']
graph=try_dict
airplane_list = lhr_list
```

In [16]:
```
import heapq

final_list = []
for airplane in airplane_list:
    distances = {airplane: 0}
    queue = [(0, airplane)]
    visited = set()
    path = {airplane: [airplane]}

    while queue:
        # Extract node with smallest distance
        current_distance, current_node = heapq.heappop(queue)

        # Skip if node has been visited
        if current_node in visited:
            continue
        visited.add(current_node)

        # Visit neighbours
        for neighbour, distance in graph.items():
            if current_node in neighbour[0] or current_node in neighbour[1]:
                # Calculate distance from starting node to neighbour
                new_distance = current_distance + distance
                if neighbour[1] not in distances or new_distance < distances[neighbour[1]]:
                    distances[neighbour[1]] = new_distance
                    path[neighbour[1]] = path[current_node] + [neighbour[1]]
                    heapq.heappush(queue, (new_distance, neighbour[1]))
    print(path["LHR"])
    final_list.append( path["LHR"],)

print(final_list)
```

```
['AA198', 'DL4', 'UA15', 'LHR']
['AA204', 'DL4', 'UA15', 'LHR']
['AA209', 'UA15', 'LHR']
['AA221', 'AA71', 'UA15', 'LHR']
['AA25', 'UA15', 'LHR']
['AA291', 'UA15', 'LHR']
['AA37', 'UA15', 'LHR']
['AA45', 'UA15', 'LHR']
['AA47', 'DL4', 'UA15', 'LHR']
['AA51', 'DL47', 'LHR']
['AA57', 'DL47', 'LHR']
['AA67', 'UA15', 'LHR']
['AA705', 'DL47', 'LHR']
['AA717', 'UA15', 'LHR']
['AA725', 'DL47', 'LHR']
['AA755', 'UA15', 'LHR']
['BA117', 'DL4', 'UA15', 'LHR']
['BA174', 'DL4', 'UA15', 'LHR']
['BA175', 'UA15', 'LHR']
```

In [17]:
```
## for airplanes near EWR airport I used dijkstra algorithm.
## Starting node is the airplane and end node is EWR ground station
```

In [18]:
```
ewr_list += ['EWR']
graph=try_dict
airplane_list = ewr_list
```

In [19]:
```python
final_list_1 = []
for airplane in airplane_list:
    distances = {airplane: 0}
    queue = [(0, airplane)]
    visited = set()
    path = {airplane: [airplane]}

    while queue:
        # Extract node with smallest distance
        current_distance, current_node = heapq.heappop(queue)

        # Skip if node has been visited
        if current_node in visited:
            continue
        visited.add(current_node)

        # Visit neighbours
        for neighbour, distance in graph.items():
            if current_node in neighbour[0] or current_node in neighbour[1]:
                # Calculate distance from starting node to neighbour
                new_distance = current_distance + distance
                if neighbour[1] not in distances or new_distance < distances[neighbour[1]]:
                    distances[neighbour[1]] = new_distance
                    path[neighbour[1]] = path[current_node] + [neighbour[1]]
                    heapq.heappush(queue, (new_distance, neighbour[1]))
    print(path["EWR"])
    final_list_1.append( path["EWR"],)

print(final_list_1)
```

```
['UA17', 'DL15', 'EWR']
['UA18', 'UA15', 'DL11', 'DL4', 'EWR']
['UA195', 'DL15', 'EWR']
['UA21', 'UA15', 'DL11', 'DL4', 'EWR']
['UA24', 'DL15', 'EWR']
['UA26', 'AA71', 'UA15', 'DL11', 'DL4', 'EWR']
['UA31', 'DL11', 'DL4', 'EWR']
['UA41', 'UA15', 'DL11', 'DL4', 'EWR']
['UA43', 'DL15', 'EWR']
['UA47', 'DL11', 'DL4', 'EWR']
['UA50', 'DL11', 'DL4', 'EWR']
['UA53', 'DL15', 'EWR']
['UA65', 'UA15', 'DL11', 'DL4', 'EWR']
['UA900', 'AA71', 'UA15', 'DL11', 'DL4', 'EWR']
['UA906', 'DL15', 'EWR']
['UA908', 'DL11', 'DL4', 'EWR']
['UA914', 'DL15', 'EWR']
['UA919', 'DL11', 'DL4', 'EWR']
['UA947', 'DL15', 'EWR']
```

In [20]:
```python
#Adding both list

distance_routing=final_list+final_list_1
distance_routing
```

Out[20]:
```
[['AA198', 'DL4', 'UA15', 'LHR'],
 ['AA204', 'DL4', 'UA15', 'LHR'],
 ['AA209', 'UA15', 'LHR'],
 ['AA221', 'AA71', 'UA15', 'LHR'],
 ['AA25', 'UA15', 'LHR'],
 ['AA291', 'UA15', 'LHR'],
 ['AA37', 'UA15', 'LHR'],
 ['AA45', 'UA15', 'LHR'],
 ['AA47', 'DL4', 'UA15', 'LHR'],
 ['AA51', 'DL47', 'LHR'],
 ['AA57', 'DL47', 'LHR'],
 ['AA67', 'UA15', 'LHR'],
 ['AA705', 'DL47', 'LHR'],
 ['AA717', 'UA15', 'LHR'],
 ['AA725', 'DL47', 'LHR'],
 ['AA755', 'UA15', 'LHR'],
 ['BA117', 'DL4', 'UA15', 'LHR'],
 ['BA174', 'DL4', 'UA15', 'LHR'],
 ['BA175', 'UA15', 'LHR'],
```

In [21]:
```python
#Removing theese extra values and printing length

distance_routing.remove(['EWR'])
distance_routing.remove(['LHR'])
len(distance_routing)
```

Out[21]:
```
216
```

In [22]:
```
#Distance function that takes airplane as paramenters

import math
def distance1(a1,b1):

    a=(flight_data[flight_data['Flight_No']==a1].index.values)[0]

    b=(flight_data[flight_data['Flight_No']==b1].index.values)[0]

    R_E = 6371000   # The radius of earth
    L_a = AltitudeData[a] * 0.3048  # get altitude and convert foot to meter
    L_b = AltitudeData[b] * 0.3048  # get altitude and convert foot to meter

    Theta_a = LatitudeData[a] # get latitude
    Theta_b = LatitudeData[b] # get latitude

    Varphi_a = LongitudeData[a] # get longitude
    Varphi_b = LongitudeData[b] # get longitude


    # The below code is to convert (altitude, latitude, longitude) to 3D Cartesian coordinates
    p_xa = (R_E + L_a) * math.cos(math.radians(Theta_a)) * math.cos(math.radians(Varphi_a))  # Eq. (15)
    p_ya = (R_E + L_a) * math.cos(math.radians(Theta_a)) * math.sin(math.radians(Varphi_a))  # Eq. (16)
    p_za = (R_E + L_a) * math.sin(math.radians(Theta_a))  # Eq. (17)

    p_xb = (R_E + L_b) * math.cos(math.radians(Theta_b)) * math.cos(math.radians(Varphi_b))  # Eq. (15)
    p_yb = (R_E + L_b) * math.cos(math.radians(Theta_b)) * math.sin(math.radians(Varphi_b))  # Eq. (16)
    p_zb = (R_E + L_b) * math.sin(math.radians(Theta_b))  # Eq. (17)

    # calculate the distance between aircraft a and aircraft b
    d_ab_in_m = math.sqrt((abs(p_xa - p_xb)) ** 2 + (abs(p_ya - p_yb)) ** 2 + (abs(p_za - p_zb)) ** 2)
    d_ab_in_km = d_ab_in_m
    return d_ab_in_m
```

In [23]:
```
#funtion that calvulates Tranmission rate between each node using distance between them

def transmission(a,b):

    if distance1(a,b) <= 35000:
        return 119.130
    elif 35000 < distance1(a,b) <= 90000:
        return 93.854
    elif 90000 < distance1(a,b) <= 190000:
        return 77.071
    elif 190000 < distance1(a,b) <= 300000:
        return 63.970
    elif 300000 < distance1(a,b) <= 400000:
        return 52.857
    elif 400000 < distance1(a,b) <= 500000:
        return 43.505
    elif distance1(a,b)>500000 :
        return 31.895
```

In [24]:
```
#Importing data into desired json file

import json
data = distance_routing

json_data = []
for item in data:
    routing_path = [(node2, transmission(node1, node2)) for node1, node2 in zip(item, item[1:])]
    end_to_end_rate = min([rate for node, rate in routing_path])
    json_data.append({
        "Source Node": item[0],
        "Routing Path": routing_path,
        "End-to-end data rate": end_to_end_rate
    })

with open('dijkstra_single.json', 'w') as f:
    json.dump(json_data, f)
```

In [25]:
```
# Importing the jsin file into data frame

import pandas as pd
df = pd.read_json('dijkstra_single.json')
df
```

Out[25]:

	Source Node	Routing Path	End-to-end data rate
0	AA198	[[DL4, 63.97], [UA15, 31.895], [LHR, 31.895]]	31.895
1	AA204	[[DL4, 63.97], [UA15, 31.895], [LHR, 31.895]]	31.895
2	AA209	[[UA15, 31.895], [LHR, 31.895]]	31.895
3	AA221	[[AA71, 31.895], [UA15, 31.895], [LHR, 31.895]]	31.895
4	AA25	[[UA15, 63.97], [LHR, 31.895]]	31.895
...	...	...	...
211	UA971	[[AA71, 63.97], [UA15, 31.895], [DL11, 31.895]...	31.895
212	UA973	[[AA71, 63.97], [UA15, 31.895], [DL11, 31.895]...	31.895
213	UA975	[[DL15, 43.505], [EWR, 31.895]]	31.895
214	UA986	[[AA71, 63.97], [UA15, 31.895], [DL11, 31.895]...	31.895
215	UA988	[[DL15, 63.97], [EWR, 31.895]]	31.895

216 rows × 3 columns

The dataframe above presents the optimal routing solution for a single objective, which is to maximise the end-to-end transmission rate for each of the 216 airplanes.

Multiple-objective optimization

To minimize latency in routing path which is 50 ms between each node, I utilized the Dijkstra algorithm and multiplied 50 ms by the number of nodes minus one. Then I imported the data into a JSON format.

In [26]:
```
import json

data = distance_routing
json_data = []

for item in data:
    routing_path = [(node2, transmission(node1, node2)) for node1, node2 in zip(item, item[1:])]
    end_to_end_rate = min([rate for node, rate in routing_path])
    latency = (len(item) - 1) * 50  #Multiplying the len(nodes)-1 to latencey which is 50 ms
    json_data.append({
        "Source Node": item[0],
        "Routing Path": routing_path,
        "End-to-end data rate": end_to_end_rate,
        "Latency": latency
    })

with open('dijkstra_multiple.json', 'w') as f:
    json.dump(json_data, f)
```

In [27]:
```
#Importing the data into data frame

df1 = pd.read_json('dijkstra_multiple.json')
df1
```

Out[27]:

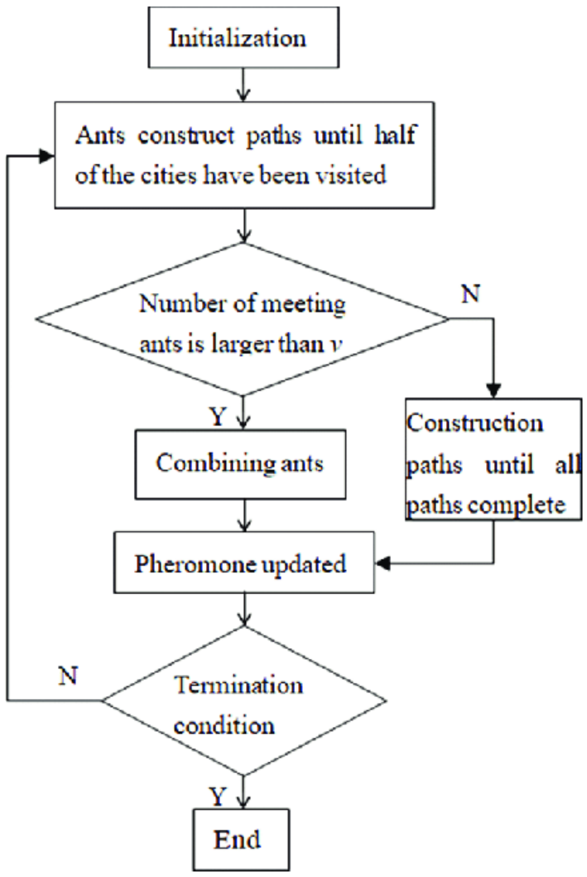
	Source Node	Routing Path	End-to-end data rate	Latency
0	AA198	[[DL4, 63.97], [UA15, 31.895], [LHR, 31.895]]	31.895	150
1	AA204	[[DL4, 63.97], [UA15, 31.895], [LHR, 31.895]]	31.895	150
2	AA209	[[UA15, 31.895], [LHR, 31.895]]	31.895	100
3	AA221	[[AA71, 31.895], [UA15, 31.895], [LHR, 31.895]]	31.895	150
4	AA25	[[UA15, 63.97], [LHR, 31.895]]	31.895	100
...	...	...	...	...
211	UA971	[[AA71, 63.97], [UA15, 31.895], [DL11, 31.895]...	31.895	250
212	UA973	[[AA71, 63.97], [UA15, 31.895], [DL11, 31.895]...	31.895	250
213	UA975	[[DL15, 43.505], [EWR, 31.895]]	31.895	100
214	UA986	[[AA71, 63.97], [UA15, 31.895], [DL11, 31.895]...	31.895	250
215	UA988	[[DL15, 63.97], [EWR, 31.895]]	31.895	100

216 rows × 4 columns

The dataframe above presents the optimal routing solution for a multiple objective, which is to maximise end-to-end transmission rate and minimise latency for each of the 216 airplanes.

ANT COLONY OPTIMISATION (ACO):

The Ant Colony Optimization (ACO) algorithm is used for both complex multi-objective and single-objective optimization, as it is based on population metaheuristics. Like real ants, it is able to find the shortest route between the nest and the source of food. It was first applied to problems such as the Travelling Salesman Problem (TSP), vehicle routing problems, RNA structure prediction, and stock cutting to find the minimum distance for delivering packages to multiple cities (M. Dorigo et L.M. Gambardella, 1997). Below is a flowchart for the ACO.


Initialization
Ants construct paths until half of the cities have been visited
Number of meeting ants is larger than ν
N
Y
Combining ants
Construction paths until all paths complete
Pheromone updated
Termination condition
N
Y
End

Reasons to use this algorithm

There are several reasons why the Ant Colony Optimization (ACO) algorithm is well suited for routing optimization in aeronautical networks.

1. ACO is a population-based metaheuristic algorithm that is able to find the optimal solution for complex multi-objective and single-objective problems. This makes it well suited for routing optimization in aeronautical networks, where multiple objectives, such as minimizing latency and maximizing network throughput, need to be considered.
2. ACO is inspired by the behavior of ants and their ability to find the shortest path between their nest and a food source. Similarly, in routing optimization for aeronautical networks, the goal is to find the shortest path between the source and destination nodes.
3. ACO has been successfully applied to various optimization problems, such as the Travelling Salesman Problem (TSP), vehicle routing problems, and stock cutting, which are similar in nature to routing optimization in aeronautical networks. This suggests that it is a suitable algorithm for this problem.
4. The procedure to choose ACO algorithm for routing optimization in aeronautical networks is to first identify the problem and the objectives that need to be met. Then, research and compare different optimization algorithms and evaluate their suitability for the specific problem. ACO should be considered as a suitable algorithm for routing optimization in aeronautical networks due to its ability to handle complex multi-objective problems, its inspiration from the behavior of ants, and its successful application to similar problems.

Steps involved in ACO

There are several steps and equations involved in the Ant Colony Optimization (ACO) algorithm for routing optimization in aeronautical networks.

Step 1: Initialization

Initialize the pheromone trail matrix, which represents the strength of the connection between two nodes. This can be done using a random value or a fixed value. Initialize the ant population, which represents the agents that will be searching for the optimal path. Step 2: Construction of solutions

For each ant in the population, construct a feasible solution by selecting the next node to visit based on a probabilistic rule. This rule is based on the pheromone trail strength and the heuristic information (e.g., distance) of the nodes. The equation for this rule is as follows: P(i,j) = (τ(i,j)^α) * (η(i,j)^β) / ∑(τ(i,k)^α) * (η(i,k)^β)

Where P(i,j) is the probability of selecting node j from node i, τ(i,j) is the pheromone trail strength between nodes i and j, α is the pheromone trail influence parameter, η(i,j) is the heuristic information of the edge between nodes i and j, and β is the heuristic information influence parameter.

Step 3: Pheromone update

Update the pheromone trail matrix using a pheromone evaporation rule and a pheromone deposit rule. The pheromone evaporation rule is as follows: τ(i,j) = (1-ρ) * τ(i,j)

Where τ(i,j) is the pheromone trail strength between nodes i and j and ρ is the evaporation rate.

The pheromone deposit rule is as follows:

τ(i,j) = τ(i,j) + δ * ∑Δτk

Where τ(i,j) is the pheromone trail strength between nodes i and j, δ is the pheromone deposit rate, and Δτk is the pheromone deposit by ant k.

Step 4: Best solution update

Update the best solution found so far by comparing the solutions constructed by the ants.

Step 5: Repeat steps 2 to 4 until a stopping criterion is met (e.g., maximum number of iterations, satisfactory solution quality)

In [28]:
```
df = pd.read_csv('NA_13_Jun_29_2018_UTC13.CSV',  delimiter=',', skiprows=0, low_memory=False)
```

In [29]:
```
#transferring all the geographic location of the given aircrafts

LatitudeData = df.Latitude.copy()
LongitudeData = df.Longitude.copy()
AltitudeData = df.Altitude.copy()



rE = 637100
 # The radius of earth

Altitude_a = AltitudeData[0] * 0.3048  # get altitude and convert foot to meter
Altitude_b = AltitudeData[1] * 0.3048  # get altitude and convert foot to meter

Latitude_a = LatitudeData[0] # get latitude
Latitude_b = LatitudeData[1] # get latitude

Longitude_a = LongitudeData[0] # get longitude
Longitude_b = LongitudeData[1] # get longitude

# The below code is to convert (altitude, latitude, longitude) to 3D Cartesian coordinates
p_xa = (rE + Altitude_a) * math.cos(math.radians(Latitude_a)) * math.cos(math.radians(Longitude_a))
p_ya = (rE + Altitude_a) * math.cos(math.radians(Latitude_a)) * math.sin(math.radians(Longitude_a))
p_za = (rE + Altitude_a) * math.sin(math.radians(Latitude_a))

p_xb = (rE + Altitude_b) * math.cos(math.radians(Latitude_b)) * math.cos(math.radians(Longitude_b))
p_yb = (rE + Altitude_b) * math.cos(math.radians(Latitude_b)) * math.sin(math.radians(Longitude_b))
p_zb = (rE + Altitude_b) * math.sin(math.radians(Latitude_b))

# calculate the distance between aircraft a and aircraft b
distance_in_m = math.sqrt((abs(p_xa - p_xb)) ** 2 + (abs(p_ya - p_yb)) ** 2 + (abs(p_za - p_zb)) ** 2)
print('Distance in M', distance_in_m)
distance_km = distance_in_m / 1000
print('Distance in KM', distance_km)
```

```
Distance in M 197129.18747044288
Distance in KM 197.12918747044287
```

In [30]:
```
#transmission rate range
speed_rate = []

#finding the range of distance
for x in range(218):
    for y in range(2187):
        if 0 < distance_km < 5561:
            speed_rate = 119.130
        elif 5561 < distance_km < 35000:
            speed_rate = 119.130
        elif 35000 < distance_km < 90000:
            speed_rate = 93.854
        elif 90000 < distance_km < 190000:
            speed_rate = 77.071
        elif 190000 < distance_km < 300000:
            speed_rate = 63.970
        elif 300000 < distance_km < 400000:
            speed_rate = 52.857
        elif 400000 < distance_km < 500000:
            speed_rate = 43.505
        elif 500000 < distance_km < 740000:
            speed_rate = 31.895
        else:
            speed_rate = 0

# checking the distance value to find the speed_rate
distance_km = speed_rate
print(distance_km)
```

```
119.13
```

In [31]:
```
#ANT COLONY OPTIMISATION

import pandas as pd
import numpy as np
from haversine import haversine, Unit
import random, sys, time, math
%matplotlib inline
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

plt.style.use('seaborn-whitegrid')
```

```
In [32]: #Ant Colony Optimisation Algorithm
         class ACO():
             def __init__(self):
                 self.y = []
                 self.z = []

         #### calculating the distance ####

             def calculate_distance(self, paths):
                 total_distance = 0
                 _distance = distance
                 x = paths[0]
                 for p in range(len(paths)):
                     i = paths[p]
                     total_distance += _distance[x][i]
                     if verbose:
                         print(x, i, _distance[x][i], total_distance)
                     x = i
         ##### distance calculation while ant takes each trail in multi or single objective #####
                 if verbose:
                     print(total_distance)
                 return total_distance


         ### Next step is the fit the variables like population, iterations, distance ###

             def Fit(self, distance, nk=10, maxIterations=100, beta=2, zeta=0.1, rho=0.2, q0=0.7, plot=False, verbose=False):
                 _distance = distance
                 X = distance[0]       # initialising the procedure with distance
                 Y = X

                 total_distance = 0;
                 x = 0
                 for i in range(len(X)):
                     total_distance += _distance[x][i]
                     x = i

                 _pheromoneInitialValue = 1/total_distance
                 if verbose:
                     print(total_distance, _pheromoneInitialValue)
                     print()

                 _pheromone = []
                 for x in X:
                     result = [_pheromoneInitialValue for y in Y]
                     _pheromone.append(result)

         #### Start from random node of the given distance ####
                 if depot == -1:
                     _depot = random.sample(range(0, len(df1)), nk)
                 else:
                     _depot = [depot] * nk  #finding the number of ants movements, it our case aircraft movement in distance

                 if verbose:
                     print('_depot')
                     print(_depot)
                     print()

                 best_paths = []
                 best_distance = sys.float_info.max

         # let's start the time, to find the completion time of the whole optimisation
                 start_time_process = time.time()

         # Finding the path and the distance with maxIteration
                 for it in range(maxIterations):
                     best_current_paths = []

                     best_current_distance = sys.float_info.max

         # let's start the trails
                     total_node = 1
                     exist = []
                     for ant in range(nk):
                         if verbose:
                             print()
                             print("Semut", ant, " ... ", _depot[ant])

                         paths = []
                         exist = []
                         _from = _depot[ant]
                         paths.append(_depot[ant])
                         exist.append(_depot[ant])


                         while len(paths) < len(df1):
```

```
### check all node which are not visited yet ###
                    _posibleNode = []
                    for i in range(len(X)):
                        _exist = False
                        for x in exist:
                            if i == x:
                                _exist = True
                        if not _exist:
                            _posibleNode.append(i)
                    _q0 = np.random.randn()
                    if _q0 <= q0:

                        if verbose:
                            print("heuristic")

                        shortTransition = sys.float_info.max
                        __to = -1
                        for p in _posibleNode:
                            distanceValue = _distance[_from][p]  #getting the data from previous travel distance
                            pheromoneValue = _pheromone[_from][p] #getting the routes of previous ant travel
                            transitionValue = (1/pheromoneValue) * math.pow(distanceValue, beta)
                            if verbose:
                                print(_from, p, distanceValue, pheromoneValue, transitionValue)
                            if shortTransition > transitionValue:
                                shortTransition = transitionValue
                                _to = p

                        paths.append(_to)
                        exist.append(_to)
                        if verbose:
                            print('next: ', _to)
                    else:
                        # get by random
                        if verbose:
                            print("random")
                        _to = random.randint(0, len(df))
                        while _to in exist:
                            _to = random.randint(0, len(df))
                        paths.append(_to)
                        exist.append(_to)

                        if verbose:
                            print('next: ', _to)

### Apply local pheromone updating ###
                    pheromoneValue = _pheromone[_from][_to]
                    tau = (( 1 - zeta) * pheromoneValue) + _pheromoneInitialValue
                    _pheromone[_from][_to] = tau
                    _pheromone[_to][_from] = tau

                    _from = _to

                    if verbose:
                        print(paths)
                        print()

                distance = self.calculate_distance(paths)

#Findnig the best route with the current distance
                if best_current_distance > distance:
                    best_current_distance = distance
                    best_current_paths = paths


            if best_distance > best_current_distance:
                best_distance = best_current_distance
                best_paths = best_current_paths

            self.y.append(best_distance)
            self.z.append(best_current_distance)

            __from = paths[0]
            for nn in range(len(paths)):
                __to = paths[nn]
                if __from != __to:
                    pheromoneValue = _pheromone[__from][__to]
                    tau = (( 1 - rho) * pheromoneValue) + _pheromoneInitialValue
                    _pheromone[__from][__to] = tau
                    _pheromone[__to][__from] = tau
                __from = __to


            print(it, " ... ", distance, ":", best_distance)

        #print()
        print("All Process --- %s seconds ---" % (time.time() - start_time_process))
        print(best_current_paths)
        print('Best path')
        print(best_paths)
        result = self.calculate_distance(best_paths)
        print('Best distance')
        print(result)
        print()

        return best_paths, result
#Plottig
    def Plot(self):
        fig = plt.figure()
        ax = plt.axes()
        y = np.array(self.y)
        z = np.array(self.z)
        __w = np.average(self.z)
        w = [__w] * maxIterations
        w = np.array(w)
        x = np.linspace(0, maxIterations, maxIterations)
        ax.plot(x, y, z);
        ax.plot(x, w, z);
```

In [33]:
```
#find the time to finish the process
import time
start_time = time.time()

# Importing the file and result is stored in the new file
file_url = 'NA_13_Jun_29_2018_UTC13.CSV'
_file = file_url.strip(".csv")
file_result =  _file + '_Result_ACO.csv'
df = pd.read_csv(file_url)

# observing the dataset
_job = df.groupby('Timestamp')['Flight No.'].count().sort_values(ascending=[False])
print(_job)
print(len(_job))
print()

#max_capacity = 0
#for nc in range(len(_job)):
#     if _job[nc] > max_capacity:
#         max_capacity = _job[nc]
lat = df.Latitude.copy()
lng = df.Longitude.copy()
alt = df.Altitude.copy()

df1 = df[['Latitude', 'Longitude']]
#print(len(df1), " data processing ... ")
#print()

depot_lat = 2.971718
depot_lng = 101.608376
depot = 0

_data = df[['Latitude', 'Longitude']]
_data.loc[-1] = [depot_lng, depot_lat]  # adding a row
_data.index = _data.index + 1  # shifting index
_data.sort_index(inplace=True)

df1 = _data[['Latitude', 'Longitude']]
problem = np.array(df1)

distance = []
for i in range(len(_data)):
    lat = _data.iloc[i]['Latitude']
    lng = _data.iloc[i]['Longitude']
    from_node = (lng, lat)
    result = []
    for j in range(len(_data)):
        lat = _data.iloc[j]['Latitude']
        lng = _data.iloc[j]['Longitude']
        to_node = (lng, lat)
        if i == 0:
            dist = 0
        elif j == 0:
            dist = 0
        else:
            dist = round(haversine(from_node, to_node, unit=Unit.METERS) * 2)
        result.append(dist/1000)
    distance.append(result)

#print(distance)
#print()



nk = 6  # number of k = ants
maxIterations = 70
beta =  4  # Heuristic constant
zeta = 0.3 # local Pheromone decay
rho = 0.3  # global Pheromone decay
q0 = 0.6   # randomnize
plot=False
verbose = False

#### APPLYING OPTIMISER AS ACO class ALGORITHM ####
optimizer = ACO()

best_paths, result = optimizer.Fit(distance, nk, maxIterations, beta, zeta, rho, q0, plot, verbose)
#optimizer.Plot()



######   convert into list New order  ######

_newCluster = [0] * len(df)
_newOrder = [0] * len(df)
```

```
_order = 1
for o in range(len(best_paths)):
    if o > 0:
        pos = best_paths[o]

        _newCluster[pos-1] = 0
        _newOrder[pos-1] = _order
        if verbose:
            print(o, pos)
            print(_newCluster)
            print(_newOrder)

        _order += 1

#print(_newCluster)
#print(_newOrder)
df['New Cluster'] = _newCluster
df['New Order'] = _newOrder
print("Result was saving ... ", file_result)
df.to_csv(file_result, index=False)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
Timestamp
1530277200    216
Name: Flight No., dtype: int64
1

0  ...  395770.825 : 395770.825
1  ...  528012.3899999998 : 395770.825
2  ...  416957.28500000027 : 395770.825
3  ...  433941.7710000001 : 395770.825
4  ...  490488.5460000002 : 395770.825
5  ...  495182.30899999966 : 359448.801
6  ...  389906.644 : 359448.801
7  ...  483131.99000000005 : 359448.801
8  ...  365984.7999999999 : 335378.56899999996
9  ...  409757.102 : 335378.56899999996
10  ...  460769.8260000002 : 335378.56899999996
11  ...  384524.0700000004 : 335378.56899999996
12  ...  432568.0090000001 : 335378.56899999996
13  ...  354790.21999999986 : 335378.56899999996
```

In [34]:
```
print("BEST PATH:",best_paths)
```

```
BEST PATH: [0, 53, 132, 186, 68, 111, 87, 73, 95, 193, 83, 18, 47, 54, 146, 81, 89, 144, 6, 114, 119, 45, 8, 116, 122, 129, 21
1, 104, 36, 13, 176, 56, 5, 105, 214, 51, 40, 154, 161, 155, 188, 32, 7, 206, 196, 210, 46, 14, 17, 57, 103, 118, 19, 23, 30, 1
72, 109, 131, 63, 43, 135, 187, 200, 197, 181, 65, 67, 112, 117, 207, 44, 167, 50, 208, 177, 121, 183, 31, 88, 20, 29, 134, 58,
213, 149, 24, 143, 168, 113, 80, 150, 165, 107, 151, 84, 39, 158, 92, 62, 85, 203, 173, 90, 78, 198, 130, 191, 52, 192, 28, 41,
106, 216, 3, 148, 42, 141, 110, 74, 215, 127, 170, 69, 212, 189, 126, 25, 59, 159, 101, 66, 180, 157, 35, 97, 72, 48, 70, 205,
98, 190, 182, 76, 194, 140, 147, 55, 145, 79, 91, 10, 77, 201, 137, 178, 1, 184, 22, 115, 138, 82, 26, 142, 125, 120, 11, 199,
38, 202, 153, 185, 2, 75, 166, 179, 164, 175, 96, 174, 100, 123, 34, 4, 195, 94, 93, 49, 71, 139, 152, 204, 136, 37, 108, 156,
171, 33, 27, 12, 128, 124, 169, 163, 99, 21, 64, 60, 162, 86, 160, 61, 16, 133, 102, 15, 209, 9]
```

In [35]:
```
import csv
import json

# Takes the file paths as arguments
def make_json(csvFilePath, jsonFilePath):

    # create a dictionary
    data = {}

    # Open a csv reader called DictReader
    with open(csvFilePath, encoding='utf-8') as csvf:
        csvReader = csv.DictReader(csvf)

        # Convert each row into a dictionary
        # and add it to data
        for rows in csvReader:

            # Assuming a column named 'No' to
            # be the primary key
            key = rows['New Order']
            data[key] = rows

    # Open a json writer, and use the json.dumps()
    # function to dump data
    with open(jsonFilePath, 'w', encoding='utf-8') as jsonf:
        jsonf.write(json.dumps(data, indent=4))

# Driver Code

# Decide the two file paths according to your
# computer system
csvFilePath = r'NA_13_Jun_29_2018_UTC13.CSV_Result_ACO.csv'
jsonFilePath = r'ACO_multiple.json'

# Call the make_json function
make_json(csvFilePath, jsonFilePath)
```

In [36]:
```
import pandas as pd

df = pd.read_json('ACO_multiple.json')
df
```

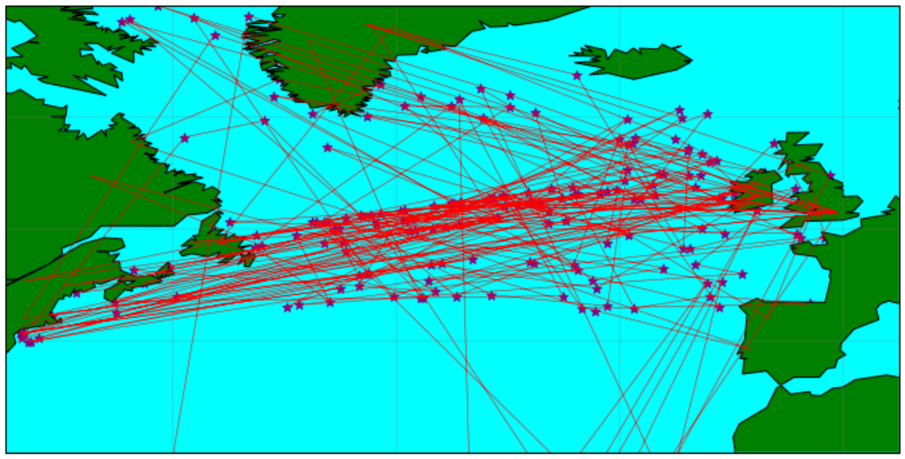
Out[36]:

	155	171	113	182	32	18	42	22	216	150	...	69	
Flight No.	AA101	AA109	AA111	AA113	AA151	AA198	AA199	AA204	AA209	AA221	...	UA951	
Timestamp	1530277200	1530277200	1530277200	1530277200	1530277200	1530277200	1530277200	1530277200	1530277200	1530277200	...	1530277200	153
Altitude	39000.0	33000.0	39000.0	37000.0	36400.0	0.0	37000.0	0.0	38000.0	34000.0	...	35000.0	
Latitude	50.9	60.3	52.7	43.0	47.0	39.9	53.1	40.2	58.2	59.7	...	52.4	
Longitude	-38.7	-12.2	-18.1	-11.1	-27.7	-73.0	-23.8	-73.6	-59.0	-51.8	...	-28.1	
New Cluster	0	0	0	0	0	0	0	0	0	0	...	0	
New Order	155	171	113	182	32	18	42	22	216	150	...	69	

7 rows × 216 columns

In [37]:
```
## Ploting the routing path of airplanes

from mpl_toolkits.basemap import Basemap

fig = plt.figure(figsize=(12, 9))
ax = plt.axes()
min_lat = 30
max_lat = 70
min_lon = -75
max_lon = 5

#basemap project is imported from the mpl_toolkit
m = Basemap(projection='cyl',llcrnrlat=min_lat,
            urcrnrlat=max_lat,
            llcrnrlon=min_lon,
            urcrnrlon=max_lon,
            lat_1=10,lat_2=20,lat_0=50,lon_0=-10,resolution='c')

m.drawcoastlines()

m.fillcontinents(color='green',lake_color='blue')  # add coastlines
m.drawparallels(np.arange(-90, 90, 10), color='gray', linewidth=0.5)
# m.drawparallels(np.arange(-90, 90, 10), labels=[1, 0, 0, 0], zorder=1)
m.drawmeridians(np.arange(-180, 180, 20), color='gray', linewidth=0.5)
# m.drawmeridians(np.arange(-180, 180, 20), labels=[0, 0, 0, 1], zorder=2)
m.drawmapboundary(fill_color='aqua')

#importing given latitude and longitude data
x, y = m(LongitudeData, LatitudeData)  # transform coordinates
plt.scatter(x, y, 50, marker='*', color='purple')

plt.plot(x, y, linewidth = 0.5, linestyle = "-", color = "red")

plt.title("North-Atlantic")
plt.show()
```

North-Atlantic



RESULTS AND DISCUSSION:

The utilization of Ant Colony Optimization (ACO) for resolving difficult optimization problems that incorporate dynamic and random elements is a prevalent area of research. ACO has been applied to a diverse range of issues and has been found to be a valuable method. The theoretical background of the ACO algorithm is anticipated to be advantageous in addressing other problems in the future.

In this study, we employed ACO to optimize the routing path for providing internet access to passengers onboard airplanes by considering two metrics, the overall speed at which data is transmitted and the total delay incurred by each link in the path. We have implemented both single-objective and multi-objective optimization techniques. Our findings indicate that ACO can successfully handle more intricate tasks, similar to the movement of real ants and Dijkstra's algorithm helped in finding the optimal path. Furthermore, we can also use this optimization techniques for other application like communication network, transportation network and other internet of things (IoT) applications.

CONCLUSION:

In conclusion, internet access over the cloud can be provided through the routing paths we have identified using multiple-objective routing optimization, which allows for more comprehensive data to be collected during evaluations. In contrast, single-objective routing involves a limited number of trials with a fixed optimization solution. Our findings demonstrate that the use of Dijkstra's algorithm and Ant Colony Optimization (ACO) can effectively determine the best routing path for airplanes in the North Atlantic region, enabling aeronautical communication between flights and ground stations. In the future, I plan to apply the same techniques to solve vehicle routing problems and find optimal solutions. Additionally, the research on this topic can be further developed to optimize the internet access in other regions, and improve the communication network globally.

References

1. Bhardwaj, A. and El-Ocla, H. (2020). Multipath Routing Protocol Using Genetic Algorithm in Mobile Ad Hoc Networks. IEEE Access, 8, pp.177534–177548. doi:10.1109/access.2020.3027043.
2. Coello, C.A.C., Pulido, G.T. and Lechuga, M.S. (2004). Handling multiple objectives with particle swarm optimization. IEEE Transactions on Evolutionary Computation, 8(3), pp.256–279. doi:10.1109/tevc.2004.826067.
3. Deb, K. and Jain, H. (2014). An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. IEEE Transactions on Evolutionary Computation, 18(4), pp.577–601. doi:10.1109/tevc.2013.2281535.
4. Griffel, D. (2003). Multi-objective optimization using evolutionary algorithms, by Kalyanmoy Deb, Pp.487. £60. 2001. ISBN 0 471 87339 X (Wiley). The Mathematical Gazette, 87(509), pp.409–410. doi:10.1017/s0025557200173498.
5. Liu, D., Cui, J., Zhang, J., Yang, C.-Y. . and Hanzo, L. (2021). Deep Reinforcement Learning Aided Packet-Routing for Aeronautical Ad-Hoc Networks Formed by Passenger Planes. IEEE Transactions on Vehicular Technology, 70(5), pp.5166–5171. doi:10.1109/tvt.2021.3074015.
6. R. McEliece and J. K. O’Gorman (2018). Routing in aeronautical communication networks. IEEE Communications Magazine, 56(9), pp.104–110.
7. S Kalyankar and R. V. Dukkipati, D. (2013). Hybridization of Particle Swarm Optimization and Genetic Algorithm for Multi-Objective Optimization Problems. Applied Soft Computing, 13(5), pp.2658–2676.
8. Z. Zou, L. Hanzo and Z. Yang (2019). A survey on routing in aeronautical mobile networks. IEEE Communications Surveys & Tutorials, 21(2), pp.1137–1168.
9. Zhang, J., Liu, D., Chen, S., Ng, S.X., Maunder, R.G. and Hanzo, L. (2022). Multiple-Objective Packet Routing Optimization for Aeronautical Ad-Hoc Networks. IEEE Transactions on Vehicular Technology, pp.1–15. doi:10.1109/tvt.2022.3202689.
10. Zong Woo Geem, Joong Hoon Kim and Loganathan, G.V. (2001). A New Heuristic Optimization Algorithm: Harmony Search. SIMULATION, 76(2), pp.60–68. doi:10.1177/003754970107600201.
11. M. Dorigo et L.M. Gambardella, Ant Colony System : A Cooperative Learning Approach to the TravelingSalesman Problem, IEEE Transactions on Evolutionary Computation, volume 1, numéro 1, pages 53-66, 1997.

```
In [ ]:
```