Project code:

https://github.com/poopcaap/SI507-final

Data sources:

http://api.travelpayouts.com/data/en/airports.json, form website https://support.travelpayouts.com/hc/en-us/articles/360018907280-Flight-data-by-Aviasales
The request returns a file with a list of airports from the database, e.g.:


```
[{
  "code":"MQP",
  "name":"Kruger Mpumalanga International Airport",
   "iata_type": "airport",
   "flightable": true,
  "coordinates":{
    "lon":31.098131,
    "lat":-25.384947
  },
  "time_zone":"Africa/Johannesburg",
  "name_translations":{
    "de":"Nelspruit",
    "en":"Kruger Mpumalanga International Airport",
    "tr":"International Airport",
    "it":"Kruger Mpumalanga",
    "fr":"Kruger Mpumalanga",
    "es":"Kruger Mpumalanga",
    "th":"สนามบินเนลสปรุต"
  },
  "country_code":"ZA",
  "city_code":"NLP"
}]
```


There're 3693 number of data, all available and retrieved. This data is used to calculate distance between two airports, and to show them on the map designed,

http://api.travelpayouts.com/data/routes.json
The request returns a file with a list of routes from the database. E.g.:


```
[{
    "airline_iata":"2B",
    "airline_icao":null,
    "departure_airport_iata":"LON",
    "departure_airport_icao":null,
    "arrival_airport_iata":"BCN",
    "arrival_airport_icao":null,
```

```
    "codeshare":false,
    "transfers":0,
    "planes":[
        "CR2"
    ]
}]
```


There are 64964 data in this json, all available and retrieved. We will use the airline code, departure airport, arrival airport and plane type to do the search tree.

Data Structure

Description of the Graphs or Trees

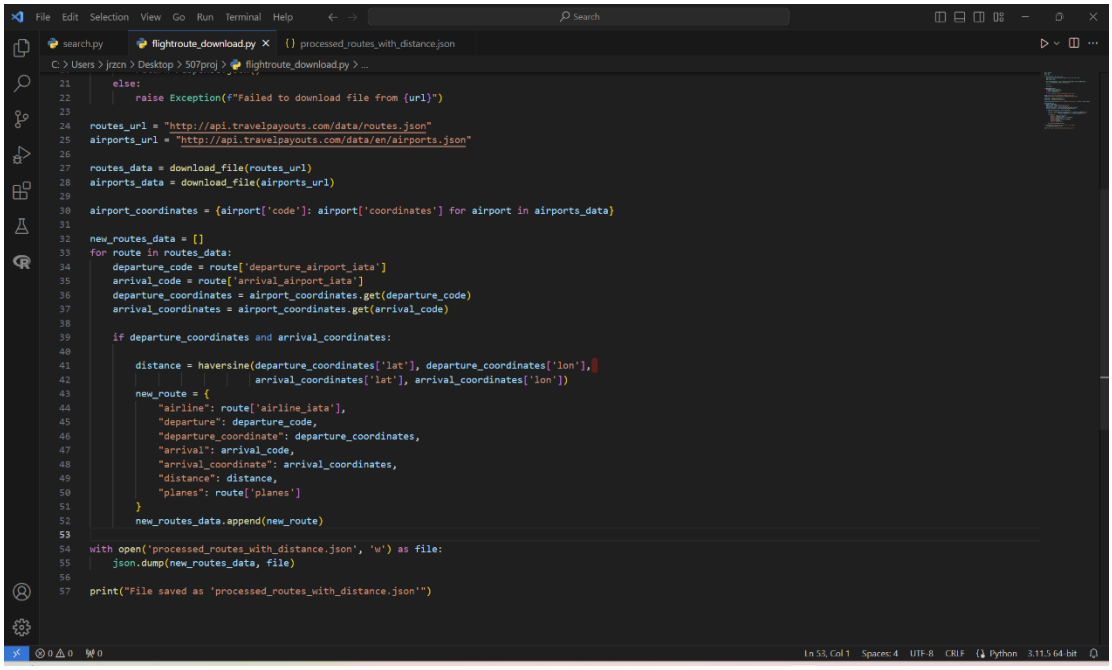
The data structure designed for the project is primarily a tree-based structure, which organizes flight route information in a hierarchical and accessible manner. The tree's nodes represent various attributes of flight data, such as airlines, departure locations, and plane models. Each node is linked in a way that reflects the logical relationship and hierarchy of these attributes:

1. Root Node: The root of the tree is conceptual, representing the start of the search criteria.
2. First Level Nodes - Airlines: Each node at this level represents an airline, serving as a gateway to further detailed information.
3. Second Level Nodes - Departures: Branching from each airline node, these nodes represent various departure locations.
4. Third and fourth Level Nodes - Planes and Distances: Each departure node leads to information about plane models and their respective distances, categorized into distinct intervals (e.g., 1-650 miles, 651-1150 miles, etc.).

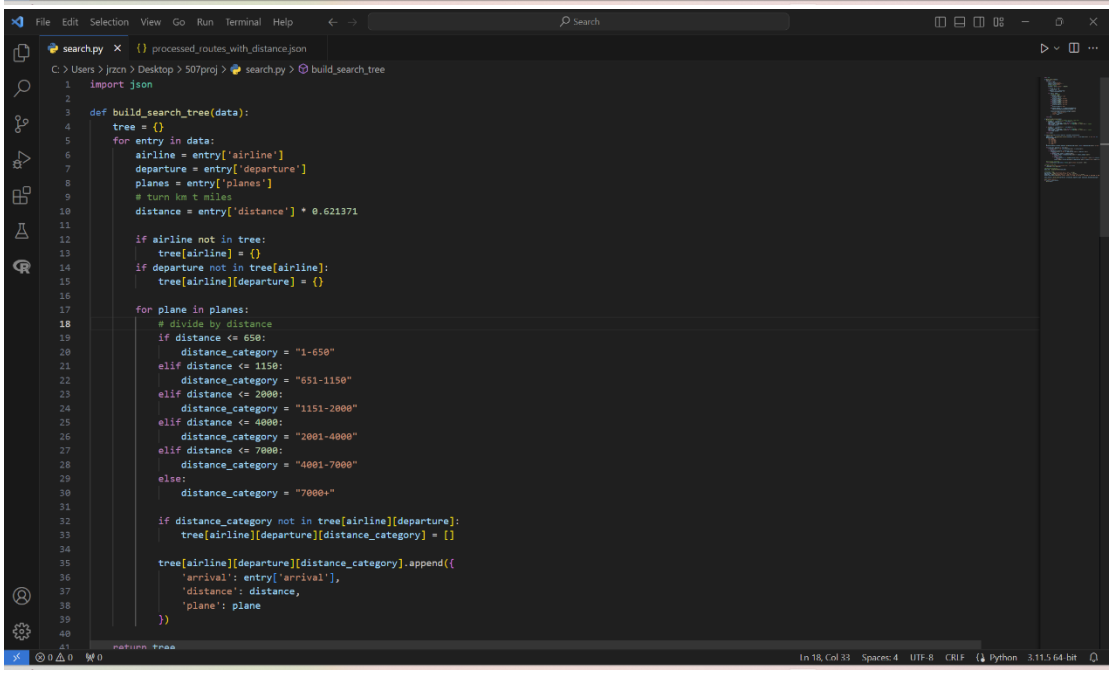
Assembling Data into Trees

Data is assembled into this tree structure by parsing two JSON files: one containing routes (routes.json) and the other containing airport details (airports.json). The process involves the following steps:

1. Reading and Merging Data: Extract relevant data from both JSON files, focusing on airlines, departures, arrivals, plane models, and distances.
2. Calculating Distances: Using the Haversine formula to calculate the distance between departure and arrival locations based on their latitude and longitude.
3. Building the Tree: For each route, create nodes under the corresponding airline and departure, and then organize plane models and distances under these nodes.



```
    else:
        raise Exception(f"Failed to download file from {url}")

routes_url = "http://api.travelpayouts.com/data/routes.json"
airports_url = "http://api.travelpayouts.com/data/en/airports.json"

routes_data = download_file(routes_url)
airports_data = download_file(airports_url)

airport_coordinates = {airport['code']: airport['coordinates'] for airport in airports_data}

new_routes_data = []
for route in routes_data:
    departure_code = route['departure_airport_iata']
    arrival_code = route['arrival_airport_iata']
    departure_coordinates = airport_coordinates.get(departure_code)
    arrival_coordinates = airport_coordinates.get(arrival_code)

    if departure_coordinates and arrival_coordinates:

        distance = haversine(departure_coordinates['lat'], departure_coordinates['lon'],
                             arrival_coordinates['lat'], arrival_coordinates['lon'])
        new_route = {
            "airline": route['airline_iata'],
            "departure": departure_code,
            "departure_coordinate": departure_coordinates,
            "arrival": arrival_code,
            "arrival_coordinate": arrival_coordinates,
            "distance": distance,
            "planes": route['planes']
        }
        new_routes_data.append(new_route)

with open('processed_routes_with_distance.json', 'w') as file:
    json.dump(new_routes_data, file)

print("File saved as 'processed_routes_with_distance.json'")
```



```
import json

def build_search_tree(data):
    tree = {}
    for entry in data:
        airline = entry['airline']
        departure = entry['departure']
        planes = entry['planes']
        # turn km t miles
        distance = entry['distance'] * 0.621371

        if airline not in tree:
            tree[airline] = {}
        if departure not in tree[airline]:
            tree[airline][departure] = {}

        for plane in planes:
            # divide by distance
            if distance <= 650:
                distance_category = "1-650"
            elif distance <= 1150:
                distance_category = "651-1150"
            elif distance <= 2000:
                distance_category = "1151-2000"
            elif distance <= 4000:
                distance_category = "2001-4000"
            elif distance <= 7000:
                distance_category = "4001-7000"
            else:
                distance_category = "7000+"

            if distance_category not in tree[airline][departure]:
                tree[airline][departure][distance_category] = []

            tree[airline][departure][distance_category].append({
                'arrival': entry['arrival'],
                'distance': distance,
                'plane': plane
            })

    return tree
```

Interaction and Presentation Plans

User-Facing Capabilities

The project allows users to interactively query and explore flight route data. Users can:

Select an Airline: Choose from a list of airlines or input an airline code.
Choose a Departure Location: Select or input a departure airport code.
Specify a Plane Model: Input a specific plane model or a category (e.g., '320' for all models starting with '320').
Filter by Distance: Choose from predefined distance ranges (e.g., A for 1-650 miles) or input 'any' for no distance filter.

View Results: Display filtered flight routes in a sorted order based on distance, presented as readable text output (e.g., "Airline ABC, Departure XYZ – Arrival QWE, Distance: 500 miles, Plane: 320").

Interactive and Presentation Technologies

Command Line Interface (CLI): The primary interaction mode, offering prompts for user inputs and displaying results.

Future Plan:

Flask: For developing a web-based interface, enabling a more interactive and user-friendly experience.

Plotly: For graphical presentation of routes and data, such as interactive maps showing flight paths or histograms for distance distributions**.**