Optimal routing path of Multi-Output Data Packets for Aeronautical Networks

Author: David Cuellar
s5535743@bournemouth.ac.uk (mailto:s5535743@bournemouth.ac.uk)
Dept. Computing & Informatics
Bournemouth University

Search and Optimization
Professor Jiankang Zhang Ph.D.

Abstract

The rise of the Internet and Industry 4.0 has resulted in numerous changes in how people live and work. The aeronautical industry is one of those that has been severely impacted. The use of commercial flights has grown in popularity, as has the need for effective communication systems. The Aeronautical Ad hoc NETwork (AANET) is a wireless communication system that allows aircraft to communicate with one another as well as with ground stations. The application of various optimization algorithms is critical in determining the best data packet routing path within the AANET. The impact of the Internet and Industry 4.0 on the aeronautical industry, the use of commercial flights and the need for efficient communication systems, the Aeroneutical Ad hoc NETwork (AANET) and its importance, and the use of optimization algorithms in determining the optimal data packet routing path within the AANET will all be discussed, having important metrics such as end-to-end transmission rate and end-to-end latency.

Kewwords - AANET,Industry 4.0, Internet, Optimal Routing Path.

1. Introduction

Today, due to Industry 4.0, internet access is becoming more and more important for society. Airborne mobile communications are being explored for more efficient, faster and cheaper in-flight access. Airline companies are motivated to develop communications infrastructures that provide Internet access to users (Neji et al. 2013). Some companies offer this service using satellite communication, however, Zhang et al. (2019) note that “they suffer from expensive subscription, limited coverage, limited capacity, and high end-to-end delay”.

Aeronautical networks present routing optimization problems for providing Internet access to onboard passengers. Vey et al. (2014) propose Aeronautical Ad hoc NETwork (AANET) as a solution for aeronautical communications, which consists of involving each aircraft as a network node, capable of sending and receiving signals; it is necessary to have a Ground Station that provides the main signal. For this, it is necessary to find an optimal data packet routing path to a ground station. Important metrics such as the end-to-end transmission rate, end-to-end latency, the end-to-end spectral efficiency (SE), and the path expiration time (PET) (Zhang et al. 2022).

In this report, Dijkstras Algorithm is introduced and improved to give multiple outputs as a possible solution to optimize network routing taking into account the end-to-end transmission rate and the end-to-end latency metrics. The algorithm is used to find the Longest Path, taking into account that the highest end-to-end transmission rate is desired and lowest end-to-end latency. Additionally, a solution to the multiple possible routes with the same end-to-end transmission rate is proposed. To evaluate the algorithm, its results will be evaluated with a comparison with the Breadth-First Search Algorithm.

2. Literature review

Next, the potential methods to find the longest path route will be explained:

A. Dijkstra's Algorithm

This algorithm created by Edsger W. Dijkstra “solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time” (Javaid 2013). The technique operates by keeping track of a set of vertices V with known shortest distances to the source vertex S. If the distance to v through u is less than its current distance, it repeatedly chooses the vertex u from V with the smallest distance d[u] from the source and updates the distance of all the vertices v next to u. It can also determine the longest path with the greatest distance d[u] with only minor adjustments (Gass and Fu 2013).

The steps presented by various research (Gass and Fu 2013; Javaid 2013; Johnson 1973; Vey et al. 2014) are very similar. Here is an outline of the steps the algorithm to find the shortest path follows:

1. Initialize the distances of all vertices to **infinity**, except for the source vertex which has a distance of zero.
2. Change the current vertex to be the source vertex.
3. Consider the neighbours of the current vertex and estimate their approximate distances. The tentative distance for a neighbour v that is not in V is determined by adding the weight of the edge that connects the current vertex to the neighbour to the distance from the source to the current vertex.
4. If the tentative distance of a neighbor v is **less** than its current distance, update the distance of v to the new **lower** value.
5. Include the existing vertex in the group V of established vertices.
6. Choose the vertex that is outside of V and make it the current vertex by choosing it. In the absence of such a vertex, the procedure is complete.

7.Continue until all vertices have been added to V by repeating steps 3 through 6.

At the conclusion of the procedure, the predecessor array pred[] will have the predecessor vertex of each vertex in the shortest path from the source vertex, and the distance array d[i] will contain the shortest distances from the source vertex to all other vertices. By working backwards from the target vertex and using these arrays, it is possible to reconstruct the shortest path from the source to any vertex (Javaid 2013).

Bulterman et al. (2002) in their article ”On computing a longest path in a tree” makes a concise definition of how to implement Dijkstra’s Algorithm to find the longest path:

```
"Build a physical model of the tree by connecting each pair of adjacent nodes by a piece of string of the given
edge length. Now pick up the physical tree at an arbitrary node U, let the contraption hang down, and determine
a deepest node X. Then pick up the tree at X and determine a deepest node Y . The claim is that the path betwee
n X and Y is a longest path in the tree” (Bulterman et al. 2002)
```

What it comes down to is that in step 4. above, it would change to: "If the tentative distance of a neighbor v is **greater** than its current distance, update the distance of v to the new **higher** value.". Additionally, for this to work, in step 1. all vertices except the source must be initialized to **minus infinity**.

In addition, the authors (Bulterman et al. 2002) make the mathematical demonstration of its operation.

This method was selected due to our extensive comprehension of it and the simplicity with which it can be put into practice.

B. Breadth-First Search

The Breadth-First Search (BFS) algorithm is a popular algorithm for traversing and searching graphs and trees (Awerbuch and Gallager 1987; Kurant et al. 2010). It is a systematic way of visiting each vertex of a graph and traversing its edges in the most efficient way possible. The BFS algorithm is particularly useful in finding the shortest path in a matrix, regardless of the weights at each node (Kurant et al. 2010).

The BFS algorithm begins by visiting the starting vertex and then traversing through its neighbours in a breadth- first manner. It continues this process until it reaches the destination vertex or finds that there is no path from the starting vertex to the destination vertex (Awerbuch and Gallager 1987).

Overall, the BFS algorithm is a powerful tool for finding the shortest path in a matrix. While it has some limitations, it can be a useful tool for solving problems related to network latency and routing. With a little modification, it can be useful for a variety of other graph traversal and search problems (Kurant et al. 2010; Awerbuch and Gallager 1987).

C. Genetic Algorithm

A genetic algorithm (GA) can be used to determine the longest path routing in a network (Portugal et al. 2010). The core idea inspired in the theory about evolution by Darwin is to create a population of viable solutions using the concepts of natural selection and evolution, and then iteratively improve the population using genetic operators like mutation, crossover, and selection (Portugal et al. 2010).

The problem must be defined, and the solutions must be represented, before a GA may be used for longest path routing (Portugal et al. 2010). Finding the longest path in a network is the issue here, and a list of nodes that represent the path might be one way to express the solutions. The longest path can contain cycles, but it is unusual, since you can keep going without reaching the last node (Portugal et al. 2010)

D. Particle Swarm Optimization

Another optimization method for determining the longest path routing in a network is Particle Swarm Optimization (PSO). Similar to a genetic algorithm, PSO developed by Kennedy and Eberhart (Kennedy and Eberhart 1995) uses natural system principles to incrementally enhance a collection of solutions known as particles (Chen et al. 2006).

Each particle in PSO is a potential solution to the problem, and the particles navigate the solution space using their own ”personal best” positions and the ”global best” positions of the entire swarm (Chen et al. 2006). It’s crucial to take into account the computational complexity of the algorithm and any potential limitations of the problem because the problem’s complexity could be quite high and the graph’s edges and nodes are numerous (Chen et al. 2006).

3. Methodology

3.1. Data importing and fixing

To work with the data, it was first necessary to follow the following steps:

1. Import the data: When analyzing the first and last 5 data it is observed that there are 216 data, that all are at the same Timestamp and that their coordinates are in polar system.
2. Add the two Ground Stations (GS) in the last two rows of the dataframe.
3. Create a function to change the polar coordinates (Altitude, Latitude, Longitude) to Cartesian coordinates (Px, Py, Pz).
4. Create a new dataframe with polar and Cartesian coordinates (df_res).
5. Plot the data Px, Py, Pz of each Node. Differentiate with blue circles the aircraft and with red Xs the GS: Analyze the graph and with this you can make assumptions of how some results will look like (like some aircraft are close to the GS and others will not have routes since they are far away from any GS).
6. Calculate the Euclidean distance between each node. Matrix of n x n (In this case 218 x 218).
7. Create a Transmission Rate matrix and make a matrix (tr_df) of the relationship between each Euclidean distance and the Transmission Rate matrix.

In [1]:

```python
#####-----------------------------------------#####
#---- Data importing and fixing: Step 1       ----#
#####-----------------------------------------#####

!pip install natsort

#Import libraries

#To read and write data
import pandas as pd
import json

#To analyze data
from collections import Counter, OrderedDict
import natsort

#To plot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#Math packages
import math
import numpy as np
import scipy.spatial.distance as dist

#To calculate times
import time

#To convert matrix to graphs
import networkx as nx

#Read data
datapath = "data.csv"
df = pd.read_csv(datapath)

#Show data
df
```

```
Requirement already satisfied: natsort in /Users/davidcuellar/opt/anaconda3/lib/python3.9/site-packages (8.2.0)
```

Out[1]:

	Flight No.	Timestamp	Altitude	Latitude	Longitude
0	AA101	1530277200	39000.0	50.9	-38.7
1	AA109	1530277200	33000.0	60.3	-12.2
2	AA111	1530277200	39000.0	52.7	-18.1
3	AA113	1530277200	37000.0	43.0	-11.1
4	AA151	1530277200	36400.0	47.0	-27.7
...	...	...	...	...	...
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

216 rows × 5 columns

In [2]:

```python
#####-----------------------------------------#####
#---- Data importing and fixing: Step 2       ----#
#####-----------------------------------------#####

#Create Ground Stations (GS)
LHR = {'Flight No.' : 'GS_LHR',
       'Timestamp' : 1530277200,
       'Altitude': 81.73,
       'Latitude': 51.4700,
       'Longitude': -0.4543}
EWR = {'Flight No.' : 'GS_EWR',
       'Timestamp' : 1530277200,
       'Altitude': 8.72,
       'Latitude': 40.6895,
       'Longitude': -74.1745}

#Append GS Airports to dataframe
df = df.append(LHR, ignore_index=True)
df = df.append(EWR, ignore_index=True)
```

In [3]:

```python
#####-----------------------------------------#####
#---- Data importing and fixing: Step 3       ----#
#####-----------------------------------------#####

#Create constants
Re = 6371000 #Radius of the Earth (meters)
mf = 0.3048   #1 Foot = 0.3048 (meters)

#FUNCTION Polar Coordinates to Cartesian coordinates
def coordinates(obj):
    #Altitude (L), Latitude (theta), Longitude (psi)
    L = obj.Altitude
    theta = obj.Latitude
    psi = obj.Longitude

    #Functions for X (Px), Y(Py), Z(Pz)
    Px = (Re + L*mf)*math.cos(math.radians(theta))*math.cos(math.radians(psi))
    Py = (Re + L*mf)*math.cos(math.radians(theta))*math.sin(math.radians(psi))
    Pz = (Re + L*mf)*math.sin(math.radians(theta))
    return [Px, Py, Pz]

#####-----------------------------------------#####
#---- Data importing and fixing: Step 4       ----#
#####-----------------------------------------#####

#Create new temporal array result_pc and use coordinates function to fill
#result_pc with every Cartesian coordinates
result_pc = []
for i in range(len(df)):
    t = coordinates(df.iloc[i])
    result_pc.append(t)

#Create temporal DataFrame df_pc with result_pc
df_pc = pd.DataFrame(result_pc, columns = ['Px', 'Py', 'Pz'])

#Append new columns to dataframe DF_RES
df_res = pd.concat([df , df_pc], axis="columns")
df_res
```

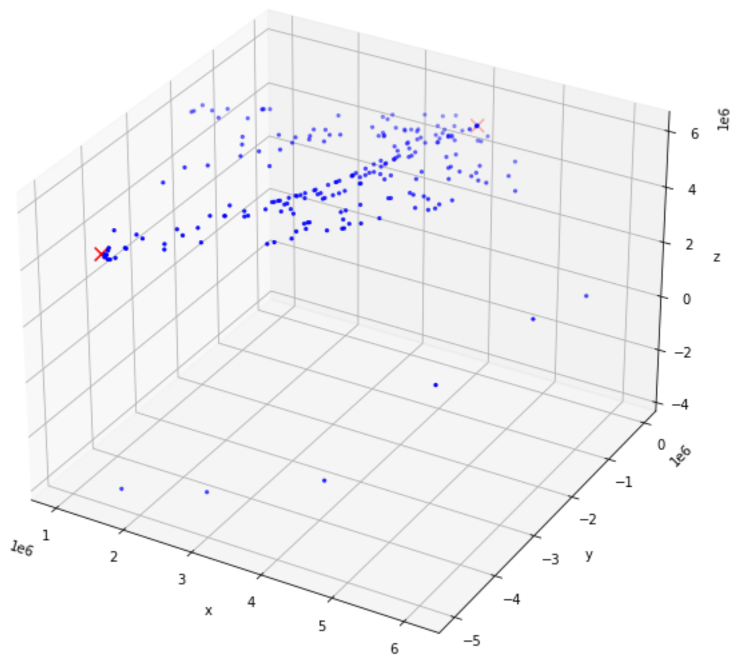
Out[3]:

	Flight No.	Timestamp	Altitude	Latitude	Longitude	Px	Py	Pz
0	AA101	1530277200	39000.00	50.9000	-38.7000	3.141648e+06	-2.516935e+06	4.953417e+06
1	AA109	1530277200	33000.00	60.3000	-12.2000	3.090150e+06	-6.681141e+05	5.542788e+06
2	AA111	1530277200	39000.00	52.7000	-18.1000	3.676553e+06	-1.201683e+06	5.077417e+06
3	AA113	1530277200	37000.00	43.0000	-11.1000	4.580382e+06	-8.986352e+05	4.352703e+06
4	AA151	1530277200	36400.00	47.0000	-27.7000	3.853745e+06	-2.023261e+06	4.667569e+06
...	...	...	...	...	...	...	...	...
213	UA975	1530277200	36000.00	50.5000	-26.4000	3.636083e+06	-1.804967e+06	4.924487e+06
214	UA986	1530277200	36000.00	60.0000	-32.2000	2.700191e+06	-1.700401e+06	5.526951e+06
215	UA988	1530277200	36100.00	52.7000	-18.8000	3.661090e+06	-1.246337e+06	5.076714e+06
216	GS_LHR	1530277200	81.73	51.4700	-0.4543	3.968542e+06	-3.146735e+04	4.983939e+06
217	GS_EWR	1530277200	8.72	40.6895	-74.1745	1.317410e+06	-4.647733e+06	4.153635e+06

218 rows × 8 columns

In [4]:

```python
#####-----------------------------------------#####
#---- Data importing and fixing: Step 5      ----#
#####-----------------------------------------#####

#Plot ariplanes with 0 and GS with x
#%matplotlib notebook

fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(111, projection='3d')

ax.scatter3D(df_res['Px'][:216], df_res['Py'][:216],df_res['Pz'][:216], c='b', marker='o', s =5)
ax.scatter3D(df_res['Px'][216:218], df_res['Py'][216:218],df_res['Pz'][216:218], c='r', marker='x', s =100)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```



In [5]:

```python
#####-----------------------------------------#####
#---- Data importing and fixing: Step 6      ----#
#####-----------------------------------------#####

#Calculate euclidean distance in every node using scipy.spatial.distance as dist
distance_matrix = dist.squareform(dist.pdist(df_pc, 'euclidean'), force='no', checks=True)
```

Transmission Rate table

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

Distance greater than 740km has 0 Transmission rate

In [6]:

```
#####----------------------------------------#####
#---- Data importing and fixing: Step 7      ----#
#####----------------------------------------#####

#Table transmission rate
tr = pd.DataFrame([['Red',    500000,  31.895],
                   ['Orange', 400000,  43.505],
                   ['Yellow', 300000,  52.857],
                   ['Green',  190000,  63.970],
                   ['Blue',   90000,   77.071],
                   ['Pink',   35000,   93.854],
                   ['Purple', 5560, 119.130]],
                   columns=['Mode_color', 'Switching_threshold', 'Transmission_rate'])
max_tr = 740000

#Create switch case to relate Switching_threshold to Transmission_rate
def switch_tr(dist):
    if dist > tr['Switching_threshold'][0] and dist <= max_tr:
        return tr['Transmission_rate'][0]
    elif dist > tr['Switching_threshold'][1] and dist <= tr['Switching_threshold'][0]:
        return tr['Transmission_rate'][1]
    elif dist > tr['Switching_threshold'][2] and dist <= tr['Switching_threshold'][1]:
        return tr['Transmission_rate'][2]
    elif dist > tr['Switching_threshold'][3] and dist <= tr['Switching_threshold'][2]:
        return tr['Transmission_rate'][3]
    elif dist > tr['Switching_threshold'][4] and dist <= tr['Switching_threshold'][3]:
        return tr['Transmission_rate'][4]
    elif dist > tr['Switching_threshold'][5] and dist <= tr['Switching_threshold'][4]:
        return tr['Transmission_rate'][5]
    elif dist > 0 and dist <= tr['Switching_threshold'][5]:
        return tr['Transmission_rate'][6]
    elif dist > max_tr or dist == 0:
        return 0

#Create function to change Switching_threshold to Transmission_rate
def tr_matrix(matrix):
    row_x = []
    column_x = []
    for row in range(0, matrix.shape[0]):
        column_x = []
        for column in range(0, matrix.shape[1]):
            column_x.append(switch_tr(matrix[row][column]))
        row_x.append(column_x)
    return row_x

#use tr_matrix function
tr_df = []
tr_df = tr_matrix(distance_matrix)
tr_df = np.array(tr_df)
tr_df
```

Out[6]:

```
array([[ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ..., 93.854,  0.   ,  0.   ],
       ...,
       [ 0.   ,  0.   , 93.854, ...,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ]])
```

* * * Now data is ready. Now it will work with the df_res column "Flight No." and with the "Transmission rate" tr_df matrix * * *

3.2. Optimization algorithm: Multi-Output Dijkstra's Algorithm

Dijkstra's Algorith was improved to solve the problem of finding the longest path from a node to either of the two Ground Stations. In understanding the problem, two main problems were encountered:

1. A node can have one or several equal paths (with the same Transmission Rate on each path). To solve this problem, it was designed for each solution that the lists of E2E Transmission rate, previous node and visited, have 2 dimensions. Each possible solution is a fork, so that all possible solutions with the maximum E2E Transmission rate can be delivered.
2. The algorithm can find a Ground Station, but having several forks, the algorithm continues searching for the rest of the paths. A break was performed when the algorithm finds a Ground Station with the maximum E2E Transmission rate.

Figure 1 shows the flowchart of the design created to solve the longest path looking for the maximum E2E Transmission rate and solving the above mentioned problems. The algorithm starts in the green box labeled START and ends in the red termination box. Additionally, the numbers in each red box represent the main parts of the algorithm, which are marked in each part of the functions.

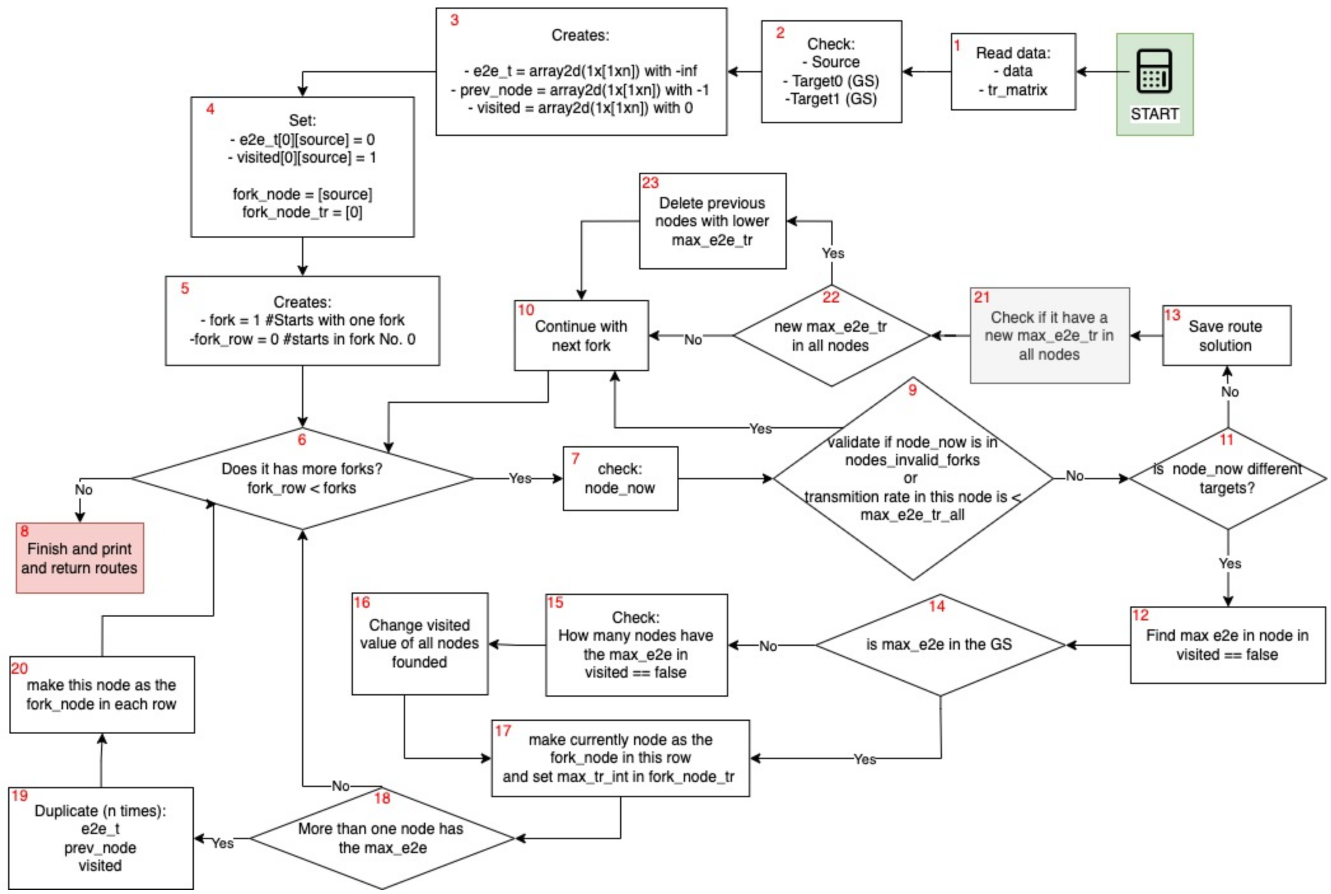


Figure 1. Flowchart - Dijkstra's Algorithm

Using the given set of solutions, carefully evaluate each one and select the solution or solutions that demonstrate the lowest end-to-end latency while still maintaining an acceptable level of performance and functionality.

The output of the algorithm is presented as the example below:

Example output: *[{'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA717', 77.071], ['AA57', 63.97], ['AA198', 31.895], ['GS_EWR', 77.071]], 'End-to-end rate': 31.895, 'End-to-end latency': 150},...]*

In [7]:

```python
#####---------------------------------------#####
#---- Functions                           ----#
#####---------------------------------------#####

#-Class Dijkstras: Evaluates longest
#routh path using transmission rate
class Dijkstras:

    #***---  1 ---***#
    #init function: Read data
    def __init__(self, data, tr_matrix):
        self.data = data
        self.tr_matrix = tr_matrix

    #***---  2 ---***#
    #longest transmission rate function
    #source = Airplane  --  target0 and target1 = Ground Station
    def longest_tr(self, source, target0, target1):

        #***---  3 ---***#
        n = len(self.tr_matrix[source]) #length matrix

        #Create E2E Transimition rate 2D array (e2e_tr) and set all values with -inf
        e2e_tr = np.full(n, (-np.inf))
        e2e_tr = np.array([e2e_tr],).tolist()

        #Create Previous Node 2D array (prev_node) and set all values with None (uses -1 as none)
        prev_node = np.full(n, -1)
        prev_node = np.array([prev_node],).tolist()

        #Create Visited 2D array (visited) and set all values with false (uses 0 as False)
        visited = np.full(n, 0)
        visited = np.array([visited],).tolist()

        #***---  4 ---***#
        #Set source values as:
        #  - e2e_tr in 0 and
        #  - visited as True
        e2e_tr[0][source] = 0
        visited[0][source] = 1

        #***---  5 ---***#
        forks = 1 #Starts with only one fork
        fork_row = 0 #Starts in fork number 0

        fork_node = [source] #Creates Array. First fork_node value in source position
        fork_node_tr = [-np.inf] #Creates Array. It valids fork_node transmission rate

        all_solutions = [] #Array to save all solutions
        valid_solutions = [] #Array to save only valid solutions

        #Array to save invalid nodes [from,to]
        #Invalid forks are:
        #max_tr_int < max_e2e_tr_all or (max_tr_int == -np.inf and forks > 1)
        invalid_nodes = []

        max_e2e_tr_all = -np.inf #Set max e2e transmission rate as -inf

        #***---  6 ---***#
        #To treat each fork_row
        while fork_row < forks:

            #***---  7 ---***#
            node_now = fork_node[fork_row] #Currently node checked

            #print('fork', fork_row, 'node', node_now)

            #Variable to check if node_now founds a GS
            #Set with idx of node founded
            #-1 means None
            found_gs = -1

            #***---  9 ---***#
            #Checks if prev_node and node_now is in invalid_nodes or fork_node_tr < max_e2e_tr
            #If true, change to next fork_row because already exists other route with a better tr
            if ([prev_node[fork_row][node_now],node_now] in invalid_nodes or
                fork_node_tr[fork_row] < max_e2e_tr_all
               ):
                #***---  10 ---***#
                fork_row = fork_row + 1

            else:
                #***---  11 ---***#
                if node_now != target0 and node_now != target1: #Check if node_now is different to any target

                    max_tr_int = (-np.inf) #Set max transmission rate intern as -inf
```

```python

#***---  12 ---***#
#To treat each e2e_tr and check max number
#Condition: Only visited false
for idx, i in enumerate(e2e_tr[fork_row]):
    if visited[fork_row][idx] == 0:

        #Condition1: Value in tr_matrix > cero
        #Condition2: Value in tr_matrix > e2e_tr in this position
        if (
            self.tr_matrix[node_now][idx] > 0 and
            self.tr_matrix[node_now][idx] > e2e_tr[fork_row][idx]
        ):
            #Set new e2e_tr in this position
            #Set prev_node as node_now
            e2e_tr[fork_row][idx] = self.tr_matrix[node_now][idx]
            prev_node[fork_row][idx] = node_now

            #Check if this index is different to any target
            if idx == target0 or idx == target1:
                found_gs = idx

            ###ESTA ES LA QUE DEBO MOVER PARA ATRÁS Y ARREGLAR EL CÓDIGO
        #To update max_tr_int in this fork_row
        if e2e_tr[fork_row][idx] > max_tr_int and e2e_tr[fork_row][idx] > 0:
            max_tr_int = e2e_tr[fork_row][idx]
            #print('neww')

#Condition: Node has a max_tr_int == inf and it is the first fork
#It means every possible answere is in cero
if max_tr_int == -np.inf and forks == 1:
    break

tot_max_tr_int = 0 #To check how many visited false has this max_tr_int
arr_temp_node_idx_max_tr_int = [] #Array to add all idx with the max_tr_int

#***---  14 ---***#
#If max_tr_int is with a target
#Just create one fork
#To avoid unimportant cycles
if found_gs != -1:
    max_tr_int = e2e_tr[fork_row][found_gs]
    arr_temp_node_idx_max_tr_int.append(found_gs)
    tot_max_tr_int = 1

##-> This else ===>> if found_gs == -1:
else:
    #print('max: ', max_tr_int)
    #print('entraaaaaaaa')
    #***---  15 ---***#
    #To treat each e2e_tr and
    #check how many max_tr_int are in this node
    axx = 0
    for idx, i in enumerate(e2e_tr[fork_row]):
        if visited[fork_row][idx] == 0 and e2e_tr[fork_row][idx] == max_tr_int:
            tot_max_tr_int = tot_max_tr_int + 1
            #print('entraxxxxx')

            #append idx arr_temp_node_idx_max_tr_int
            arr_temp_node_idx_max_tr_int.append(idx)
            axx = 1




#Check and append invalid nodes to invalid_nodes
if (
    max_tr_int < max_e2e_tr_all or
    (max_tr_int == -np.inf and forks > 1)
):
    invalid_nodes.append([prev_node[fork_row][node_now],node_now])
    fork_row = fork_row + 1

##-> This else ===>> if max_tr_int >= max_e2e_tr_all:
else:

    #***---  16 ---***#
    #Set visited True in the every position of arr_temp_node_idx_max_tr_int
    #arr_temp_node_idx_max_tr_int[0] is the idx with max_tr_int  found in
    #e2e_tr[fork_row]

    #print(tot_max_tr_int)

    for i in range(tot_max_tr_int):
        visited[fork_row][arr_temp_node_idx_max_tr_int[i]] = 1
        #print('aaaa')
```

```
                            #***---  17 ---***#
                            #Set fork_node with the corresponfing idx
                            #and fork_node_tr with the max_tr_int
                            fork_node[fork_row] = arr_temp_node_idx_max_tr_int[0]
                            fork_node_tr[fork_row] = max_tr_int

                        #print('tot', tot_max_tr_int == 0)
                        #***---  18 ---***#
                        #More than one tot_max_tr_int
                        #It means is necessary duplicate a fork
                        #if tot_max_tr_int > 1:
                        if tot_max_tr_int > 1:

                            #Increase forks with tot_max_tr_int - 1
                            # - 1 because it already has the current fork
                            forks = forks + tot_max_tr_int - 1

                            #***---  19 ---***#
                            #Duplicate tot_max_tr_int - 1 times e2e_tr, prev_node and, visited in this node
                            e2e_tr_temp = np.tile(e2e_tr[fork_row], (tot_max_tr_int - 1, 1)).tolist()
                            prev_node_temp = np.tile(prev_node[fork_row], (tot_max_tr_int - 1, 1)).tolist()
                            visited_temp = np.tile(visited[fork_row], (tot_max_tr_int - 1, 1)).tolist()

                            #Append each duplicate in each matrix
                            # - 1 because it already has the current fork
                            for i in range(tot_max_tr_int - 1):
                                e2e_tr.append(e2e_tr_temp[i])
                                prev_node.append(prev_node_temp[i])
                                visited.append(visited_temp[i])

                            #***---  20 ---***#
                            #Append in fork_node with each the corresponfing idx
                            #and fork_node_tr with the max_tr_int
                            for i in range(tot_max_tr_int - 1):
                                fork_node.append(arr_temp_node_idx_max_tr_int[i+1])
                                fork_node_tr.append(max_tr_int)

                #To save and print routes
                ##-> This else ===>> if node_now == target0 or node_now == target1:
                else:

                    #Create a temporal last max_e2e_tr_all
                    last_max_e2e_tr_all = max_e2e_tr_all

                    #***---  13 ---***#
                    #Use function print_node_route
                    route = print_node_route(E = e2e_tr[fork_row],
                                             P = prev_node[fork_row],
                                             V = visited[fork_row],
                                             source = source,
                                             target = fork_node[fork_row],
                                             data = self.data)

                    #Get min E2E tr in route
                    min_E2E = min(np.array(route)[:,1].astype(float))

                    #***---  21 ---***#
                    #Verify if min_E2E is greater than the last max_e2e_tr_all
                    if min_E2E.astype(float) > max_e2e_tr_all:
                        #Update max_e2e_tr_all
                        max_e2e_tr_all = min_E2E.astype(float)

                    #Get E2E latency in route
                    latency = len(np.array(route)[:,1].astype(float))*50

                    #Create json with route path
                    json = {"source": self.data[source],
                            "routing path": route,
                            "End-to-end rate": min_E2E,
                            "End-to-end latency": latency }

                    #Because if not, the solution is not optimized
                    if min_E2E.astype(float) == max_e2e_tr_all:
                        #Add json to all_solutions array
                        all_solutions.append(json)

                    #Update fork_node_tr with the min_E2E
                    fork_node_tr[fork_row] = min_E2E

                    #***---  22 ---***#
                    #Validate how many forks should be deleted if
                    #It has a new max_e2e_tr_all
                    if last_max_e2e_tr_all < max_e2e_tr_all:
                        i_idx = 0

```

```
                        #To treat each fork
                        while i_idx < len(fork_node_tr):
                            if fork_node_tr[i_idx] < max_e2e_tr_all:

                                #Delete each row of each array
                                #Note: When create temporal arrays, it uses less
                                #self memory, then it runs faster
                                e2e_tr_temp2 =  np.delete(e2e_tr, i_idx, 0)
                                prev_node_temp2 =  np.delete(prev_node, i_idx, 0)
                                visited_temp2 =  np.delete(visited, i_idx, 0)
                                fork_node_temp2 = np.delete(fork_node, i_idx)
                                fork_node_tr_temp2 = np.delete(fork_node_tr, i_idx)

                                e2e_tr = e2e_tr_temp2
                                prev_node = prev_node_temp2
                                visited = visited_temp2
                                fork_node = fork_node_temp2
                                fork_node_tr = fork_node_tr_temp2

                                forks = forks - 1
                                fork_row = fork_row -1

                            else:
                                i_idx = i_idx + 1

                        #fork_row can't be negative
                        #then it should starts in 0 again
                        if fork_row < 0:
                            fork_row = 0

                        #To validate all arrays are a lists
                        if type(e2e_tr) != list:
                            e2e_tr = e2e_tr.tolist()
                            prev_node = prev_node.tolist()
                            visited = visited.tolist()
                            fork_node = fork_node.tolist()
                            fork_node_tr = fork_node_tr.tolist()

                    #***---  10 ---***#
                    #Continue with next node
                    fork_row = fork_row + 1

            #valid_solutions uses max_e2e function
            valid_solutions = max_e2e_tr_fn(solutions = all_solutions, e2e = max_e2e_tr_all)

            #To show n-2 solutions or to stop if all_solutions are greater than n*4 because some
            #forks are repeated multiple times or the fork_row is greater than n*5 to stop cycles
            if len(valid_solutions) >= n-2 or len(all_solutions) > n*4 or fork_row > n*5:
                break

        #Array with solutions with min e2e latency
        solutions_min_latency = []

        #If no valid solutions, then save No Connection routing path
        if len(valid_solutions) == 0:
            valid_solutions = ([{"source": self.data[source],
                                "routing path": [["No conection"]],
                                "End-to-end rate": None,
                                "End-to-end latency": None }])
            len_valid_solutions = 1
            len_solutions_min_latency = 1
            solutions_min_latency = valid_solutions
        else:
            len_valid_solutions = len(valid_solutions)
            #Get solutions with min e2e latency
            solutions_min_latency = min_e2e_latency_fn(valid_solutions)
            len_solutions_min_latency = len(solutions_min_latency)

        #To see the source and how many optimal solutions it has
        print('source:', source,
              'n_solutions:', len_valid_solutions,
              'n_solutions_min_latency:', len_solutions_min_latency )

        #***---  8 ---***#
        return [valid_solutions, len_valid_solutions, solutions_min_latency, len_solutions_min_latency]

#Print route
def print_node_route(E,P,V, source, target, data):
    #Set route_path and prev_node_route_path from target
    #because it checks the route from back to front
    route_path = [[data[target],E[target]]]
    prev_node_route_path = P[target]

    print_nodes = True

    #Because it could not have any Connection
```

```python
    if (prev_node_route_path == source or prev_node_route_path == -1):
        print_nodes = False

    #To insert each route to routing path
    while(print_nodes):
        route_path.insert(0, [data[prev_node_route_path],E[prev_node_route_path]])
        prev_node_route_path = P[prev_node_route_path]

        #When arrives to the source it should break the while
        if (prev_node_route_path == source or prev_node_route_path == -1):
            print_nodes = False

    return route_path

#Verify maximum e2e transmission rate function
#Also check any repeated solution and delete it
def max_e2e_tr_fn(solutions, e2e):

    #Result is an array of all non repeted solutions
    result = []

    #Verify max e2e transmission rate in all solutions
    for i in range(len(solutions)):
        if solutions[i]['End-to-end rate'].astype(float) == e2e:
            result.append(solutions[i])

    #Check repeted solutions and delete it
    not_repeated = [i for n, i in enumerate(result) if i not in result[n + 1:]]

    return not_repeated

#Get minimum e2e latency function
def min_e2e_latency_fn(solutions):

    #Result is an array of all non repeted solutions
    result = []

    min_e2e_latency_all = min(item['End-to-end latency'] for item in solutions)

    #Verify max e2e transmission rate in all solutions
    for i in range(len(solutions)):
        if solutions[i]['End-to-end latency'] == min_e2e_latency_all:
            result.append(solutions[i])

    return result

#Run Dijkstras alghorithm "range_data" number of nodes
def run_test_dijkstra(range_data, data, tr_matrix, target0, target1):

    solutions_json = [] #Array to save solutions
    times = [] #Times it takes each node to use the function
    n_solutions = []
    solutions_latency_json = []
    n_solutions_latency_json = []

    #To treat "range_data" number of nodes
    for i in range(0,range_data):

        # get the start time
        st = time.time()
        #Save solutions
        run_function = Dijkstras(data = data, tr_matrix = tr_matrix
                                ).longest_tr(source = i, target0 = target0, target1 = target1)
        solutions_json.append(run_function[0])
        n_solutions.append(run_function[1])
        solutions_latency_json.append(run_function[2])
        n_solutions_latency_json.append(run_function[3])

        # get the end time
        et = time.time()
        times.append(et - st)

    return [solutions_json, n_solutions, solutions_latency_json, n_solutions_latency_json, times]

```

3.3. Optimization algorithm: BFS Algorithm

Finding the shortest path through a matrix can be done effectively using the Breadth-First Search (BFS) algorithm. The technique, which is available on the geekforgeeks website (https://www.geeksforgeeks.org/building-an-undirected-graph-and-finding-shortest-path-using-dictionaries-in-python/ (https://www.geeksforgeeks.org/building-an-undirected-graph-and-finding-shortest-path-using-dictionaries-in-python/)), moves through the matrix breadth-first, stopping at each node at a particular level before moving on to the next level. Regardless of the weights at each node, this enables it to determine the minimal end-to-end latency for each node.

Finding the shortest path in a matrix is one of the primary benefits of the BFS algorithm, which makes it beneficial for a range of issues relating to network latency and routing. It's crucial to keep in mind that this approach will identify the lowest end-to-end latency regardless of the end-to-end transmission rate because it does not take into account the weights of each node. Furthermore, BFS only displays the first optimal outcome.

The code was modified in order to compare it to the Dijkstra's Algorithm solution. This makes it simpler to compare the two methods and decide which one is better suited for a given task because the results can be presented in the same way.

In [8]:

```python

# Function to find the shortest
# path between two nodes of a graph
def BFS_SP(data, matrix, source, target0, target1 ):

    graph = nx.from_numpy_matrix(np.matrix(matrix), create_using=nx.DiGraph)

    explored = []
    weights = []

    # Queue for traversing the
    # graph in the BFS
    queue = [[source]]

    # If the desired node is
    # reached
    if source == target0 or source == target1:
        return

    # Loop to traverse the graph
    # with the help of the queue
    while queue:
        path = queue.pop(0)
        node = path[-1]

        # Condition to check if the
        # current node is not visited
        if node not in explored:
            neighbours = graph[node]

            # Loop to iterate over the
            # neighbours of the node
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)

                # Condition to check if the
                # neighbour node is the goal
                if neighbour == target0 or neighbour == target1:

                    route = print_node_route_BFS(new_path,data,matrix)

                    min_E2E = min(np.array(route)[:,1].astype(float))

                    #Get E2E latency in route
                    latency = len(np.array(route)[:,1].astype(float))*50

                    solution = ([{"source": data[source],
                                "routing path": route,
                                "End-to-end rate": min_E2E,
                                "End-to-end latency": latency }])

                    return solution

            explored.append(node)

    solution = ([{"source": data[source],
                "routing path": [["No conection"]],
                "End-to-end rate": None,
                "End-to-end latency": None }])
    return solution

#Print route
def print_node_route_BFS(path, data, matrix):

    if path != None :
        node = len(path)
        print_nodes = True
        route_path = []

        while print_nodes == True:
            route_path.insert(0, [data[path[node-1]], matrix[path[node-1]][path[node-2]]])
            node = node - 1
            if node <= 1:
                print_nodes = False
        return route_path

#Run Dijkstras alghorithm "range_data" number of nodes
def run_test_BFS(range_data, data, matrix, target0, target1):

    solutions_json = [] #Array to save solutions
    times = [] #Times it takes each node to use the function

    #To treat "range_data" number of nodes
    for i in range(0,range_data):
```

```
        # get the start time
        st = time.time()
        #Save solutions
        run_function = BFS_SP(data = data, matrix = matrix, source = i, target0 = target0, target1 = target1)
        solutions_json.append(run_function[0])
        # get the end time
        et = time.time()
        times.append(et - st)

    return [solutions_json, times]
```

4. Experiments

4.1 Testing Multi-Output Dijkstra's Algorithm: Made-up Examples

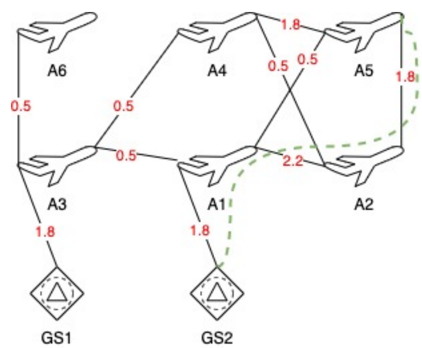
Several were performed using small examples to test the efficiency of the algorithm.

The three main tests are listed below.

4.1.1. Test 1: Single Output - From A5 to any Ground Station

In this test, the Single Output is tested, from node A5 to any Ground Station. It is called Single Output, because it only has a single optimal route with the maximum E2E Transmission Rate.

The green dashed line represents the optimized route.



In [9]:

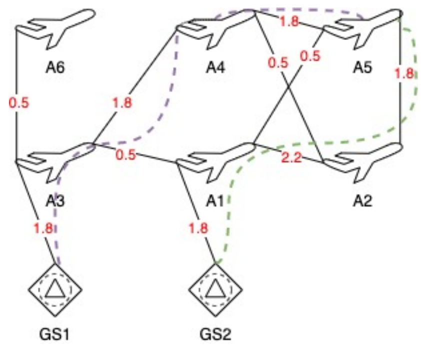
```
#Test 1.1. - Data names
data1 = np.array(['A1','A2','A3','A4','A5','A6','GS1','GS2'])

#Test TR Matrix           [A1 ,A2 ,A3 ,A4 ,A5 ,A6 ,GS1,GS2]
tr_matrix1 = np.array((  [0  ,2.2,0.5,0  ,0.5,0  ,0  ,1.8],
                         [2.2,0  ,0  ,0.5,1.8,0  ,0  ,0  ],
                         [0.5,0  ,0  ,0.5,0  ,0.5,1.8,0  ],
                         [0  ,0.5,0.5,0  ,1.8,0  ,0  ,0  ],
                         [0.5,1.8,0  ,1.8,0  ,0  ,0  ,0  ],
                         [0  ,0  ,0.5,0  ,0  ,0  ,0  ,0  ],
                         [0  ,0  ,1.8,0  ,0  ,0  ,0  ,0  ],
                         [1.8,0  ,0  ,0  ,0  ,0  ,0  ,0  ]))

#source = 4 (A5) -- target0 = 6 (GS1) -- target1 = 7 (GS2)
Test1_1 = Dijkstras(data = data1, tr_matrix = tr_matrix1
                   ).longest_tr(source = 4, target0 = 6, target1 = 7)

print("\033[1m Solutions with maximun E2E TR: \033[0m")
print(Test1_1[0])
print("\033[1m Solutions with maximum E2E TR and minimum E2E Latency:  \033[0m")
print(Test1_1[2])
```

```
source: 4 n_solutions: 1 n_solutions_min_latency: 1
 Solutions with maximun E2E TR:
[{'source': 'A5', 'routing path': [['A2', 1.8], ['A1', 2.2], ['GS2', 1.8]], 'End-to-end rate': 1.8, 'End-to-end laten
cy': 150}]
 Solutions with maximum E2E TR and minimum E2E Latency:
[{'source': 'A5', 'routing path': [['A2', 1.8], ['A1', 2.2], ['GS2', 1.8]], 'End-to-end rate': 1.8, 'End-to-end laten
cy': 150}]
```

4.1.2. Test 2: Multi-output - From A5 to any Ground Station

In this test, the Multiple Output is tested, from node A5 to any Ground Station. It is called Multiple Output, because it has more than one optimal route with the maximum E2E Transmission Rate. In this case, the expected output is 2 routes.

The green and purple dashed lines represent the optimized routes.

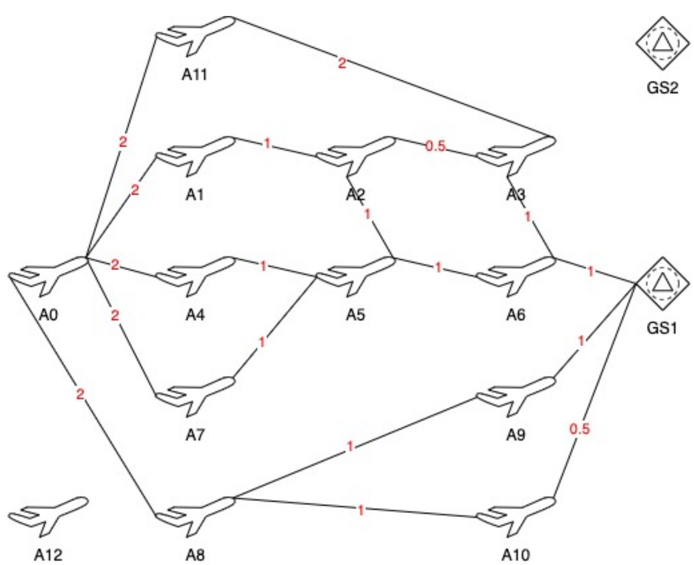



```
In [10]:
#Test 1.2. -> Data names
data2 = np.array(['A1','A2','A3','A4','A5','A6','GS1','GS2'])

#Test TR Matrix          [A1 ,A2 ,A3 ,A4 ,A5 ,A6 ,GS1,GS2]
tr_matrix2 = np.array((  [0  ,2.2,0.5,0  ,0.5,0  ,0  ,1.8],
                         [2.2,0  ,0  ,1.8,1.8,0  ,0  ,0  ],
                         [0.5,0  ,0  ,1.8,0  ,0.5,1.8,0  ],
                         [0  ,1.8,1.8,0  ,1.8,0  ,0  ,0  ],
                         [0.5,1.8,0  ,1.8,0  ,0  ,0  ,0  ],
                         [0  ,0  ,0.5,0  ,0  ,0  ,0  ,0  ],
                         [0  ,0  ,1.8,0  ,0  ,0  ,0  ,0  ],
                         [1.8,0  ,0  ,0  ,0  ,0  ,0  ,0  ]))

#source = 4 (A5) -- target0 = 6 (GS1) -- target1 = 7 (GS2)
Test1_2 = Dijkstras(data = data2, tr_matrix = tr_matrix2
                   ).longest_tr(source = 4, target0 = 6, target1 = 7)

print("\033[1m Solutions with maximun E2E TR: \033[0m")
print(Test1_2[0])
print("\033[1m Solutions with maximum E2E TR and minimum E2E Latency:  \033[0m")
print(Test1_2[2])
```

```
source: 4 n_solutions: 2 n_solutions_min_latency: 2
 Solutions with maximun E2E TR:
[{'source': 'A5', 'routing path': [['A2', 1.8], ['A1', 2.2], ['GS2', 1.8]], 'End-to-end rate': 1.8, 'End-to-end laten
cy': 150}, {'source': 'A5', 'routing path': [['A4', 1.8], ['A3', 1.8], ['GS1', 1.8]], 'End-to-end rate': 1.8, 'End-to
-end latency': 150}]
 Solutions with maximum E2E TR and minimum E2E Latency:
[{'source': 'A5', 'routing path': [['A2', 1.8], ['A1', 2.2], ['GS2', 1.8]], 'End-to-end rate': 1.8, 'End-to-end laten
cy': 150}, {'source': 'A5', 'routing path': [['A4', 1.8], ['A3', 1.8], ['GS1', 1.8]], 'End-to-end rate': 1.8, 'End-to
-end latency': 150}]
```

4.1.3. Test 3: All routes for every airplane

In this test, the routes for each of the nodes to any Ground Station are checked. The function run_test_dijkstra is used to run a path for all nodes except Ground Station.



In [11]:

```python
#Test 1.3. - Data names
data3 = np.array(['A0','A1','A2','A3','A4','A5','A6','A7','A8','A9','A10','A11','A12','GS1', 'GS2'])

#Test TR Matrix           [A0 ,A1 ,A2 ,A3 ,A4 ,A5 ,A6 ,A7 ,A8 ,A9 ,A10,A11,A12,GS1,GS2]
tr_matrix3 = np.array((  [0  ,2  ,0  ,0  ,2  ,0  ,0  ,2  ,2  ,0  ,0  ,2  ,0  ,0  ,0  ],
                          [2  ,0  ,1  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [0  ,1  ,0  ,0.5,0  ,1  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [0  ,0  ,0.5,0  ,0  ,0  ,1  ,0  ,0  ,0  ,0  ,2  ,0  ,0  ,0  ],
                          [2  ,0  ,0  ,0  ,0  ,1  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [0  ,0  ,1  ,0  ,1  ,0  ,1  ,1  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [0  ,0  ,0  ,1  ,0  ,1  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,1  ,0  ],
                          [2  ,0  ,0  ,0  ,0  ,1  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [2  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,1  ,1  ,0  ,0  ,0  ,0  ],
                          [0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,1  ,0  ,0  ,0  ,0  ,1  ,0  ],
                          [0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,1  ,0  ,0  ,0  ,0  ,0.5,0  ],
                          [2  ,0  ,0  ,2  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],
                          [0  ,0  ,0  ,0  ,0  ,0  ,1  ,0  ,0  ,1  ,0.5,0  ,0  ,0  ,0  ],
                          [0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ],))

#Check all routes
#source = all -- target0 = 13 (GS1) -- target1 = 14 (GS2)
Test1_3 = run_test_dijkstra(len(data3)-2, data3, tr_matrix3, 13, 14)

print("\033[1m Solutions with maximun E2E TR: \033[0m")
print(Test1_3[0])
print("\033[1m Solutions with maximum E2E TR and minimum E2E Latency:  \033[0m")
print(Test1_3[2])
```

```
source: 0 n_solutions: 5 n_solutions_min_latency: 1
source: 1 n_solutions: 5 n_solutions_min_latency: 2
source: 2 n_solutions: 5 n_solutions_min_latency: 1
source: 3 n_solutions: 2 n_solutions_min_latency: 1
source: 4 n_solutions: 3 n_solutions_min_latency: 1
source: 5 n_solutions: 4 n_solutions_min_latency: 1
source: 6 n_solutions: 1 n_solutions_min_latency: 1
source: 7 n_solutions: 3 n_solutions_min_latency: 1
source: 8 n_solutions: 5 n_solutions_min_latency: 1
source: 9 n_solutions: 1 n_solutions_min_latency: 1
source: 10 n_solutions: 1 n_solutions_min_latency: 1
source: 11 n_solutions: 5 n_solutions_min_latency: 1
source: 12 n_solutions: 1 n_solutions_min_latency: 1
 Solutions with maximun E2E TR:
[[{'source': 'A0', 'routing path': [['A1', 2.0], ['A2', 1.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end ra
te': 1.0, 'End-to-end latency': 250}, {'source': 'A0', 'routing path': [['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End
-to-end rate': 1.0, 'End-to-end latency': 150}, {'source': 'A0', 'routing path': [['A11', 2.0], ['A3', 2.0], ['A6',
1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 200}, {'source': 'A0', 'routing path': [['A4', 2.
0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 200}, {'source': 'A0', 'ro
uting path': [['A7', 2.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 20
0}], [{'source': 'A1', 'routing path': [['A2', 1.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0,
 'End-to-end latency': 200}, {'source': 'A1', 'routing path': [['A0', 2.0], ['A4', 2.0], ['A5', 1.0], ['A6', 1.0], ['G
S1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}, {'source': 'A1', 'routing path': [['A0', 2.0], ['A7',
2.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}, {'source': 'A1',
'routing path': [['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency':
200}, {'source': 'A1', 'routing path': [['A0', 2.0], ['A11', 2.0], ['A3', 2.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-e
nd rate': 1.0, 'End-to-end latency': 250}], [{'source': 'A2', 'routing path': [['A5', 1.0], ['A4', 1.0], ['A0', 2.0],
['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 300}, {'source': 'A2', 'routin
g path': [['A1', 1.0], ['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end late
ncy': 250}, {'source': 'A2', 'routing path': [['A1', 1.0], ['A0', 2.0], ['A11', 2.0], ['A3', 2.0], ['A6', 1.0], ['GS
1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 300}, {'source': 'A2', 'routing path': [['A5', 1.0], ['A6',
1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 150}, {'source': 'A2', 'routing path': [['A5', 1.
0], ['A7', 1.0], ['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency':
300}], [{'source': 'A3', 'routing path': [['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 1
00}, {'source': 'A3', 'routing path': [['A11', 2.0], ['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-en
d rate': 1.0, 'End-to-end latency': 250}], [{'source': 'A4', 'routing path': [['A0', 2.0], ['A8', 2.0], ['A9', 1.0],
['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 200}, {'source': 'A4', 'routing path': [['A5', 1.0], ['A
6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 150}, {'source': 'A4', 'routing path': [['A0',
2.0], ['A11', 2.0], ['A3', 2.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}],
[{'source': 'A5', 'routing path': [['A2', 1.0], ['A1', 1.0], ['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'E
nd-to-end rate': 1.0, 'End-to-end latency': 300}, {'source': 'A5', 'routing path': [['A6', 1.0], ['GS1', 1.0]], 'End-
to-end rate': 1.0, 'End-to-end latency': 100}, {'source': 'A5', 'routing path': [['A4', 1.0], ['A0', 2.0], ['A8', 2.
0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}, {'source': 'A5', 'routing path':
[['A7', 1.0], ['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 25
0}], [{'source': 'A6', 'routing path': [['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 50}], [{'sourc
e': 'A7', 'routing path': [['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end
latency': 200}, {'source': 'A7', 'routing path': [['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'E
nd-to-end latency': 150}, {'source': 'A7', 'routing path': [['A0', 2.0], ['A11', 2.0], ['A3', 2.0], ['A6', 1.0], ['GS
1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}], [{'source': 'A8', 'routing path': [['A0', 2.0], ['A
1', 2.0], ['A2', 1.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 300},
{'source': 'A8', 'routing path': [['A0', 2.0], ['A11', 2.0], ['A3', 2.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rat
e': 1.0, 'End-to-end latency': 250}, {'source': 'A8', 'routing path': [['A0', 2.0], ['A4', 2.0], ['A5', 1.0], ['A6',
1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}, {'source': 'A8', 'routing path': [['A0', 2.
0], ['A7', 2.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}, {'sourc
e': 'A8', 'routing path': [['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 100}], [{'sourc
e': 'A9', 'routing path': [['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 50}], [{'source': 'A10', 'rou
ting path': [['GS1', 0.5]], 'End-to-end rate': 0.5, 'End-to-end latency': 50}], [{'source': 'A11', 'routing path':
[['A0', 2.0], ['A1', 2.0], ['A2', 1.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end
latency': 300}, {'source': 'A11', 'routing path': [['A3', 2.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0,
'End-to-end latency': 150}, {'source': 'A11', 'routing path': [['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]],
'End-to-end rate': 1.0, 'End-to-end latency': 200}, {'source': 'A11', 'routing path': [['A0', 2.0], ['A4', 2.0], ['A
5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 250}, {'source': 'A11', 'routing
path': [['A0', 2.0], ['A7', 2.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latenc
y': 250}], [{'source': 'A12', 'routing path': [['No conection']], 'End-to-end rate': None, 'End-to-end latency': Non
e}]]
 Solutions with maximum E2E TR and minimum E2E Latency:
[[{'source': 'A0', 'routing path': [['A8', 2.0], ['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end late
ncy': 150}], [{'source': 'A1', 'routing path': [['A2', 1.0], ['A5', 1.0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rat
e': 1.0, 'End-to-end latency': 200}, {'source': 'A1', 'routing path': [['A0', 2.0], ['A8', 2.0], ['A9', 1.0], ['GS1',
1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 200}], [{'source': 'A2', 'routing path': [['A5', 1.0], ['A6', 1.
0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 150}], [{'source': 'A3', 'routing path': [['A6', 1.
0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 100}], [{'source': 'A4', 'routing path': [['A5', 1.
0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 150}], [{'source': 'A5', 'routing pat
h': [['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 100}], [{'source': 'A6', 'routing pat
h': [['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 50}], [{'source': 'A7', 'routing path': [['A5', 1.
0], ['A6', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 150}], [{'source': 'A8', 'routing pat
h': [['A9', 1.0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 100}], [{'source': 'A9', 'routing pat
h': [['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 50}], [{'source': 'A10', 'routing path': [['GS1',
0.5]], 'End-to-end rate': 0.5, 'End-to-end latency': 50}], [{'source': 'A11', 'routing path': [['A3', 2.0], ['A6', 1.
0], ['GS1', 1.0]], 'End-to-end rate': 1.0, 'End-to-end latency': 150}], [{'source': 'A12', 'routing path': [['No cone
ction']], 'End-to-end rate': None, 'End-to-end latency': None}]]
```

4.2. Testing Multi-Output Dijkstra's Algorithm: Using real data

The main tests of 3 solutions with real data are shown.

```
Note: To plot the data, it was necessary to follow the steps in the tutorial "Easy Steps To Plot Geographic Dat
a on a Map - Python" from: https://towardsdatascience.com/easy-steps-to-plot-geographic-data-on-a-map-python-11
217859a2db as an alternative due to an error when installing package basemap in Mac M1. This makes the graphs n
ot very accurate.
```

The maps show:

- Green: All nodes
- Red: Ground stations
- Blue: Source and nodes available connected

In [12]:
```python
#Function to plot in map

#Example: data = Test2_2[0]  --  img_map='map2.png'  --  title='Test 2.2. - Plotting Nodes in map'
def plot_in_map(data, img_map, title=""):
    #Get data to plot
    #Get source and nodes
    p_source = data.get("source")
    p_nodes = np.array(data.get("routing path"))[:,0]

    #Create a dataframe with nodes and vLookup with df_res
    arr_plot_x = []
    arr_plot_x = [p_source]

    if p_nodes[0] != 'No conection':
        arr_plot_x = np.append(arr_plot_x,p_nodes)

    df_plot_x = pd.DataFrame(arr_plot_x, columns = ['Flight No.'])
    df_join = pd.merge(df_plot_x, df, on ='Flight No.', how ='inner')

    if p_nodes[0] != 'No conection':
        #Get max and min Longitudes and Latitudes
        Plot_Box = (df_join.Longitude.min(),
                    df_join.Longitude.max(),
                    df_join.Latitude.min(),
                    df_join.Latitude.max())
        len_plot = len(arr_plot_x)-1
    else:
        Plot_Box = (df.Longitude.min(), df.Longitude.max(),df.Latitude.min(), df.Latitude.max())
        len_plot = len(arr_plot_x)

    plt_map = plt.imread(img_map)

    fig, ax = plt.subplots(figsize = (10,10))

    #Blue routing path nodes
    ax.scatter(df_join.Longitude[:len_plot], df_join.Latitude[:len_plot], zorder=1, alpha= 0.9, c='b', s=100)
    plt.plot(df_join.Longitude, df_join.Latitude)

    #Green all nodes
    ax.scatter(df.Longitude[:216], df.Latitude[:216], zorder=1, alpha= 0.5, c='g', s=10)

    #Red Ground Stations
    ax.scatter(df.Longitude[216:218], df.Latitude[216:218], zorder=1, alpha= 1, c='r', s=100, marker = 'x')

    for i, txt in enumerate(arr_plot_x):
        ax.annotate(txt, (df_join.Longitude[i], df_join.Latitude[i]), fontsize=8)

    ax.set_title(title)
    ax.set_xlim(Plot_Box[0]-0.2,Plot_Box[1]+0.2)
    ax.set_ylim(Plot_Box[2]-0.2,Plot_Box[3]+0.2)
    ax.imshow(plt_map, zorder=0, extent = Plot_Box, aspect= 'equal')
```

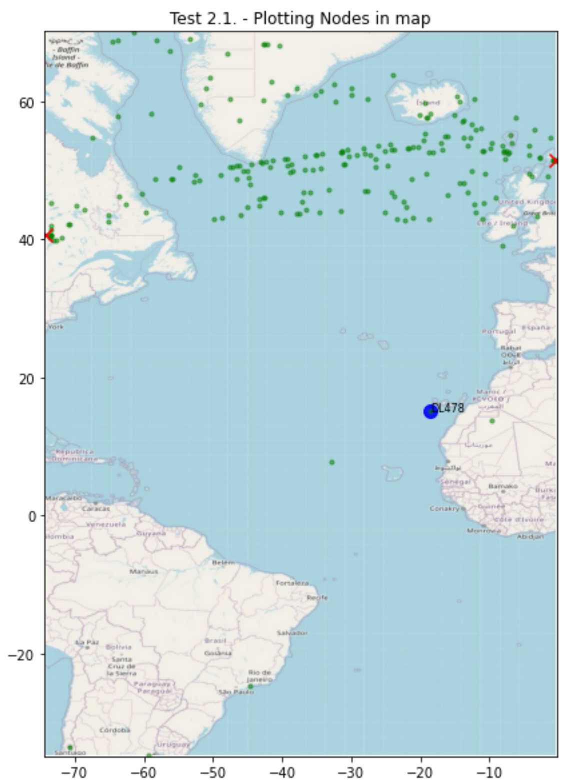
4.2.1. Test 1: No Connection - From BA244 to any Ground Station

The objective of this test is to verify a node when there is no route to any Ground Station.

In [13]:
```python
#Test 2.1. - No Connection
#source = 123 (DL478) -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
Test2_1 = Dijkstras(data = df.iloc[:, 0].to_numpy(), tr_matrix = tr_df
                   ).longest_tr(source = 123, target0 = 216, target1 = 217)

print("\033[1m Solutions with maximun E2E TR: \033[0m")
print(Test2_1[0])
print("\033[1m Solutions with maximum E2E TR and minimum E2E Latency:  \033[0m")
print(Test2_1[2])
```

```
source: 123 n_solutions: 1 n_solutions_min_latency: 1
```
Solutions with maximun E2E TR:
```
[{'source': 'DL478', 'routing path': [['No conection']], 'End-to-end rate': None, 'End-to-end latency': None}]
```
Solutions with maximum E2E TR and minimum E2E Latency:
```
[{'source': 'DL478', 'routing path': [['No conection']], 'End-to-end rate': None, 'End-to-end latency': None}]
```

In [14]:
```python
#data=Test2_2[0]
#imG_map='map2.png'
#title='Test 2.2. - Plotting Nodes in map'
plot_in_map(Test2_1[0][0], 'map.jpg', 'Test 2.1. - Plotting Nodes in map')
```



4.2.2. Test 2: Close to any Ground Station - From AA198 to any Ground Station

In this test, a node very close to a Ground Station is checked and the maximum E2E Transmission rate is displayed.

In [15]:
```python
#Test 2.2. - Close to any Ground Station
#source = 5 (AA198) -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
Test2_2 = Dijkstras(data = df.iloc[:, 0].to_numpy(), tr_matrix = tr_df
                   ).longest_tr(source = 5, target0 = 216, target1 = 217)

print("\033[1m Solutions with maximun E2E TR: \033[0m")
print(Test2_2[0])
print("\033[1m Solutions with maximum E2E TR and minimum E2E Latency:  \033[0m")
print(Test2_2[2])
```

```
source: 5 n_solutions: 1 n_solutions_min_latency: 1
```
Solutions with maximun E2E TR:
```
[{'source': 'AA198', 'routing path': [['GS_EWR', 77.071]], 'End-to-end rate': 77.071, 'End-to-end latency': 50}]
```
Solutions with maximum E2E TR and minimum E2E Latency:
```
[{'source': 'AA198', 'routing path': [['GS_EWR', 77.071]], 'End-to-end rate': 77.071, 'End-to-end latency': 50}]
```

In [16]:
```
plot_in_map(Test2_2[0][0], 'map2.jpg', 'Test 2.2. - Plotting Nodes in map')
```

4.2.3. Test 3: Node with multiple outputs - From AA101 to any Ground Station

The objective of this test is to test a node with Multiple Outputs. The interesting thing about this node is that it can reach the two Ground Stations with the maximum E2E Transmission Rate.

In [17]:

```python
#Far away from GS
#source = 0 (AA101) -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
Test2_3 = Dijkstras(data = df.iloc[:, 0].to_numpy(), tr_matrix = tr_df
                   ).longest_tr(source = 0, target0 =216, target1 =217)

print("\033[1m Solutions with maximun E2E TR: \033[0m")
print(Test2_3[0])
print("\033[1m Solutions with maximum E2E TR and minimum E2E Latency:  \033[0m")
print(Test2_3[2])
```

```
source: 0 n_solutions: 11 n_solutions_min_latency: 1
```

Solutions with maximun E2E TR:

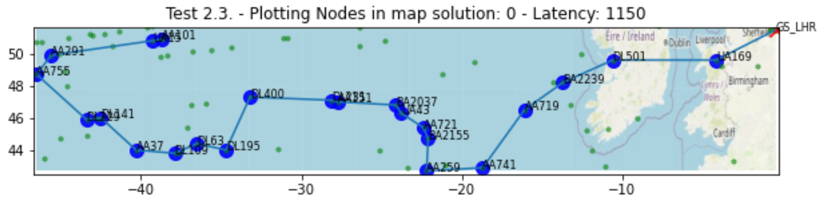

```
[{'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA291', 43.505], ['AA755', 77.071], ['DL229', 52.857], ['DL
141', 93.854], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL195', 77.071], ['DL400', 52.857], ['DL231', 5
2.857], ['AA151', 119.13], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.9
7], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['UA169', 43.505], ['GS_LHR', 52.857]],
'End-to-end rate': 43.505, 'End-to-end latency': 1150}, {'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA71
7', 77.071], ['DL49', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.071], ['BA2167', 77.071], ['DL141', 6
3.97], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['DL400', 63.97], ['AA151', 43.505],
['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.97], ['AA7
19', 43.505], ['DL213', 63.97], ['DL501', 52.857], ['LH418', 52.857], ['GS_LHR', 43.505]], 'End-to-end rate': 43.505,
'End-to-end latency': 1250}, {'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA717', 77.071], ['DL49', 93.85
4], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.071], ['BA2167', 77.071], ['DL141', 63.97], ['AA37', 63.97], ['D
L109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['DL400', 63.97], ['AA151', 43.505], ['BA2037', 63.97], ['UA43', 9
3.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.9
7], ['DL501', 63.97], ['LH418', 52.857], ['GS_LHR', 43.505]], 'End-to-end rate': 43.505, 'End-to-end latency': 1250},
{'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA717', 77.071], ['DL49', 93.854], ['AA25', 93.854], ['BA17
5', 77.071], ['AA45', 77.071], ['BA2167', 77.071], ['DL141', 63.97], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.0
71], ['DL117', 63.97], ['DL400', 63.97], ['AA151', 43.505], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071],
['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['LH8
224', 43.505], ['GS_LHR', 43.505]], 'End-to-end rate': 43.505, 'End-to-end latency': 1250}, {'source': 'AA101', 'rout
ing path': [['UA15', 93.854], ['AA717', 77.071], ['DL49', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.0
71], ['BA2167', 77.071], ['DL141', 63.97], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['D
L400', 63.97], ['AA151', 43.505], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA25
9', 63.97], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['UA169', 43.505], ['GS_LHR', 5
2.857]], 'End-to-end rate': 43.505, 'End-to-end latency': 1250}, {'source': 'AA101', 'routing path': [['UA15', 93.85
4], ['AA717', 77.071], ['DL49', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.071], ['BA2167', 77.071],
['DL141', 63.97], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['DL400', 63.97], ['AA151',
43.505], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.9
7], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['UA31', 43.505], ['GS_LHR', 52.857]], 'End-to-end rate':
43.505, 'End-to-end latency': 1250}, {'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA717', 77.071], ['DL4
9', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.071], ['BA2167', 77.071], ['DL419', 63.97], ['AA37', 6
3.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['DL400', 63.97], ['AA151', 43.505], ['BA2037', 63.97],
['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.97], ['AA719', 43.505], ['DL2
13', 63.97], ['DL501', 52.857], ['LH418', 52.857], ['GS_LHR', 43.505]], 'End-to-end rate': 43.505, 'End-to-end latenc
y': 1250}, {'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA717', 77.071], ['DL49', 93.854], ['AA25', 93.85
4], ['BA175', 77.071], ['AA45', 77.071], ['BA2167', 77.071], ['DL419', 63.97], ['AA37', 63.97], ['DL109', 63.97], ['D
L63', 77.071], ['DL117', 63.97], ['DL400', 63.97], ['AA151', 43.505], ['BA2037', 63.97], ['UA43', 93.854], ['AA721',
77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.9
7], ['LH418', 52.857], ['GS_LHR', 43.505]], 'End-to-end rate': 43.505, 'End-to-end latency': 1250}, {'source': 'AA10
1', 'routing path': [['UA15', 93.854], ['AA717', 77.071], ['DL49', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA
45', 77.071], ['BA2167', 77.071], ['DL419', 63.97], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 6
3.97], ['DL400', 63.97], ['AA151', 43.505], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.85
4], ['AA259', 63.97], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['LH8224', 43.505],
['GS_LHR', 43.505]], 'End-to-end rate': 43.505, 'End-to-end latency': 1250}, {'source': 'AA101', 'routing path': [['U
A15', 93.854], ['AA717', 77.071], ['DL49', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.071], ['BA2167',
77.071], ['DL419', 63.97], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['DL400', 63.97],
['AA151', 43.505], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA
741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['UA169', 43.505], ['GS_LHR', 52.857]], 'End-to
-end rate': 43.505, 'End-to-end latency': 1250}, {'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA717', 77.
071], ['DL49', 93.854], ['AA25', 93.854], ['BA175', 77.071], ['AA45', 77.071], ['BA2167', 77.071], ['DL419', 63.97],
['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL117', 63.97], ['DL400', 63.97], ['AA151', 43.505], ['BA203
7', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.97], ['AA741', 63.97], ['AA719', 4
3.505], ['BA2239', 63.97], ['DL501', 63.97], ['UA31', 43.505], ['GS_LHR', 52.857]], 'End-to-end rate': 43.505, 'End-t
o-end latency': 1250}]
```

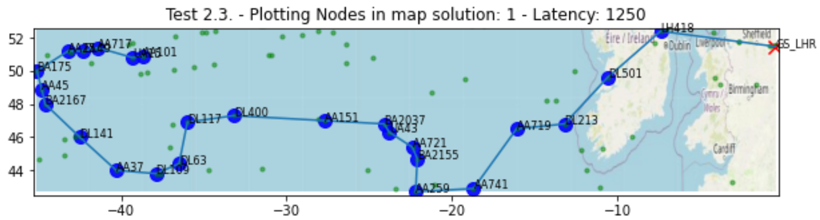

Solutions with maximum E2E TR and minimum E2E Latency:

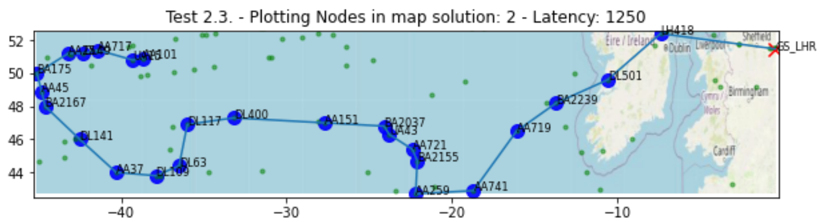

```
[{'source': 'AA101', 'routing path': [['UA15', 93.854], ['AA291', 43.505], ['AA755', 77.071], ['DL229', 52.857], ['DL
141', 93.854], ['AA37', 63.97], ['DL109', 63.97], ['DL63', 77.071], ['DL195', 77.071], ['DL400', 52.857], ['DL231', 5
2.857], ['AA151', 119.13], ['BA2037', 63.97], ['UA43', 93.854], ['AA721', 77.071], ['BA2155', 93.854], ['AA259', 63.9
7], ['AA741', 63.97], ['AA719', 43.505], ['BA2239', 63.97], ['DL501', 63.97], ['UA169', 43.505], ['GS_LHR', 52.857]],
'End-to-end rate': 43.505, 'End-to-end latency': 1150}]
```

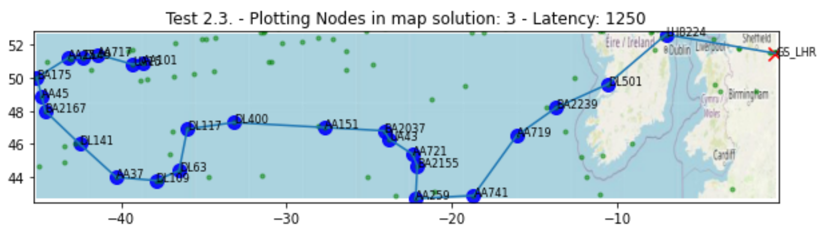
In [18]:

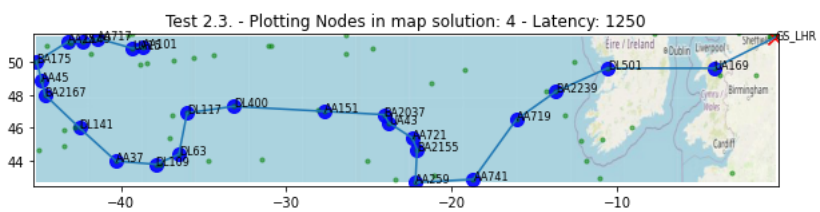
```python
#Print first 5 solutions to compare
for i in range(5):
    plot_in_map(Test2_3[0][i],
                'map3.jpg',
                'Test 2.3. - Plotting Nodes in map solution: ' + str(i) +
                ' - Latency: ' + str(Test2_3[0][i].get('End-to-end latency')))

```





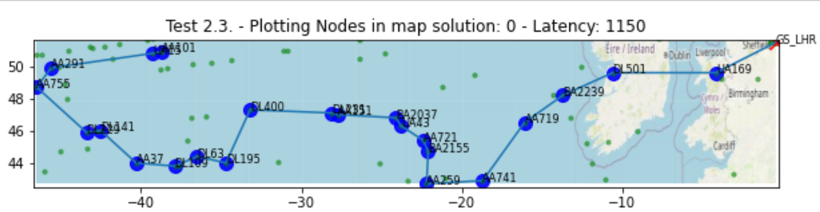






In [19]:

```python
#Print solutions with minimun latency
for i in range(len(Test2_3[2])):
    plot_in_map(Test2_3[2][i],
                'map3.jpg',
                'Test 2.3. - Plotting Nodes in map solution: ' + str(i) +
                ' - Latency: ' + str(Test2_3[0][i].get('End-to-end latency')))

```



4.3. Testing BFS Algorithm: Using real data

The same tests of point 4.2 will be used. to check the output of the algorithm

4.3.1. Test 1: No Connection - From BA244 to any Ground Station

The objective of this test is to verify a node when there is no route to any Ground Station.

In [20]:

```
#Test 3.1. - No Connection
#source = 123 (DL478) -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
Test3_1 = BFS_SP(data = df_res.iloc[:, 0].to_numpy(), matrix = tr_df, source = 123, target0 = 216, target1 = 217)

print("\033[1m Solution with minimum E2E Latency: \033[0m")
print(Test3_1[0])
```

```
 Solution with minimum E2E Latency:
{'source': 'DL478', 'routing path': [['No conection']], 'End-to-end rate': None, 'End-to-end latency': None}
```

4.3.2. Test 2: Close to any Ground Station - From AA198 to any Ground Station

In this test, a node very close to a Ground Station is checked and the maximum E2E Transmission rate is displayed.

In [21]:

```
#Test 3.2. - Close to any Ground Station
#source = 5 (AA198) -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
Test3_2 = BFS_SP(data = df_res.iloc[:, 0].to_numpy(), matrix = tr_df, source = 5, target0 = 216, target1 = 217)

print("\033[1m Solution with minimum E2E Latency: \033[0m")
print(Test3_2[0])
```

```
 Solution with minimum E2E Latency:
{'source': 'AA198', 'routing path': [['GS_EWR', 77.071]], 'End-to-end rate': 77.071, 'End-to-end latency': 50}
```

4.3.3. Test 3: Node with multiple outputs - From AA101 to any Ground Station

The objective of this test is to test a node with Multiple Outputs. The interesting thing about this node is that it can reach the two Ground Stations with the maximum E2E Transmission Rate.

In [22]:

```
#Far away from GS
#source = 0 (AA101) -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
Test3_2 = BFS_SP(data = df_res.iloc[:, 0].to_numpy(), matrix = tr_df, source = 0, target0 = 216, target1 = 217)

print("\033[1m Solution with minimum E2E Latency: \033[0m")
print(Test3_2[0])
```

```
 Solution with minimum E2E Latency:
{'source': 'AA101', 'routing path': [['AA723', 31.895], ['AA53', 31.895], ['DL75', 31.895], ['GS_LHR', 31.895]], 'End
-to-end rate': 31.895, 'End-to-end latency': 200}
```

5. Evaluation

5.1 Visualization

From the import and fixing of the data, you can start to have an idea of the results that can be obtained. For this reason, it is necessary to create visual aids such as plots of the data to be worked with. In step 5, the data import shows several nodes far away from any Ground Station, so it is inferred that they will not have any connection. Indeed, in Test 4.2.1. it is possible to verify the non-connection of a node and its respective visualization.

Additionally, with the visualization of some data, for example in Test 4.2.2. or 4.2.3. in addition to obtaining the respective solutions, it is also observed that there will be very close nodes, which cause multiple outputs, i.e. multiple solutions for the node. In Test 4.2.3. the graph of the solution with the lowest end-to-end latency and maximum end-to-end transmission rate is obtained.

These graphical solutions demonstrate in a visual way the effectiveness and veracity of the results obtained using the Multi-Output Dijkstra's Algorithm.

5.2. Efficiency

The efficiency of the algorithm is evaluated by running through all the nodes and saving the results in a text file. Additionally, the execution time of the algorithm is taken.

5.2.1. Dijkstra's Algorithm

Note: Before run the next functions please change run_code to TRUE. It can be take up to 40 min to go through all the nodes.

In [23]:

```python
#PLEASE CHANGE run_code = true
run_code = False

if run_code:
    # get the start time
    st = time.time()

    #source = all -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
    evaluation_d = run_test_dijkstra(len(df.iloc[:, 0].to_numpy())-2, df.iloc[:, 0].to_numpy(), tr_df, 216, 217)

    # get the end time
    et = time.time()

    # get the execution time
    elapsed_time = et - st
    print('Execution time:', elapsed_time, 'seconds')

    #save data
    with open('evaluation_D', 'w') as fout:
        json.dump(evaluation_d, fout)

    with open('evaluationAll_D', 'w') as fout:
        json.dump(evaluation_d[0], fout)

    with open('evaluationLatency_D', 'w') as fout:
        json.dump(evaluation_d[2], fout)
```

```
source: 197 n_solutions: 5 n_solutions_min_latency: 5
source: 198 n_solutions: 5 n_solutions_min_latency: 5
source: 199 n_solutions: 25 n_solutions_min_latency: 10
source: 200 n_solutions: 6 n_solutions_min_latency: 2
source: 201 n_solutions: 13 n_solutions_min_latency: 4
source: 202 n_solutions: 53 n_solutions_min_latency: 6
source: 203 n_solutions: 9 n_solutions_min_latency: 2
source: 204 n_solutions: 1 n_solutions_min_latency: 1
source: 205 n_solutions: 13 n_solutions_min_latency: 3
source: 206 n_solutions: 10 n_solutions_min_latency: 10
source: 207 n_solutions: 13 n_solutions_min_latency: 6
source: 208 n_solutions: 35 n_solutions_min_latency: 10
source: 209 n_solutions: 5 n_solutions_min_latency: 5
source: 210 n_solutions: 15 n_solutions_min_latency: 15
source: 211 n_solutions: 216 n_solutions_min_latency: 11
source: 212 n_solutions: 216 n_solutions_min_latency: 3
source: 213 n_solutions: 8 n_solutions_min_latency: 2
source: 214 n_solutions: 216 n_solutions_min_latency: 8
source: 215 n_solutions: 13 n_solutions_min_latency: 5
Execution time: 2300.468538045883 seconds
```

```
##### 5.2.1.1. Results when run_code = True

source: 0 n_solutions: 11 n_solutions_min_latency: 1 <br>
source: 1 n_solutions: 10 n_solutions_min_latency: 2 <br>
source: 2 n_solutions: 13 n_solutions_min_latency: 3 <br>
source: 3 n_solutions: 2 n_solutions_min_latency: 1 <br>
source: 4 n_solutions: 12 n_solutions_min_latency: 3 <br>
source: 5 n_solutions: 1 n_solutions_min_latency: 1 <br>
.
.
.
source: 0 n_solutions: 11 n_solutions_min_latency: 1
source: 1 n_solutions: 10 n_solutions_min_latency: 2
source: 2 n_solutions: 13 n_solutions_min_latency: 3
source: 3 n_solutions: 2 n_solutions_min_latency: 1
source: 4 n_solutions: 12 n_solutions_min_latency: 3
source: 5 n_solutions: 1 n_solutions_min_latency: 1

Elapsed
```

The execution of the algorithm for ALL nodes can take several minutes, in this case approximately 38 minutes. So its time efficiency is not the most adequate.

5.2.2. BFS

In [24]:

```python
# get the start time
st = time.time()

#ESTE ES
#source = all -- target0 = 216 (GS_LHR) -- target1 = 217 (GS_EWR)
evaluation_bfs = run_test_BFS(len(df_res.iloc[:, 0].to_numpy())-2, df_res.iloc[:, 0].to_numpy(), tr_df, 216, 217)

#Test
#evaluation = run_test_dijkstra(2, df.iloc[:, 0].to_numpy(), tr_df, 216, 217)

#evaluation = run_test_dijkstra(214, df.iloc[:, 0].to_numpy(), tr_df, 216, 217)

# get the end time
et = time.time()

# get the execution time
elapsed_time = et - st
print('Execution time:', elapsed_time, 'seconds')

with open('evaluation_BFS', 'w') as fout:
    json.dump(evaluation_bfs, fout)

```

```
Execution time: 2.4095354080200195 seconds
```

The execution of the algorithm for ALL nodes take only some seconds, in this case approximately 3 seconds. So its time efficiency is very fast compared to Multi-Output Dijkstra's Algorithm.

5.3. Data analysis

5.3.1. Dijkstra's Algorithm

In [25]:

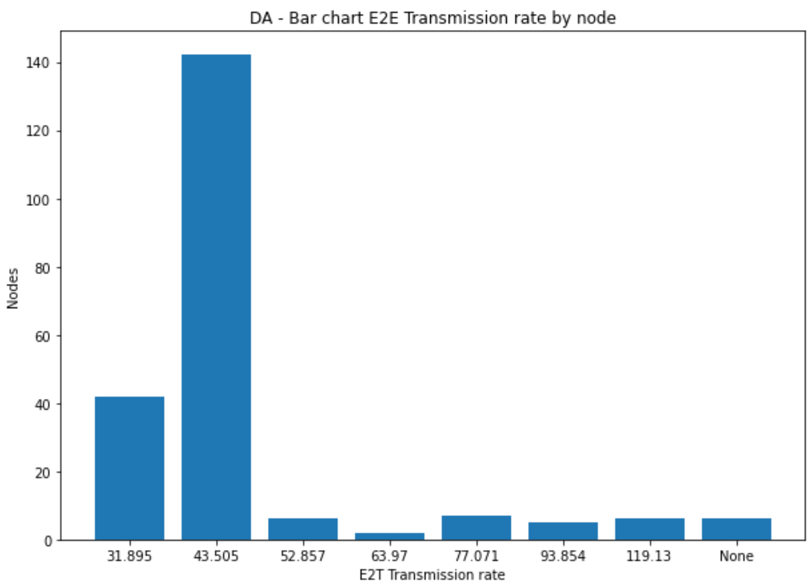
```python
# Opening JSON file
file_evaluation_D = open('evaluation_D')

# returns JSON object as a dictionary
read_data_D = json.load(file_evaluation_D)
```

5.3.1.1. E2E Transmission rate analysis

In [26]:

```
e2e_tr_nodes_D = []
for i in range(len(read_data_D[0])):
    e2e_tr_nodes_D.append(str(read_data_D[0][i][0].get('End-to-end rate')))

# Use a Counter to count the number of instances in x
c = Counter(e2e_tr_nodes_D)

keys = natsort.natsorted(c.keys())
c_new = OrderedDict((k, c[k]) for k in keys)

print(c_new)

plt.figure(figsize =(10, 7))

plt.bar(c_new.keys(), c_new.values())
plt.xlabel('E2T Transmission rate')
plt.ylabel('Nodes')
plt.title('DA - Bar chart E2E Transmission rate by node')
plt.show()

```

```
OrderedDict([('31.895', 42), ('43.505', 142), ('52.857', 6), ('63.97', 2), ('77.071', 7), ('93.854', 5), ('119.13', 6), ('None', 6)])
```



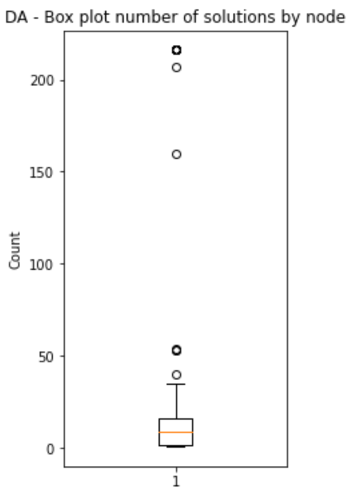
It is observed that with Dijkstra's Algorithm most nodes have an end-to-end transmission rate of 43.505 Mbps, followed by 31.895 Mbps. However, it is also shown that some nodes achieve an end-to-end Transmission rate of 52.857, 63.970, 77.071, 93.854 and even 11.9.13 Mbps.

Additionally, the nodes that do not have any end-to-end transmission rate because there is no connection with any Ground Station are shown.

In [27]:

```
#Number solutions by node
quant_D = np.quantile(read_data_D[1], [0,0.25,0.5,0.75,1])

print('min: ', quant_D[0]) # min
print('Q1: ', quant_D[1]) # min
print('median: ', quant_D[2]) # min
print('Q3: ', quant_D[3]) # min
print('max: ', quant_D[4]) # min

plt.figure(figsize =(3, 6))
plt.boxplot(read_data_D[1])
plt.ylabel('Count')
plt.title('DA - Box plot number of solutions by node')
plt.show()
```

```
min:  1.0
Q1:  2.0
median:  9.0
Q3:  16.5
max:  216.0
```



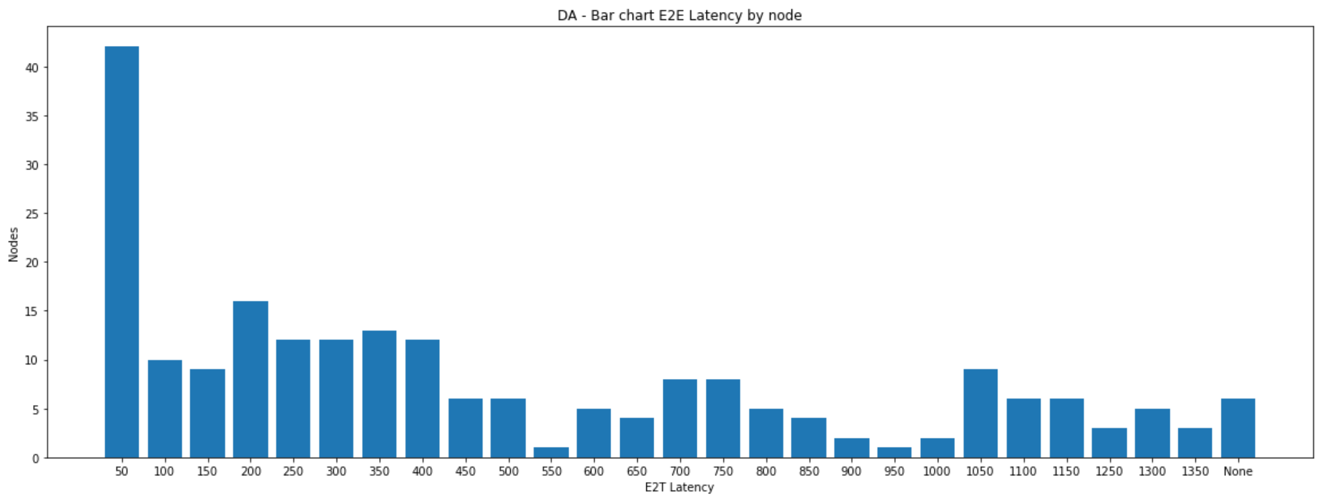
This graph shows that quartiles 1 and 3 are very close to the box with 2 and 16.5 respectively and the average generated is 9 solutions per node. The minimum is 1 solution, because the nodes with no connection also have a "No connection" solution. There are some nodes with too many edges around, so they reach up to a maximum of 216 solutions.

5.3.1.2. E2E Latency analysis

In [28]:

```
e2e_latency_nodes_D = []
for i in range(len(read_data_D[2])):
    if(read_data_D[2][i] != []):
        e2e_latency_nodes_D.append(str(read_data_D[2][i][0].get('End-to-end latency')))
    else:
        e2e_latency_nodes_D.append("None")

# Use a Counter to count the number of instances in x
c = Counter(e2e_latency_nodes_D)

keys = natsort.natsorted(c.keys())
c_new = OrderedDict((k, c[k]) for k in keys)

print(c_new)

plt.figure(figsize =(20, 7))

plt.bar(c_new.keys(), c_new.values())
plt.xlabel('E2T Latency')
plt.ylabel('Nodes')
plt.title('DA - Bar chart E2E Latency by node')
plt.show()
```

```
OrderedDict([('50', 42), ('100', 10), ('150', 9), ('200', 16), ('250', 12), ('300', 12), ('350', 13), ('400', 12),
('450', 6), ('500', 6), ('550', 1), ('600', 5), ('650', 4), ('700', 8), ('750', 8), ('800', 5), ('850', 4), ('900',
2), ('950', 1), ('1000', 2), ('1050', 9), ('1100', 6), ('1150', 6), ('1250', 3), ('1300', 5), ('1350', 3), ('None',
6)])
```



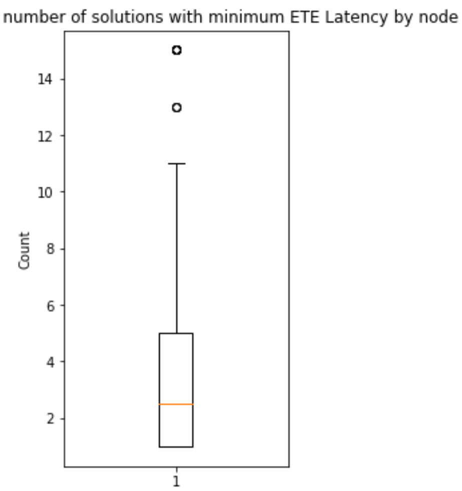
When observing this graph, it obtains a high number of nodes with an end-to-end latency of 50ms and the rest of the nodes distributed between 100ms and 1350ms. With this it can be concluded that in addition to finding an algorithm with a high range in end-to-end transmission rate, it also has a high performance with the end-to-end latency.

5.3.1.3. Number of solutions with minimum latency by node

In [29]:

```
#Number of solutions with minimum latency by node
quant_D = np.quantile(read_data_D[3], [0,0.25,0.5,0.75,1])

print('min: ', quant_D[0]) # min
print('Q1: ', quant_D[1]) # min
print('median: ', quant_D[2]) # min
print('Q3: ', quant_D[3]) # min
print('max: ', quant_D[4]) # min

plt.figure(figsize =(3, 6))
plt.boxplot(read_data_D[3])
plt.ylabel('Count')
plt.title('DA - Box plot number of solutions with minimum ETE Latency by node')
plt.show()
```

```
min:  1.0
Q1:  1.0
median:  2.5
Q3:  5.0
max:  15.0
```



This graph shows the box plot of the number of solutions with minimum end-to-end latency, with an average of 2.5 solutions, a minimum of 1 solution and a maximum of 15 solutions. This shows that the most optimal solutions with higher end-to-end transmission rate and lower end-to-end latency are much less than the main solutions provided by Dijkstra's Algorithm.

5.3.2. BFS

In [30]:

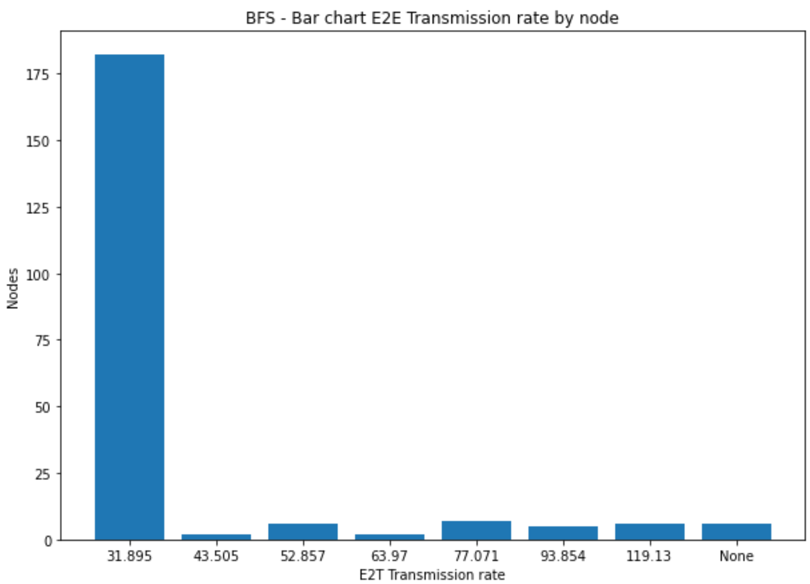
```
# Opening JSON file
file_evaluation_BFS = open('evaluation_BFS')

# returns JSON object as a dictionary
read_data_BFS = json.load(file_evaluation_BFS)

```

5.3.2.1. E2E Transmission rate analysis

In [31]:

```
e2e_tr_nodes_BFS = []
for i in range(len(read_data_BFS[0])):
    e2e_tr_nodes_BFS.append(str(read_data_BFS[0][i].get('End-to-end rate')))

# Use a Counter to count the number of instances in x
c = Counter(e2e_tr_nodes_BFS)

keys = natsort.natsorted(c.keys())
c_new = OrderedDict((k, c[k]) for k in keys)

print(c_new)

plt.figure(figsize =(10, 7))

plt.bar(c_new.keys(), c_new.values())
plt.xlabel('E2T Transmission rate')
plt.ylabel('Nodes')
plt.title('BFS - Bar chart E2E Transmission rate by node')
plt.show()

```

```
OrderedDict([('31.895', 182), ('43.505', 2), ('52.857', 6), ('63.97', 2), ('77.071', 7), ('93.854', 5), ('119.13', 6), ('None', 6)])
```

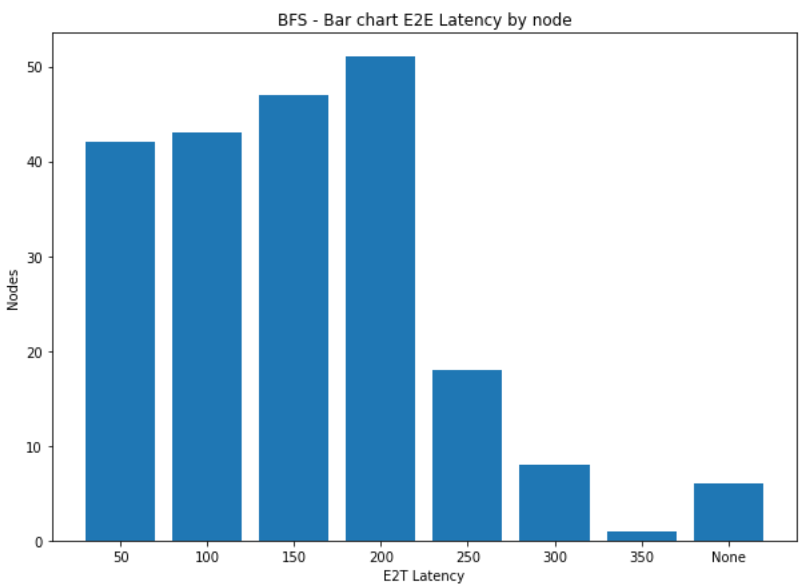


Using the BFS Algorithm it can be observed that most nodes will obtain a very low end-to-end transmission rate of 31.895Mbps. It is inferred that the other nodes with a higher end-to-end transmission rate are located near one of the Ground Stations.

5.3.2.2. E2E Latency analysis

In [32]:
```
e2e_latency_nodes_BFS = []
for i in range(len(read_data_BFS[0])):
    if(read_data_BFS[0][i] != []):
        e2e_latency_nodes_BFS.append(str(read_data_BFS[0][i].get('End-to-end latency')))
    else:
        e2e_latency_nodes_BFS.append("None")

# Use a Counter to count the number of instances in x
c = Counter(e2e_latency_nodes_BFS)

keys = natsort.natsorted(c.keys())
c_new = OrderedDict((k, c[k]) for k in keys)

print(c_new)

plt.figure(figsize =(10, 7))

plt.bar(c_new.keys(), c_new.values())
plt.xlabel('E2T Latency')
plt.ylabel('Nodes')
plt.title('BFS - Bar chart E2E Latency by node')
plt.show()
```

```
OrderedDict([('50', 42), ('100', 43), ('150', 47), ('200', 51), ('250', 18), ('300', 8), ('350', 1), ('None', 6)])
```



It is observed that the BFS Algorithm effectively provides a lower latency in its solutions, with similar results between 50, 100, 150 and 200ms. The nodes with higher latency only reach 350ms. Again, the nodes with None latency are those with no connection to any Ground Station.

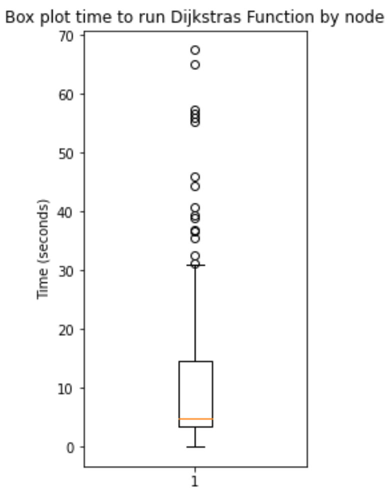
5.3.3. Comparison Dijkstra's Algorithm vs BFS

5.3.3.1. Runtime by node

In [33]:

```
#Time by node
quant_D = np.quantile(read_data_D[4], [0,0.25,0.5,0.75,1])

print('min: ', quant_D[0]) # min
print('Q1: ', quant_D[1]) # min
print('median: ', quant_D[2]) # min
print('Q3: ', quant_D[3]) # min
print('max: ', quant_D[4]) # min

plt.figure(figsize =(3, 6))
plt.boxplot(read_data_D[4])
plt.ylabel('Time (seconds)')
plt.title('Box plot time to run Dijkstras Function by node')
plt.show()
```

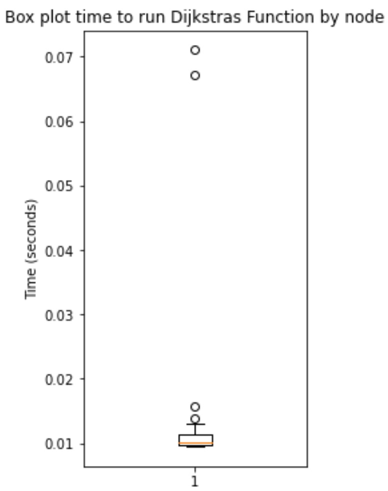
```
min:  0.00016117095947265625
Q1:  3.372894048690796
median:  4.849276423454285
Q3:  14.47942441701889
max:  67.46265912055969
```



In [34]:

```
#Time by node
quant_BFS = np.quantile(read_data_BFS[1], [0,0.25,0.5,0.75,1])

print('min: ', quant_BFS[0]) # min
print('Q1: ', quant_BFS[1]) # min
print('median: ', quant_BFS[2]) # min
print('Q3: ', quant_BFS[3]) # min
print('max: ', quant_BFS[4]) # min

plt.figure(figsize =(3, 6))
plt.boxplot(read_data_BFS[1])
plt.ylabel('Time (seconds)')
plt.title('Box plot time to run Dijkstras Function by node')
plt.show()
```

```
min:  0.009525060653686523
Q1:  0.009686946868896484
median:  0.010144591331481934
Q3:  0.01128464937210083
max:  0.07101607322692871
```

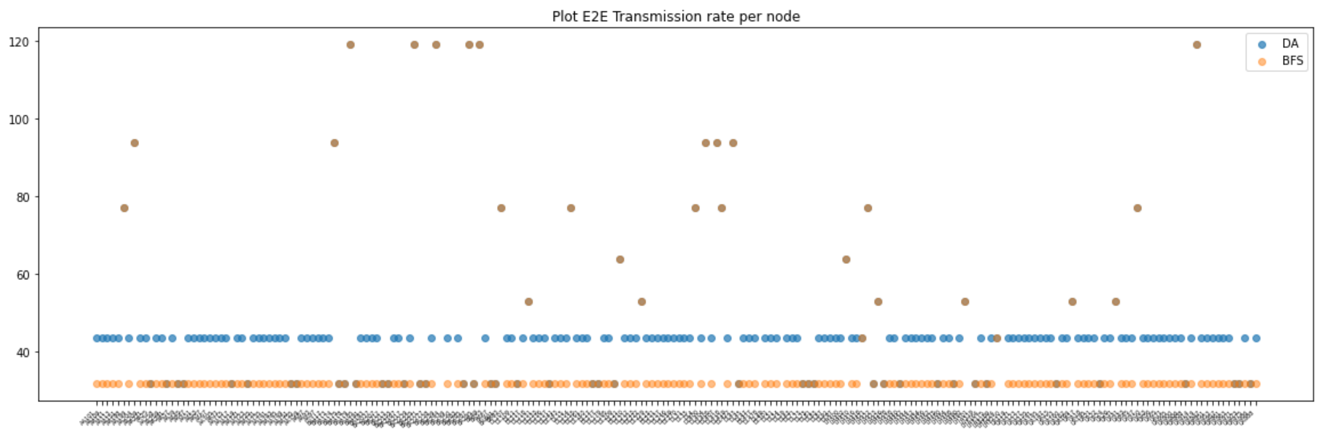


While Dijkstra's Algorithm has an average of 5 seconds per node and a maximum of up to 67 seconds, BFS has a much higher efficiency, with an average of 11ms and a maximum of 74ms.

5.4.3.1. End-to-end Transmission Rate per node

In [35]:

```
df_D_temp = []
for i in range(len(read_data_D[0])):
    df_D_temp.append(read_data_D[0][i][0])

data_D = df_D_temp
df_D = pd.DataFrame.from_records(data_D)

data_BFS = read_data_BFS[0]
df_BFS = pd.DataFrame.from_records(data_BFS)

# create data
x = df_BFS.get('source')
y_D = df_D.get('End-to-end rate')
y_BFS = df_BFS.get('End-to-end rate')


plt.figure(figsize =(20, 6))
plt.scatter(x, y_D, label = "DA", alpha= 0.7)
plt.scatter(x, y_BFS, label = "BFS", alpha= 0.5)
plt.title('Plot E2E Transmission rate per node')
plt.xticks(rotation=45, ha='right', size=5)
plt.legend()
plt.show()
```

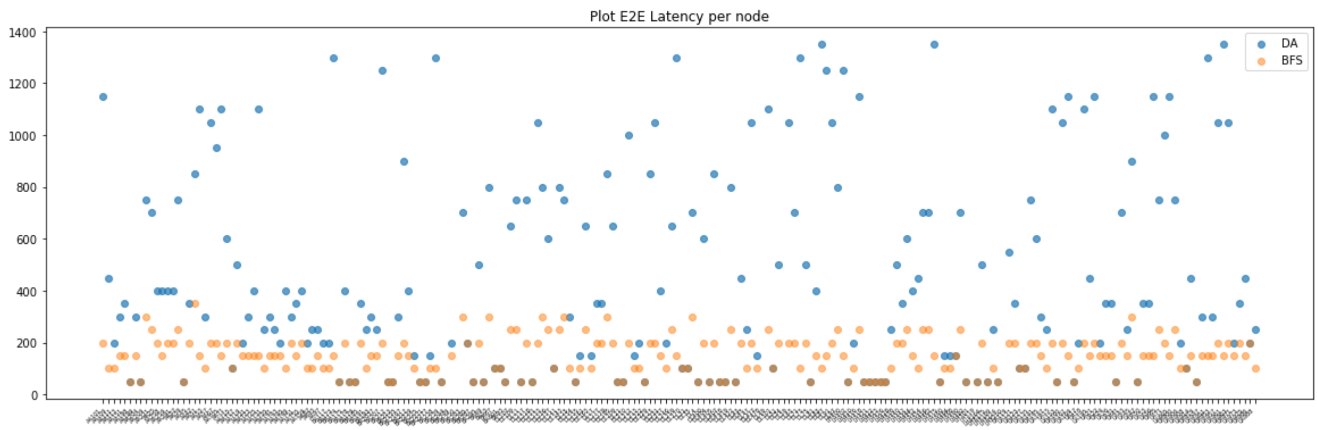


This graph compares the end-to-end transmission rate per node. In blue color the Dijkstra's Algorithm shows that it has a higher end-to-end transmission rate compared to the orange color of the BFS. It can be concluded that Dijkstra's Algorithm has a higher performance in this metric. In brown the values repeated in each node for each algorithm.

5.4.3.1. End-to-end Latency per node

In [36]:

```python
df_D_temp = []
for i in range(len(read_data_D[2])):
    df_D_temp.append(read_data_D[2][i][0])

data_D = df_D_temp
df_D = pd.DataFrame.from_records(data_D)

data_BFS = read_data_BFS[0]
df_BFS = pd.DataFrame.from_records(data_BFS)

# create data
x = df_BFS.get('source')
y_D = df_D.get('End-to-end latency')
y_BFS = df_BFS.get('End-to-end latency')


plt.figure(figsize =(20, 6))
plt.scatter(x, y_D, label = "DA", alpha= 0.7)
plt.scatter(x, y_BFS, label = "BFS", alpha= 0.5)
plt.title('Plot E2E Latency per node')
plt.xticks(rotation=45, ha='right', size=5)
plt.legend()
plt.show()
```



In contrast to the end-to-end transmission rate graph, this graph shows a better performance in the end-to-end latency metric by BFS in orange color, mostly distributed between 0 and 350ms. In blue color the distribution of Dijkstra's Algorithm is very varied and ranges from 50 to 1350ms. In brown the values repeated in each node for each algorithm.

6. Conclusions

The use of two optimization algorithms helped to evaluate the outputs of each. Both algorithms have advantages and disadvantages in response and efficiency that can be improved. A good understanding of the complexity of the problem makes it possible to achieve better solutions. The optimization algorithms were evaluated with respect to their responses in the metrics of end-to-end transmission rate, end-to-end latency and execution times per node.

By creating Dijkstra's Algorithm allowed to deliver a solution with multiple outputs for nodes with several nearby edges. There are some nodes close to some of the Ground Stations, these nodes take much less response time to deliver the data, with a minimum of 0.1 ms and an average of 5 seconds. However, by having several solutions for each node, the algorithm becomes more complex and can sometimes take up to 1 minute to deliver the solutions. To partially cover this flaw, the iterations were limited, however, a more robust algorithm can be designed, using fewer cycles to increase the efficiency of the algorithm.

On the other hand, the BFS Algorithm, showed a fairly high efficiency, in which its maximum execution times do not exceed 80 ms, the minimum time and the average are quite close at approximately 10 ms. BFS algorithm is an effective method for determining the shortest path in a matrix and can be used to address issues with routing and network latency. The problem context must be taken into account when selecting an algorithm due to its limitations, which include not accounting for each node's weight and simply displaying the first optimal solution. It may be useful to compare the results and come to a decision by making adjustments to the code to display the results similarly to the Dijkstra's Algorithm solution.

Regarding the maximum end-to-end transmission rate metric, Dijkstra's algorithm presents better results, with 65.7% with 43.505 Mbps and 19.4% with 31.895 Mbps, while 84.3% of the BFS results have a minimum of 31.895 Mbps. Furthermore, there is a big difference with respect to the minimum end-to-end latency metric, since in Dijkstra's Algorithm, although 19. 4% has a response of 50ms, there are some nodes that reach a high latency of up to 1350 ms, which implies inefficiency in the algorithm; for the case of BFS, the response is much better with a total sum of 84.7% in the latency range between 50 and 200 ms (23.6%, 21.7%, 19.9% and 19.4% at 200, 150, 100 and 50 ms respectively).

Finally, in addition to the advantage of higher end-to-end transmission rate of Dijkstra's Algorithm over BFS, the proposed algorithm provides multiple outputs, i.e. multiple options from which the best one can be chosen and thus find the Pareto Optimal of the responses.

7. Future research agenda

Firstly, it is recommended for advanced search for specific cases with low end-to-end latency but low end-to-end transmission rate. It is necessary to investigate new network protocols and architectures that prioritize low latency over high transmission rates. It is possible to examine the trade-offs between latency and transmission rate in different types of communication systems and networks, and then it could be found the Pareto Optimal. The performance of existing optimization algorithms could be evaluated in such scenarios and identifying potential improvements.

Furthermore, solutions can compare the trade-offs between low end-to-end latency and low end-to-end transmission rate versus high end-to-end latency and high end-to-end transmission rate. It is essential to measure and compare the performance of different communication systems and networks in terms of these parameters, investigating the use cases and applications where one trade-off is more desirable over the other. In different scenarios it can be developed theoretical models that can predict the optimal balance between latency and transmission rate. Nowadays, it is also indispensable to study the latency and transmission rate constraints in 5G and its impact on the IoT devices.

Since these evaluations were performed on a single timestamp, Evaluations are necessary measuring the performance of existing optimization algorithms under different timestamp conditions. For this, it is necessary to develop new optimization algorithms that are robust to changes in timestamp conditions. When studying the impact of different timezones on the optimization algorithm used in different geographical areas, more accurate and effective solutions can be achieved for real-time data.

References

Awerbuch, B. and R. Gallager (1987). “A new distributed algorithm to find breadth first search trees”. In: IEEE Transactions on Information Theory 33.3, pp. 315–322. doi: 10.1109/TIT.1987.1057314.

Bulterman, R.W., F.W. van der Sommen, G. Zwaan, T. Verhoeff, A.J.M. van Gasteren, and W.H.J. Feijen (2002). “On computing a longest path in a tree”. In: Information Processing Letters 81.2, pp. 93–96. issn: 0020-0190. doi: https://doi.org/10.1016/S0020-0190(01)00198-3 (https://doi.org/10.1016/S0020-0190(01)00198-3). url: https://www.sciencedirect.com/science/article/pii/S0020019001001983 (https://www.sciencedirect.com/science/article/pii/S0020019001001983).

Chen, Tingwei, Bin Zhang, Xianwen Hao, and Yu Dai (2006). “Task Scheduling in Grid Based on Particle Swarm Optimization”. In: 2006 Fifth International Symposium on Parallel and Distributed Computing, pp. 238–245. doi: 10.1109/ISPDC.2006.46.

“Dijkstra’s Algorithm” (2013). In: Encyclopedia of Operations Research and Management Science. Ed. by Saul I. Gass and Michael C. Fu. Boston, MA: Springer US, pp. 428–428. isbn: 978-1-4419-1153-7. doi: 10.1007/978-1-4419-1153-7_200148. url: https://doi.org/10.1007/978-1-4419-1153-7_200148 (https://doi.org/10.1007/978-1-4419-1153-7_200148).

Javaid, Adeel (Jan. 2013). “Understanding Dijkstra Algorithm”. In: SSRN Electronic Journal. doi: 10.2139/ssrn.2340905.

Johnson, Donald B (1973). “A note on Dijkstra’s shortest path algorithm”. In: Journal of the ACM (JACM) 20.3, pp. 385–388.

Kennedy, James and Russell Eberhart (1995). “Particle Swarm Optimization”. In: Proceedings of ICNN’95- international conference on neural networks. Vol. 4. IEEE, pp. 1942–1948.

Kurant, Maciej, Athina Markopoulou, and Patrick Thiran (2010). “On the bias of BFS (Breadth First Search)”. In: 2010 22nd International Teletraffic Congress (ITC 22), pp. 1–8. doi: 10.1109/ITC.2010.5608727.

Neji, Najett, Raul de Lacerda, Alain Azoulay, Thierry Letertre, and Olivier Outtier (2013). “Survey on the Future Aeronautical Communication System and Its Development for Continental Communications”. In: IEEE Transactions on Vehicular Technology 62.1, pp. 182–191. doi: 10.1109/TVT.2012.2207138.

Portugal, David, Carlos Henggeler Antunes, and Rui Rocha (2010). “A study of genetic algorithms for approximating the longest path in generic graphs”. In: 2010 IEEE International Conference on Systems, Man and Cybernetics, pp. 2539–2544. doi: 10.1109/ICSMC.2010.5641920.

Vey, Quentin, Alain Pirovano, Jos´e Radzik, and Fabien Garcia (2014). “Aeronautical ad hoc network for civil aviation”. In: International Workshop on Communication Technologies for Vehicles. Springer, pp. 81–93.

Zhang, Jiankang, Taihai Chen, Shida Zhong, Jingjing Wang, Wenbo Zhang, Xin Zuo, Robert G. Maunder, and Lajos Hanzo (2019). “Aeronautical Ad-Hoc Networking for the Internet-Above-the-Clouds”. In: Proceedings of the IEEE 107.5, pp. 868–911. doi: 10.1109/JPROC.2019.2909694.

Zhang, Jiankang, Dong Liu, Sheng Chen, Soon Xin Ng, Robert G Maunder, and Lajos Hanzo (2022). “Multiple-objective packet routing optimization for aeronautical ad-hoc networks”. In: IEEE Transactions on Vehicular Technology.