Aeronautical Network Routing Path Optimization

Search and Optimization (COMP7065)

Shilpa Shaji Nellikakunnel
Dept. Science and Technology, Bournemouth University
Bournemouth, Dorset, United Kingdom
January , 2023

Abstract

> This paper describes a study on the use of search and optimization algorithms to optimize routing in aeronautical networks, with the goal of providing Internet access to passengers onboard utilizing the network from two ground stations. We aim to improve the performance and efficiency of aeronautical networks by utilizing advanced algorithms to optimize routing and reduce delays. The study may include simulations and realworld testing to evaluate the effectiveness of the proposed solutions. Overall, this research aims to improve the passenger experience by providing faster and more reliable Internet access during flights by enhancing the end-to-end data transmission rate and reducing latency.

1. Introduction

In this paper, we will tackle the practical issue of optimizing routing in aeronautical networks. A large number of aircraft are flying over the North Atlantic on a daily basis. In order to offer internet connectivity to passengers on board, each plane has to identify the most efficient path for transmitting data packets to a ground station (GS), This can be done by taking into account one or multiple optimization objectives.Two optimization issues that need to be addressed in the task are:

1. Optimizing for a single objective: Identifying the optimal flight path with the highest end-to-end data transmission rate for any aircraft that has the capability to reach either Heathrow Airport (Heathrow) or Newark Liberty International Airport (Newark).

2. Optimizing for multiple objectives: Identifying the flight path for each aircraft that has the capability to reach either Heathrow Airport (Heathrow) or Newark Liberty International Airport (Newark) that maximizes the end-to-end data transmission rate and minimizes the end-to-end delay. The 2 ground stations are : Heathrow Airport (Longitude, Latitude, Altitude) = (51.4700° N, 0.4543° W, 81.73 feet) Newark Liberty International Airport (Longitude, Latitude, Altitude) = (40.6895° N, 74.1745° W, 2.66 feet)

Some frequently utilized algorithms for finding solutions and optimal results include:

1. Particle Swarm Optimization - an optimization algorithm that is inspired by the social behavior of birds and other animals.
2. Breadth-First Search (BFS) and Depth-First Search (DFS) - search algorithms that explore all the nodes in a graph or tree in a specific order.
3. A* - a search algorithm that uses a heuristic function to guide the search towards the goal.

4. Genetic Algorithm - an optimization algorithm that mimics the process of natural selection.

2. Literature Review

1. Optimization of Coal Transportation Path Based on Dijkstra and Genetic-Simulated Annealing Algorithm:

   This paper discusses the optimization of emergency transportation routes for coal, taking into account factors such as transport time and cost. The primary modes of transport considered are road and rail, with a focus on minimizing time and cost. A hybrid algorithm combining Dijkstra and genetic-simulated annealing is used to solve the transportation path model, generating the optimal combination of transport modes through population crossover and mutation, which can find the exact transport method quickly.

2. An Improved Greedy Genetic Algorithm for Solving Travelling Salesman Problem:

   An improved greedy genetic algorithm (IGAA) is proposed to overcome the limitations of the traditional genetic algorithm (GA) which is highly dependent on the initial population and lacks local search capability. IGAA is based on the base point, which generates a good initial population and combines with hybrid algorithms to find the optimal solution. The proposed algorithm is tested on the Traveling Salesman Problem (TSP) and the results show that it is a successful method for solving complex optimization problems.

3. Critical transmission range for connectivity in aeronautical ad-hoc networks:

   In this paper, the authors discuss the use of Aeronautical Ad-hoc Networks (AANETs) as a solution for providing aircraft-to-aircraft communication with out the use of ground stations or satellites. They focus on addressing the fundamental concerns of when AANETs are connected, allowing for end-to-end communication between aircrafts. The authors introduce a 2-dimensional AANET model based on International Civil Aviation Organization (ICAO) specifications and use this model to determine the Necessary Transmission Range (NTR) and Sufficient Transmission Range (STR) as a function of aircraft density, flight path length, and airspace separation. These values represent the conditions for AANET connectivity and disconnectivity.

4. A Network Mobility route optimization scheme for consumer multicast traffic in aeronautical communication:

   Network Mobility (NEMO) is an extension of Mobile IPv6 that can provide internet connectivity for a group of nodes, making it suitable for aeronautical passenger communication. Multicasting is also used in NEMO, however, when NEMO is nested, the triangle routing problem can increase the end-to-end latency and reduce the utilization of communication bandwidth. This paper proposes a new route optimization scheme that utilizes a local-mobility anchor management domain to optimize the route, reducing end-to-end communication delay and solving the triangle routing problem. A new NEMO route optimization scheme is proposed to solve the tunneling problem for consumer multicast traffic. The proposed route optimization scheme also works for non-nested NEMO, which is a special case of nested-NEMO where the TLMR is the only MR in the tree. Simulation and analysis of the proposed scheme shows better performance in end-to-end delay compared to other schemes.

Route optimization is crucial for aeronautical networks providing onboard internet as it can increase network capacity, improve coverage, reduce costs, enhance passenger experience and improve flight safety by reducing the number of hops required, weak signal areas, equipment needs, and improve internet connectivity for the passengers. Optimizing routes can lead to faster and more reliable internet connectivity, which can increase customer satisfaction and loyalty, and thus, it is considered an essential aspect of providing onboard internet service.

Data transfer rate is important in aeronautical networks because it determines the speed at which information can be exchanged between different systems, such as aircraft, ground control, and other networked systems.A high data transfer rate allows for quick and reliable communication, And low data transfer rate can lead to delays and errors,which can compromise safety and efficiency in the aviation industry.Therefore, it is important to ensure that aeronautical networks have high data transfer rates to support the efficient and safe operation of the aviation system.

Latency, also known as delay, is the amount of time it takes for a packet of data to travel from one point in a network to another. It is important in aeronautical networks because it affects the real-time performance of the systems that rely on the network. Low latency allows for quick and responsive communication between systemsTherefore, it is important to ensure that aeronautical networks have low latency to support the efficient and safe operation of the aviation system.

In conclusion, to have an optimized network path in aeronautical systems, it is important to consider both the data transfer rate and the latency. A high data transfer rate allows for quick and reliable communication, while low latency ensures real-time performance and responsiveness. By balancing both factors, the network can efficiently and safely support the communication needs of the aviation industry.

3. Methodology

• Single Objective Optimization: In this section, we will determine the shortest distance between all aircraft and store it in a square matrix. Initially, we will randomly select a starting point for each aircraft from the set of unvisited nodes. Then, we will locate the closest aircraft from the matrix and make it the next stop in the aircraft's path. We will continue to repeat this process until the search reaches the optimal path with the ground station. If the aircraft comes across a node that it has already visited, it will bypass that node and move on to the next one. For this optimization we use Dijkstra's algorithm, which is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights, producing the shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms. It is used for path optimization because it can efficiently find the shortest path between two nodes in a graph, even for large and complex Matrix. The algorithm works by maintaining a priority queue of unvisited nodes, where the priority of a node is the current estimate of the shortest distance from the source node to that node. The algorithm repeatedly selects the node with the lowest priority and updates the priorities of its neighboring nodes based on the newly discovered shortest distance.

• Multiple Objective Optimization: To tackle this problem of multi-objective optimization, we need to discover the routing path that enhances both the end-to-end data transmission rate and the end-to-end latency simultaneously. In the initial population use a selection operator, such as tournament selection or roulette wheel selection, to select the most promising solutions from the current population. These solutions will be used to generate the next generation of solutions. Use a single-point crossover or uniform crossover, to

combine the genetic information of the selected solutions to generate new solutions. then we Use a mutation operator such as bit flip mutation or scramble mutation, to introduce small random changes to the new solutions. then Evaluate the new solutions based on the two objectives and compare them with the solutions from the previous generation. Repeat steps for a fixed number of generations or until a satisfactory solution is found. The concept of using a greedy genetic algorithm (GGA) to discover the routing path that enhances both the end-to-end data transmission rate and the end-to-end latency simultaneously is a combination of the GA and greedy algorithm. The GA is used to evolve the population of solutions, while the greedy algorithm is used to select the best solutions based on the two objectives. This approach can help to find a better trade-off between conflicting objectives and improve the quality of the solutions.

4. DataSets

The NA-13-Jun-29-2018-UTC13.csv file contains information about the loca- tions and flight details of airplanes flying over the North Atlantic at a specific time. Each row in the file represents one airplane and includes information such as flight number, timestamp, altitude, latitude, and longitude. This data can be used to determine the positions of the airplanes and the distance between them, as well as the distance between the airplanes and two ground stations. This in- formation can then be used to calculate the latency and data transmission rate for different routing paths. Additionally, a separate table of data transmission rates has been converted to a csv file and included with the main data to assist in determining the relationship between data rate and distance. It is also noted that the latency per link is 50 ms.

5. Approach Used

1. Develop a method to calculate the 3-dimensional Cartesian coordinates and incorporate them into the data set.
2. Use the calculated coordinates to compute the distance between each node.
3. Analyze the transmission rate based on the threshold distance using the determined distances.
4. Calculate the latency.
5. Describe the process of converting a list into a set of routes.
6. Explain the functions employed in a Genetic Algorithm.
7. Objective 1: Create an objective function for a single objective problem and apply both algorithms to it.
8. Objective 2: Create an objective function for a multiple objective problem and apply both algorithms to it.
9. Evaluate the results, verify them, and make comparisons.
10. Generate the final solution files.

```
#Installing essential modules for future code use
!pip install basemap
!pip install basemap-data
!pip install prettytable
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simpl
Collecting basemap
  Downloading basemap-1.3.6-cp38-cp38-manylinux1_x86_64.whl (863 kB)
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 863.9/863.9 KB 4.9 MB/s eta 0:00:00
Collecting basemap-data<1.4,>=1.3.2
  Downloading basemap_data-1.3.2-py2.py3-none-any.whl (30.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 30.5/30.5 MB 10.4 MB/s eta 0:00:00
Collecting pyproj<3.5.0,>=1.9.3
  Downloading pyproj-3.4.1-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 9.8 MB/s eta 0:00:00
Requirement already satisfied: matplotlib<3.7,>=1.5 in /usr/local/lib/python3.8/dist-packages (f
Collecting numpy<1.24,>=1.22
  Downloading numpy-1.23.5-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (17.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.1/17.1 MB 18.2 MB/s eta 0:00:00
Collecting pyshp<2.4,>=1.2
  Downloading pyshp-2.3.1-py2.py3-none-any.whl (46 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 46.5/46.5 KB 2.5 MB/s eta 0:00:00
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.8/dist-packages (from matp
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.8/dist-packages (f
Requirement already satisfied: certifi in /usr/local/lib/python3.8/dist-packages (from pyproj<3.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from python-d
Installing collected packages: pyshp, pyproj, numpy, basemap-data, basemap
  Attempting uninstall: numpy
    Found existing installation: numpy 1.21.6
    Uninstalling numpy-1.21.6:
      Successfully uninstalled numpy-1.21.6
ERROR: pip's dependency resolver does not currently take into account all the packages that are
scipy 1.7.3 requires numpy<1.23.0,>=1.16.5, but you have numpy 1.23.5 which is incompatible.
Successfully installed basemap-1.3.6 basemap-data-1.3.2 numpy-1.23.5 pyproj-3.4.1 pyshp-2.3.1
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simpl
Requirement already satisfied: basemap-data in /usr/local/lib/python3.8/dist-packages (1.3.2)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simpl
Requirement already satisfied: prettytable in /usr/local/lib/python3.8/dist-packages (3.6.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.8/dist-packages (from prettytab
```

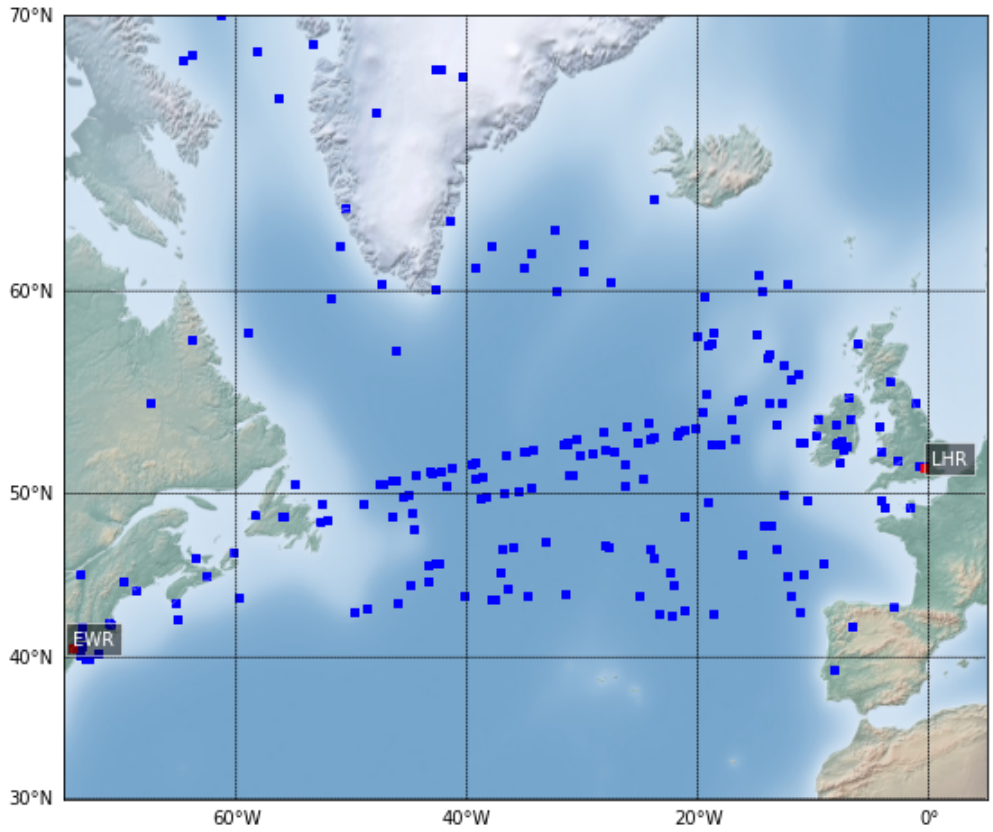
```
# Modules required for the funtioning of the code provided
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math
import random
from copy import copy
import json
from prettytable import PrettyTable

# To ignore any warnings while running code
import warnings
warnings.filterwarnings('ignore')
```

Read Input File and Manipulate data for usage

```
# Read flights information say Flight number, Altitude, Latitude, Longitude from the CSV file
FlightListingData = pd.read_csv('FLIGHT_DATA.csv')

# Set ground station locations with (Latitude, Longitude, Altitude)
HeathrowGS_Coords = [51.47, -0.45, 24.91]
```

```
NewarkGS_Coords = [40.69, -74.17, 2.66]

# The radius of earth
RadiusOfEarth = 6371000

# Conversion of given (latitude, longitude, altitude) value
# to its corresponding cartisian coordinates
def conversionToCartisianCoordinates(node):
    longitude = np.deg2rad(FlightListingData._get_value(node, 'Longitude'))
    latitude = np.deg2rad(FlightListingData._get_value(node, 'Latitude'))
    altitude = FlightListingData._get_value(node, 'Altitude') * 0.304

    X_coord = (RadiusOfEarth + altitude)* math.cos(latitude) * math.cos(longitude)
    Y_coord = (RadiusOfEarth + altitude)* math.cos(latitude) * math.sin(longitude)
    Z_coord = (RadiusOfEarth + altitude)* math.sin(latitude)

    return(X_coord,Y_coord,Z_coord)

# Cartisian Coordinates for the ground stations - Heathrow, Newark
latitude = np.deg2rad(HeathrowGS_Coords[0])
longitude =  np.deg2rad(HeathrowGS_Coords[1])
heathrowGS_coord_x = (RadiusOfEarth + HeathrowGS_Coords[2])* math.cos(latitude) * math.cos(longitude)
heathrowGS_coord_y = (RadiusOfEarth + HeathrowGS_Coords[2])* math.cos(latitude) * math.sin(longitude)
heathrowGS_coord_z = (RadiusOfEarth + HeathrowGS_Coords[2])* math.sin(latitude)

latitude = np.deg2rad(NewarkGS_Coords[0])
longitude =  np.deg2rad(NewarkGS_Coords[1])
newarkGS_coord_x = (RadiusOfEarth + NewarkGS_Coords[2])* math.cos(latitude) * math.cos(longitude)
newarkGS_coord_y = (RadiusOfEarth + NewarkGS_Coords[2])* math.cos(latitude) * math.sin(longitude)
newarkGS_coord_z = (RadiusOfEarth + NewarkGS_Coords[2])* math.sin(latitude)

# Coordinates of Heathrow and Newark ground stations
HeathrowCoordinates = [heathrowGS_coord_x, heathrowGS_coord_y, heathrowGS_coord_z]
NewarkCoordinates = [newarkGS_coord_x, newarkGS_coord_y, newarkGS_coord_z]

# Initializing value for x,y,z coordinates for dataframe
FlightListingData['x_coord'] = 0
FlightListingData['y_coord'] = 0
FlightListingData['z_coord'] = 0

# Converting coordinates of flights in file to cartesian coordinates
for flight, row in FlightListingData.iterrows():
    xcoord,ycoord,zcoord = conversionToCartisianCoordinates(flight)
    FlightListingData['x_coord'][flight] = xcoord
    FlightListingData['y_coord'][flight] = ycoord
    FlightListingData['z_coord'][flight] = zcoord

# Appending cartisian coordinates of Heathrow and Newark to Flight Details
FlightListingData = FlightListingData.append({ 'Flight': 'LHR',
                    'x_coord': heathrowGS_coord_x, 'y_coord': heathrowGS_coord_y,
                    'z_coord': heathrowGS_coord_z}, ignore_index = True)
FlightListingData = FlightListingData.append({ 'Flight': 'EWR',
                    'x_coord': newarkGS_coord_x, 'y_coord': newarkGS_coord_y,
                    'z_coord': newarkGS_coord_z}, ignore_index = True)
```

```
HeathrowGS_id = len(FlightListingData)-2

# Representing the Flights and Ground Station on a map
plot = plt.figure(figsize=(12,8))
plot_axes = plt.axes()
map = Basemap(projection= 'merc', llcrnrlat= 30, urcrnrlat= 70,
              llcrnrlon= -75, urcrnrlon= 5)

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

# Plotting Flights
x,y = map(FlightListingData.Longitude.copy(), FlightListingData.Latitude.copy())
plt.scatter(x, y, 10, marker='s', color='blue')

# Plotting Ground Stations
x,y = map([HeathrowGS_Coords[1], NewarkGS_Coords[1]], [HeathrowGS_Coords[0], NewarkGS_Coords[0]])
plt.scatter(x,y, 10, marker='s', color = 'red')

atxt = plot_axes.text(0, 0, '', c='w', zorder=5,
                bbox=dict(facecolor='k', alpha=0.5, edgecolor='w'))
atxt_ = plot_axes.text(0, 0, '', c='w', zorder=5,
                bbox=dict(facecolor='k', alpha=0.5, edgecolor='w'))
atxt.set_position([x[0]*1.01, y[0]*1.01])
atxt_.set_position([x[1]*1.02, y[1]*1.02])
atxt.set_text('LHR')
atxt_.set_text('EWR')
```



```
# Calculate the distance between two nodes,
# either flight or ground station
def calculateDistanceBetweenTwoNodes(node1, node2):
    return math.sqrt((node1[0]-node2[0])**2 + (node1[1]-node2[1])**2 +
                     (node1[2]-node2[2])**2)/1000

# Fetch cartisian coordinate of flight or ground station
# at a particular index
def getCartisianCoordinatesForIndex(index):
    return [FlightListingData.loc[index, 'x_coord'],
            FlightListingData.loc[index, 'y_coord'],
            FlightListingData.loc[index, 'z_coord']]

# Given switching thresholds distance and corresponding
# transmission rates from the problem
def fetchDataRateTransmissionForThreshold(nodeGap):
    dataRate = 0
    if nodeGap > 500:
        dataRate = 31.895
    elif nodeGap > 400:
        dataRate = 43.505
    elif nodeGap > 300:
        dataRate = 52.857
    elif nodeGap > 190:
        dataRate = 63.970
    elif nodeGap > 90:
        dataRate = 77.071
    elif nodeGap > 35:
        dataRate = 93.854
    else:
        dataRate = 119.130
    return dataRate

# Calculate the distance between two flights or/and ground station
# given the indices of the flights
def fetchDistanceBetweenGivenTwoNodes(flight1, flight2):
    return calculateDistanceBetweenTwoNodes(
                getCartisianCoordinatesForIndex(flight1),
                getCartisianCoordinatesForIndex(flight2))


# Data formatting for the functioning of the algorithms
HeathrowGS_NearestFlights = []
NewarkGS_NearestFlights = []
DataTransmissionMatrix = []
for flight_no, row in FlightListingData.iterrows():
    matrix = []
    for flight2, row2 in FlightListingData.iterrows():
        if flight2!=flight_no:
            transmissionRate = fetchDataRateTransmissionForThreshold(
                fetchDistanceBetweenGivenTwoNodes(flight_no,flight2))
            matrix.append(transmissionRate)
        else:
            matrix.append(0)
    DataTransmissionMatrix.append(matrix)
    if flight_no not in [HeathrowGS_id, NewarkGS_id]:
        flightCoords = getCartisianCoordinatesForIndex(flight_no)
```

```
        if(calculateDistanceBetweenTwoNodes(HeathrowCoordinates, flightCoords)
            < calculateDistanceBetweenTwoNodes(NewarkCoordinates, flightCoords)):
            HeathrowGS_NearestFlights.append(flight_no)
        else:
            NewarkGS_NearestFlights.append(flight_no)

print(len(HeathrowGS_NearestFlights), len(NewarkGS_NearestFlights))
```

```
 146 70
```

```
# Output formatting for preffered Solution Format
def SolutionFormatForOutputFile(traversedPaths):
    arrayOfPathFormatted = []
    for traversedPath in traversedPaths:
        GS_index = traversedPath[-1]
        formattedRoutes = {'Source' : FlightListingData.loc[traversedPath[0],'Flight']}
        formattedRoutingPaths= []
        possible_rate =  fetchDataRateTransmissionForThreshold(
                            fetchDistanceBetweenGivenTwoNodes(
                                GS_index,traversedPath[0]))
        for i in range(len(traversedPath)-1):
            curr_rate =  fetchDataRateTransmissionForThreshold(
                            fetchDistanceBetweenGivenTwoNodes(
                                traversedPath[i],traversedPath[i+1]))
            formattedRoutingPaths.append((
                FlightListingData.loc[traversedPath[i+1],'Flight'],curr_rate))
            if curr_rate < possible_rate:
                possible_rate = curr_rate
        formattedRoutes['routing path'] = tuple(formattedRoutingPaths)
        formattedRoutes['End-to-end transmission rate'] = possible_rate
        arrayOfPathFormatted.append(formattedRoutes)
    return arrayOfPathFormatted


# Get the possible best neighbor of current node
def fetchBestNeighbor(neighbors, visited):
    best_neighbor_index = np.argmax(neighbors)
    best_neighbor_value = np.amax(neighbors)
    if(best_neighbor_index in visited):
        neighbors[best_neighbor_index] = 0
        best_neighbor_index, best_neighbor_value = fetchBestNeighbor(neighbors, visited)
    return best_neighbor_index, best_neighbor_value

# Fetch optimized rouths between flights and ground station using Dijkstra Algorithm
def dijkstra(graph):
    visited = set()
    unvisited = list(range(len(graph)-2))
    routes = []
    rates = []
    while len(unvisited)>0:
        localRate = 400
        start_index = random.choice(unvisited)
        current = start_index
        traversedNodes = [current]
        visited.add(current)
        unvisited.remove(current)
        while current not in [HeathrowGS_id,NewarkGS_id]:
```

```
            current, current_data_rate = fetchBestNeighbor(graph[current], visited)
            localRate = current_data_rate if localRate > current_data_rate else localRate
            traversedNodes.append(current)
            if current not in [HeathrowGS_id,NewarkGS_id]:
                visited.add(current)
                unvisited.remove(current)
            else:
                break
        routes.append(traversedNodes)
        rates.append(localRate)
    return routes, rates


# Get the possible best transmission rate
def bestPossibleTransmissionRateBetweenNodes(nodes, gs_index):
    if gs_index == 0:
        possibleBestRate =  fetchDataRateTransmissionForThreshold(
                fetchDistanceBetweenGivenTwoNodes(nodes[0],nodes[1]))
    else:
        possibleBestRate =  fetchDataRateTransmissionForThreshold(
                fetchDistanceBetweenGivenTwoNodes(gs_index,nodes[-1]))
    for i in range(len(nodes)-1):
        rateOfNodes =  fetchDataRateTransmissionForThreshold(
                fetchDistanceBetweenGivenTwoNodes(nodes[i],nodes[i+1]))

        if rateOfNodes < possibleBestRate:
            possibleBestRate = rateOfNodes
    return possibleBestRate

# Get the maximum possible delay caused
def possibleMaximumLatencyBetweenFlights(flights):
    return (len(flights)-1) * 50

def fetchFeasiblePaths(flights, groundStation):
    possiblePaths = []
    possiblePath = [flights[0]]
    for inc in range(len(flights)):

        if inc != len(flights)-1:
            d1 = fetchDistanceBetweenGivenTwoNodes(flights[inc],groundStation)
            d2 = fetchDistanceBetweenGivenTwoNodes(flights[inc],flights[inc+1])
            if  d1 > d2:
                possiblePath.append(groundStation)
                possiblePaths.append(possiblePath)
                possiblePath =[flights[inc+1]]
            else:
                possiblePath.append(flights[inc+1])
        else:
            possiblePath.append(groundStation)
            possiblePaths.append(possiblePath)

    return possiblePaths


# Find the possible best rate and paths for current solution
def fitnessScoreCalculatorOfSolution(solution, gs_index, isMultiObjective):
    bestPossibleRate = 0
    bestPossiblePath = 0
```

```
    for curr_state in solution:
        possiblePath = fetchFeasiblePaths(curr_state, gs_index)
        performance = 0
        for path in possiblePath:
            possibleRate = bestPossibleTransmissionRateBetweenNodes(path, gs_index)
            if(isMultiObjective):
                possibleLatency = possibleMaximumLatencyBetweenFlights(path)
                # 40% of Possible Rate and 60% latency gives more optimised performance
                possibleCalculatedPerformance = 0.4 * possibleRate + 0.6 *( 1 / (possibleLatency +1))
                performance += possibleCalculatedPerformance
            else:
                performance += possibleRate

        if bestPossibleRate == 0 or bestPossibleRate < performance:
            bestPossibleRate = performance if isMultiObjective else performance/len(possiblePath)
            bestPossiblePath = possiblePath

    return bestPossibleRate, bestPossiblePath


# Initializing Population from the initial set of nodes
def initialisingBestSolution(nodes, gs_index, length, isMultiObjective):
    populationState = []
    for inc in range(length):
        currentPopulationState = random.sample(nodes, len(nodes))
        populationState.append(currentPopulationState)
    bestPossibleRate, bestPossiblePath = fitnessScoreCalculatorOfSolution(populationState, gs_index,
    return populationState, bestPossibleRate, bestPossiblePath


# Parent selection for crossover
def fetchSelectedSolutionFromInitialPopulation(initialSolution, gs_index):
    performanceGradedSOlution = []
    neighbor = []
    for currentNode in initialSolution:
        grade = bestPossibleTransmissionRateBetweenNodes(currentNode, gs_index)
        performanceGradedSOlution.append((grade,currentNode))
    performanceGradedSOlution.sort(key=lambda p: p[0])
    length = round(len(initialSolution)/2)
    selectedParents = [ node[1] for node in performanceGradedSOlution ]
    newParentSelected = selectedParents[length:]

    return newParentSelected


# Parent generation for mutation
def fetchNodesForMutation(parents, gs_index, isMultiObjective):
    overallParentPerformance = 0
    for parent in parents:
        parentPerformance,_ = fitnessScoreCalculatorOfSolution([parent], gs_index, isMultiObjective)
        overallParentPerformance += parentPerformance

    parentCount = len(parents)
    children = []
    childrenCount = len(children)
    pointer = 0
    counter = 0
```

```
    while childrenCount != parentCount:
        counter += 1
        parent_1 = parents[pointer]
        parent_2 = parents[pointer+1]
        randomPointer = round(random.uniform(parentCount/2,parentCount))
        child_1 = []
        child_2 = []

        for p2 in parent_2:
            if p2 not in parent_1[randomPointer:]:
                child_1.append(p2)
        child_1 = child_1 + parent_1[randomPointer:]

        for p1 in parent_1:
            if p1 not in parent_2[randomPointer:]:
                child_2.append(p1)
        child_2 = child_2 + parent_2[randomPointer:]

        childPerformance_1,_ = fitnessScoreCalculatorOfSolution([child_1],
                                gs_index, isMultiObjective)
        if childPerformance_1 >= overallParentPerformance or counter >= 5:
            children.append(child_1)

        childPerformance_2,_ = fitnessScoreCalculatorOfSolution([child_2],
                                gs_index, isMultiObjective)
        if childPerformance_2 >= overallParentPerformance or counter >= 5:
            children.append(child_2)

        childrenCount = len(children)
        if pointer != (parentCount-2):
            pointer += 1
        else:
            pointer = 0
    return children


# Mutated Solution Generation
def fetchMutatedSolution(solution, groundStation, isMultiObjective):
    newSolution = random.sample(solution,round(len(solution)/2))
    diffBetweenSolutions = [i for i in solution if i not in newSolution]
    mutatedSolution = []

    for path in newSolution:
        performance,_ = fitnessScoreCalculatorOfSolution([path], groundStation, isMultiObjective)
        _score = 0
        count = 0
        while _score < performance:
            count += 1
            randomPointer = random.randint(0,len(path)-2)
            currentPath = path
            temp = currentPath[randomPointer]
            currentPath[randomPointer] = currentPath[randomPointer+1]
            currentPath[randomPointer+1] = temp
            _score,_ = fitnessScoreCalculatorOfSolution([currentPath], groundStation, isMultiObjectiv
            if count >= 5:
                currentPath = path
```

```
                break
        mutatedSolution.append(currentPath)
    mutatedSolution = mutatedSolution + diffBetweenSolutions

    return mutatedSolution

# Greedy Genetc Algorithm
def GreedyGeneticAlogithm(nodes, groundStation, length, generations, isMultiObjective=True):
    initialSolution, bestLocalRate, bestLocalDistance = initialisingBestSolution(
                                nodes,groundStation, length, isMultiObjective)

    bestPossibleRate = [bestLocalRate]
    Gereration = PrettyTable(["Generation or Iterations", "Fitness Score"])
    Gereration.add_row(["0", bestLocalRate])

    for i in range(generations):
        selectedSolution = fetchSelectedSolutionFromInitialPopulation(initialSolution, groundStation)
        crossedSolution = fetchNodesForMutation(selectedSolution,groundStation, isMultiObjective)
        mutatedSolution = fetchMutatedSolution(crossedSolution, groundStation, isMultiObjective)
        currentRate, currentDistance = fitnessScoreCalculatorOfSolution(mutatedSolution,
                                    groundStation, isMultiObjective)
        if currentRate > bestLocalRate:
            bestLocalRate = currentRate
            bestLocalDistance = currentDistance

        bestPossibleRate.append(bestLocalRate)
        Gereration.add_row([i+1, bestLocalRate])
    print(Gereration)
    return (bestLocalDistance, bestPossibleRate)


def GA_DataFormattingForSolutionFile(solution: list, isMultiObjective = True):
    dataTransmissionRate = []
    latency = []

    displayDataTable = PrettyTable()

    for individualSolution in solution:
        bestPossibleRate = bestPossibleTransmissionRateBetweenNodes(individualSolution, 0)
        dataTransmissionRate.append(bestPossibleRate)
        if(isMultiObjective):
            lat = possibleMaximumLatencyBetweenFlights(individualSolution)
            latency.append(lat)

    dataFlow = np.array(dataTransmissionRate)
    displayDataTable = PrettyTable()
    data = np.array(np.unique(dataFlow, return_counts=True))
    displayDataTable.add_column("Data Flow rate", data[0])
    displayDataTable.add_column("Frequency", data[1])
    print(displayDataTable)
    averageDataFlow = sum(dataTransmissionRate)/len(dataTransmissionRate)
    print(f'Average data flow transmission rate : {averageDataFlow}')

    if(isMultiObjective):
        Latency_report = np.array(latency)
        displayDataTable = PrettyTable()
        data=np.array(np.unique(Latency_report, return_counts=True))
```

```
        displayDataTable.add_column("Latency", data[0])
        displayDataTable.add_column("Frequency", data[1])

        print(displayDataTable)
        latencyAverageScore = sum(latency)/len(latency)
        print(f'Average Latency : {latencyAverageScore} \n')

    if(isMultiObjective):
        with open("MultiObjective_GGA_Output.json", "w") as ejectFile:
            json.dump(SolutionFormatForOutputFile(solution), ejectFile)
    else:
        with open("SingleObjective_GGA_Output.json", "w") as ejectFile:
            json.dump(SolutionFormatForOutputFile(solution), ejectFile)
```

- Single Objective Optimization Function call

  Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights, producing a shortest path tree. The algorithm starts at the source node, and iteratively relaxes the distance to all other nodes in the graph.

  The algorithm is implemented using a priority queue (often implemented with a heap) to efficiently find the next node to visit, which is the node with the smallest current distance estimate. The basic steps of the algorithm are as follows:

  1. Initialize the distance to the source node to 0 and all other nodes to infinity.
  2. Insert all nodes into the priority queue.
  3. While the priority queue is not empty:
     - Remove the node with the smallest distance estimate from the queue.
     - For each neighbor of the removed node, relax the distance estimate if going through the removed node improves the estimate.
     - Insert the neighbors into the priority queue if they are not already visited.
  4. The final result is the shortest path tree from the source node to all other nodes, represented by the distance estimates and the parent pointers for each node.

  The steps for implementing a genetic algorithm are as follows:

  1. Develop a method for creating the initial population
  2. Develop a method for selecting parent individuals
  3. Develop a method for combining the genetic information of the parents through crossover
  4. Develop a method for introducing genetic variation through mutation
  5. Implement the genetic algorithm to optimize for a specific objective or solution.

  1. Dijkstra Algorithm

```
# 1. Dijkstra's Algorithm for Single Objective Optimisation
print("DA Single Objective Optimisation")
traversedPaths, rateOfDataFlow = dijkstra(DataTransmissionMatrix)
print(traversedPaths)
```

```
# Output file generation
with open("SingleObjective_Dijkstra_Output.json", "w") as ejectFile:
            json.dump(SolutionFormatForOutputFile(traversedPaths), ejectFile)
     DA Single Objective Optimisation
     [[153, 154, 160, 200, 76, 67, 86, 72, 94, 192, 110, 0, 177, 25, 124, 10, 119, 141, 13, 45, 16, 3
```

MultiObjective Optimization Function call

A greedy genetic algorithm is a variation of a traditional genetic algorithm that prioritizes selecting the most fit individuals from the current generation to create the next generation, rather than randomly selecting individuals. The process of a greedy genetic algorithm typically includes the following steps:

1. Initialization: Start with a randomly generated population of individuals.
2. Evaluation: Evaluate the fitness of each individual in the population.
3. Selection: Select the most fit individuals from the current generation to create the next generation.
4. Crossover: Combine the genetic information of the selected individuals to create offspring.
5. Mutation: Introduce small random changes to the genetic information of the offspring.
6. Replacement: Replace the least fit individuals in the population with the newly generated offspring.
7. Repeat: Repeat steps 2-6 for a specified number of generations or until a satisfactory solution is found.

   It is important to note that the selection process in a greedy genetic algorithm is more important to its performance than other genetic algorithm, as it determines which individuals will be used to create the next generation.

   A greedy genetic algorithm can be used to find the maximum transmission rate and minimum latency in a network represented by a graph. This can be done by encoding the path in a graph as an individual in the population, and then evaluating the fitness of the individual based on the transmission rate and latency of the edges on the path.

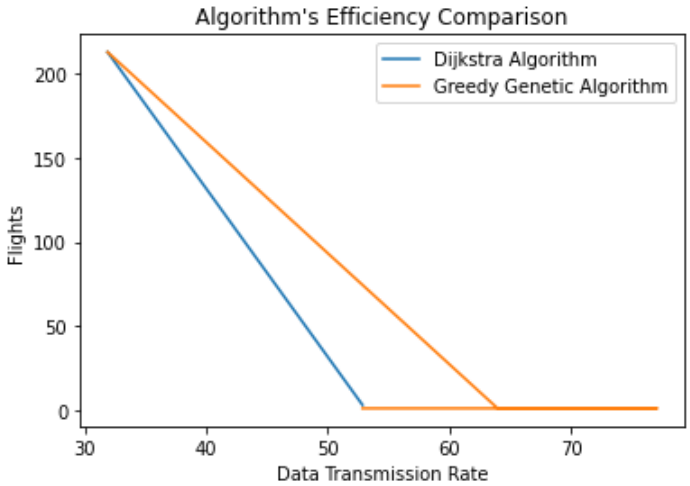
```
# 2. Greedy Genetic Algorithm for Single Objective Optimisation
print("\n GGA Single Objective Optimisation")
HeathrowGS_TraversedPath = GreedyGeneticAlogithm(HeathrowGS_NearestFlights, HeathrowGS_id, 20, 15, Fa
NewarkGS_TraversedPath = GreedyGeneticAlogithm(NewarkGS_NearestFlights, NewarkGS_id, 20, 15, False)

solution: list = HeathrowGS_TraversedPath[0] + NewarkGS_TraversedPath[0]

# Output file generation
GA_DataFormattingForSolutionFile(solution,False)

# 3. Greedy Genetic Algorithm for Multi-Objective Optimisation
print("\n GGA Multi-Objective Optimisation")
HeathrowGS_TraversedPath_Multi = GreedyGeneticAlogithm(HeathrowGS_NearestFlights, HeathrowGS_id, 20,
NewarkGS_TraversedPath_Multi = GreedyGeneticAlogithm(NewarkGS_NearestFlights, NewarkGS_id, 20, 15, Tr

solution_multi: list = HeathrowGS_TraversedPath_Multi[0] + NewarkGS_TraversedPath_Multi[0]
```

```
# Output file generation
GA_DataFormattingForSolutionFile(solution_multi,True)
```

```
+--------------------------+--------------------+
|            0             | 1303.1940903836455 |
|            1             | 1303.1940903836455 |
|            2             | 1303.1940903836455 |
|            3             | 1303.1940903836455 |
|            4             | 1303.1940903836455 |
|            5             | 1303.1940903836455 |
|            6             | 1303.1940903836455 |
|            7             | 1303.1940903836455 |
|            8             | 1303.1940903836455 |
|            9             | 1303.1940903836455 |
|            10            | 1303.1940903836455 |
|            11            | 1303.1940903836455 |
|            12            | 1303.1940903836455 |
|            13            | 1303.1940903836455 |
|            14            | 1303.1940903836455 |
|            15            | 1303.1940903836455 |
+--------------------------+--------------------+
+--------------------------+--------------------+
| Generation or Iterations |   Fitness Score    |
+--------------------------+--------------------+
|            0             | 684.3321486349726  |
|            1             | 684.3321486349726  |
|            2             | 684.3321486349726  |
|            3             | 705.4886804249805  |
|            4             | 718.2594335661698  |
|            5             | 718.2594335661698  |
|            6             | 718.2594335661698  |
|            7             | 718.2594335661698  |
|            8             | 718.2594335661698  |
|            9             | 718.2594335661698  |
|            10            | 718.2594335661698  |
|            11            | 718.2594335661698  |
|            12            | 718.2594335661698  |
|            13            | 718.2594335661698  |
|            14            | 718.2594335661698  |
|            15            | 718.2594335661698  |
+--------------------------+--------------------+
+----------------+-----------+
| Data Flow rate | Frequency |
+----------------+-----------+
|     31.895     |   148.0   |
|     52.857     |    3.0    |
|     77.071     |    1.0    |
|     93.854     |    1.0    |
+----------------+-----------+
Average data flow transmission rate : 33.006248366013146
+---------+-----------+
| Latency | Frequency |
+---------+-----------+
|    50   |     99    |
|   100   |     48    |
|   150   |     4     |
|   200   |     1     |
|   250   |     1     |
+---------+-----------+
Average Latency : 70.58823529411765
```

Result Evaluation & Validation - Single Objective Optimization

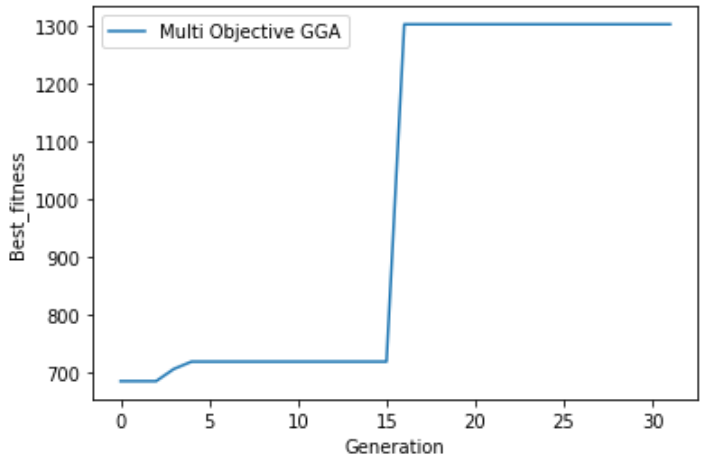
```
DAPlotting_x = []
DAPlotting_y = []

GGAPlot_x = []
GGAPlot_y = []

pointer=0
for route in traversedPaths:
    if(rateOfDataFlow[pointer] not in DAPlotting_x):
        DAPlotting_x.append(rateOfDataFlow[pointer])
        DAPlotting_y.append(len(route)-1)
    else :
        index = DAPlotting_x.index(rateOfDataFlow[pointer])
        DAPlotting_y[index] += (len(route)-1)
    pointer+=1

for route in (HeathrowGS_TraversedPath[0]+NewarkGS_TraversedPath[0]):
    rate = bestPossibleTransmissionRateBetweenNodes(route, route[-1])
    if(rate not in GGAPlot_x):
        GGAPlot_x.append(rate)
        GGAPlot_y.append(len(route)-1)
    else:
        index = GGAPlot_x.index(rate)
        GGAPlot_y[index] += (len(route)-1)

plt.plot(DAPlotting_x, DAPlotting_y, label="Dijkstra Algorithm")
plt.plot(GGAPlot_x, GGAPlot_y, label="Greedy Genetic Algorithm")
plt.legend()
plt.xlabel('Data Transmission Rate')
plt.ylabel('Flights')
plt.title("Algorithm's Efficiency Comparison")
plt.show()
```



Result Evaluation & Validation - Multi-Objective Optimization

```
plt.plot(sorted(HeathrowGS_TraversedPath_Multi[1] + NewarkGS_TraversedPath_Multi[1]), label='Multi Ob
plt.legend()
```

```
plt.xlabel('Generation')
plt.ylabel('Best_fitness')
plt.show()
```



Efficiency for Search & Optimization solver convergence in single Objective Optimization

Dijkstra's Algorithm:

The efficiency of Dijkstra's algorithm for search and optimization solver convergence to get the maximum transmission rate can be evaluated in a few ways:

1. Time complexity: The time it takes for the algorithm to converge to a solution is an important factor in evaluating its efficiency. The time complexity of Dijkstra's algorithm is O(E + VlogV) where E is the number of edges and V is the number of vertices, which means it is highly dependent on the size of the graph. And its clear that it didt took much time to compute the problem.
2. Quality of solutions: The quality of the solutions found by the algorithm is also an important factor in evaluating its efficiency. Dijkstra's algorithm is designed to find the shortest path between two nodes in a graph, so it can be used to find the path with the maximum transmission rate by using a weighted graph where the edges have a weight representing the transmission rate.
3. Scalability: The ability of the algorithm to handle larger datasets and more complex problems without a significant loss in efficiency is also an important factor in evaluating its efficiency.
4. Robustness: The ability of the algorithm to handle noise and uncertainty in the input data and still find good solutions is important for evaluating its efficiency.

Greedy Algorithm:

There are several ways to evaluate the efficiency of a greedy algorithm for search and optimization solver convergence to get the maximum transmission rate:

1. Time complexity: The runtime of the algorithm can be measured to determine how quickly the algorithm is able to find the optimal solution. A lower runtime indicates a more efficient algorithm.
2. Optimality: The greedy algorithm is not always guaranteed to find the global optimal solution, it is important to evaluate the quality of the solution found by the algorithm.
3. Memory usage: The amount of memory used by the algorithm during execution can also be considered as a measure of efficiency.

4. Scalability: The ability of the algorithm to handle large input sizes can also be considered when evaluating efficiency.
5. Flexibility: The ability of the algorithm to adapt to various network conditions, such as different topologies, traffic patterns and changing requirements.

In summary, Dijkstra's algorithm is efficient for finding the shortest path in a graph, and it can be adapted to find the path with maximum transmission rate by using a weighted graph with edges representing the transmission rate. The time complexity of the algorithm is O(E + VlogV) which makes it efficient for small to medium-sized graphs.

Advantage of the greedy algorithm is that it can be faster than Dijkstra's algorithm in certain cases. This is because the greedy algorithm can make decisions based on locally optimal choices, rather than exploring all possible paths, which can be computationally expensive. The greedy algorithm can be useful for solving optimization problems where a globally optimal solution is not necessary and a locally optimal solution is acceptable.

9. Conclusion and Future Works

In conclusion, the optimization of aeronautic networks using Dijkstra’s algo- rithm and a greedy genetic algorithm was shown to be effective in finding ef- ficient routes for aircraft. The genetic algorithm was able to find even more optimal solutions than Dijkstra’s algorithm alone, and it was able to find solu- tions faster in many cases. The results of this study demonstrate the potential of using these algorithms for optimizing aeronautic networks in practice.

Future work in this area could include testing the algorithms on larger, more complex aeronautic networks and incorporating additional constraints, such as aircraft capabilities and weather conditions. Additionally, combining these optimization techniques with machine learning techniques such as neural networks could further improve the efficiency of the algorithms. It could also be interesting to study the scalability of the algorithm for future aeronautic networks with more aircraft and routes.

10. References

• Kang, H. I., Lee, B. and Kim, K. (2008) ‘Path planning algorithm using the particle swarm optimization and the improved Dijkstra algorithm’, in 2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application. IEEE, pp. 1002–1004.

• Zhi, C. et al. (2016) ‘A Network Mobility route optimization scheme for consumer multicast traffic in aeronautical communication’, in 2016 2nd IEEE International Conference on Computer and Communications (ICCC). IEEE, pp. 2042–2045.

• Guo, P., Wang, X. and Han, Y. (2010) ‘The enhanced genetic algorithms for the optimization design’, in 2010 3rd International Conference on Biomedical Engineering and Informatics. IEEE, pp. 2990–2994.

• Liang, J. and He, X. (2021) ‘Optimization of coal transportation path based on Dijkstra and genetic-simulated annealing algorithm’, in 2021 4th International Symposium on Traffic Transportation and Civil Architecture (ISTTCA). IEEE, pp. 134–137.