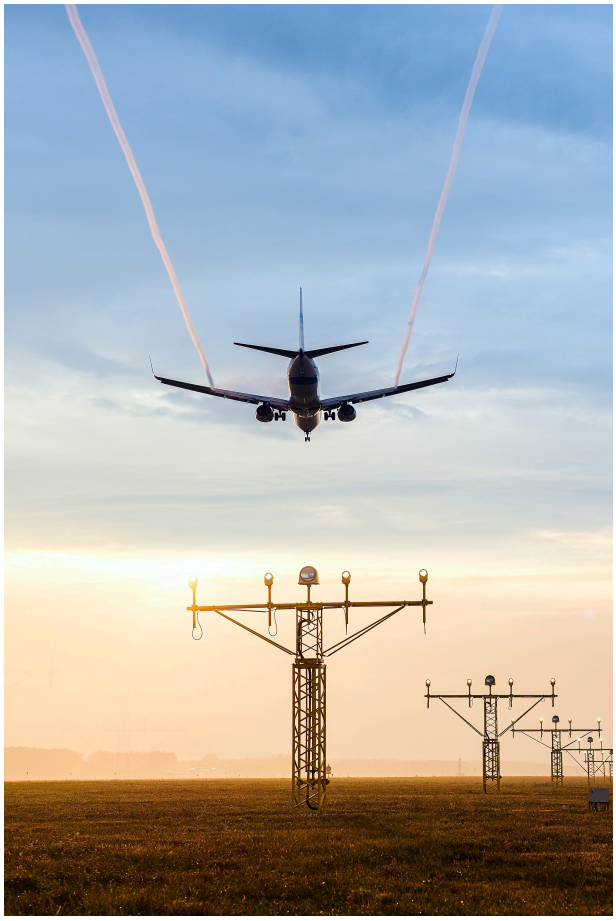


Team Blue Group Project

Flow networks and airlines

ABSTRACT

The primary objective of this project is to deliver an efficient and accessible tool for finding the maximum flight capacity between selected destinations, coupled with an option to visualize this data. This project presents a web-based application, developed with Flask, designed to optimize flight route capacities between cities. Utilizing the Edmond-Karp algorithm, the application analyzes a network of flights, represented as an adjacency matrix, to compute the maximum capacity paths. Users interact with a user-friendly interface, inputting source and destination cities, and receive detailed flight information. The application's robust error handling provides guidance on input errors, such as non-existent cities or identical source and destination entries.

Hanna Butsko, Randy Otto, Ray Bidini, Rosemary Mejia, Tim Abramov

COURSE IDC5120: Algorithms for Data Science

OBJECTIVE

The objective of the project was to consider trips between destinations that involve no more than 1 layover (i.e. trips involving a beginning and at most one intermediate stop before the destination). For a test case, New York was considered the source city and San Francisco was considered the destination city. We then had to figure out a way to manipulate the data to answer these questions:

1. Given your inferred load on each flight, what is the maximal number of people that can be moved from New York to San Francisco?
2. Which carrier can transport the greatest number of individuals from New York to San Francisco?
3. Create a dashboard or method that will be able to visually represent the flights and the output from 1 and 2.

ROLE ASSIGNMENT

- Hanna:
  - Data transformation:
    - Data import and cleaning
    - Data analysis
    - Data manipulation
  - Algorithm implementation
    - Adjacency matrix
    - Load adjacency matrix
    - Breadth First Search algorithm implementation
    - Edmond Karp algorithm implementation
    - Running the algorithm
    - Output
  - Dashboard implementation
    - Flask app initialization
    - Webpage
  - Testing
  - Final report preparation
- Randy:
  - Passenger capacity research
- Ray:
  - Passenger capacity research
- Rosemary:
  - Final report preparation
  - Final presentation preparation
- Tim:
  - Passenger capacity research
  - Testing (identified and fixed one bug)

DATA TRANSFORMATION

○ DATA IMPORT AND CLEANING

The project commenced with the transformation of the `airlines.txt` file from TXT to CSV format, facilitating universal import. We employed the Pandas library for importing each CSV file and converting it into a data frame. Additionally, we added a 'capacity' column that contains information related to passenger capacity of each plane model. This information was retrieved from seatguru.com website. During the import process, we noticed some column names contained leading spaces, which we addressed by setting the `skipinitialspace=True` parameter.

```
# reading the CSV files
# some column names had empty spaces in the beginning of the name, "skipinitialspace=True" parameter removes
# empty spaces.
routes_data = pd.read_csv('../data_sets/routes.csv', encoding='ISO-8859-1', skipinitialspace=True)
```

The `airports.csv` file initially lacked column names; therefore, we assigned these manually, guided by the data variable in each column. Additionally, we standardized missing values across all data frames by replacing '\N' with NaN.

○ DATA ANALYSIS

Comprehensive analysis was conducted on each file to establish relationships between the data frames. Below is a description of each data frames and its key components:

1. routes.csv:
   - `airline`: 2-letter or 3-letter airline code.
   - `airline ID`: Unique identifier for the airline.
   - `source airport`: Code of the source airport.
   - `destination airport`: Code of the destination airport.
   - `codeshare`: "Y" for codeshare flights, otherwise empty.
   - `stops`: Number of stops ("0" for direct flights).
   - `equipment`: Codes for plane types used.

2. airlines.csv:
   - `Airline ID`: Unique identifier for the airline.
   - `Name`: Airline name.
   - `Alias`: Airline alias.
   - `IATA`: 2-letter IATA code.
   - `ICAO`: 3-letter ICAO code.
   - `Callsign`: Airline callsign.
   - `Country`: Country of incorporation.
   - `Active`: "Y" if operational, "N" if defunct.

3. airports-extended.csv:
   - `Airport ID`: Unique identifier for the airport.

- `Name`: Airport name.
- `City`: City served by the airport.
- `Country`: Country of the airport.
- `IATA/ICAO`: Airport codes.
- `Latitude/Longitude`: Geographical coordinates.
- `Altitude`: Altitude in feet.
- `Timezone`: UTC offset.
- `DST`: Daylight savings time category.
- `Type`: Type of the airport (e.g., airport, station).
- `Source`: Data source (OurAirports, Legacy, User).

4. planes.csv:
   - `aircraft`: Plane model.
   - `IATA/ICAO`: Aircraft codes.
   - `capacity`: Seating capacity.

○ DATA MANIPULATION

We refined our dataset by removing inactive airlines and filtering the routes data frame to retain only flights with a maximum of one layover. The data frames for routes, airports, airlines, and planes were then merged, selecting relevant columns into a single, comprehensive data frame. Key merging steps included:

- Merging the airports data frame into routes based on airport ID, adding city, latitude, and longitude data.
- Joining the airlines data frame with routes using the airline ID column to obtain airline names.
- Combining the planes data frame with routes to match plane models using equipment codes from routes and IATA codes from planes.

Unnecessary columns such as airline, airline ID, codeshare, stops, and equipment were dropped to simplify the final data frame, eliminating redundancy, and reducing data complexity. The passenger capacity column was converted to an integer type for ease of calculations. The source and destination city columns were standardized to lowercase for consistency in data manipulation and identification.

```
# Haversine function to calculate distance
anutabutsko
def haversine(lat1, lon1, lat2, lon2):
    """
    Calculate distance between two points on earth based on latitude and longitude
    """
    # Convert decimal degrees to radians
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])

    # Haversine formula
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat / 2) ** 2 + cos(lat1) * cos(lat2) * sin(dlon / 2) ** 2
    c = 2 * asin(sqrt(a))
    r = 3956

    return round(c * r)
```

An additional step included calculating the distance in miles between source and destination cities using the haversine formula, leveraging the latitude and longitude data. The resulting comprehensive data frame was then saved as a CSV file for subsequent use in the project.

ALGORITHM IMPLEMENTATION:

○ INTRODUCTION

We decided to use the Edmonds-Karp maximum flow algorithm for this project, which is a specific implementation of the Ford-Fulkerson method. The Ford-Fulkerson method is a common technique and is very useful for solving maximum flow problems. The idea of this algorithm is to repeatedly find augmenting paths from the source to the destination in the flow graph, augment the flow, and repeat until no more paths exist. The Ford-Fulkerson algorithm, using a Depth-First Search (DFS) to find augmenting paths, takes O(Ef) time, where E is the number of edges and f is the maximum flow. However, a pitfall of this algorithm is that its time complexity depends on the capacity values of the edges in the graph. This is because DFS picks edges to traverse in such a way that we might only be able to push one unit of flow in each iteration, which is not optimal for our time complexity. Although this scenario is highly unlikely to happen in practice, it is something we want to avoid.

The reason why we decided to use the Edmonds-Karp algorithm is that, instead of using DFS to find augmenting paths, it uses Breadth-First Search (BFS), which offers better time complexity, O(VE^2) where V is the number of vertices. At first glance, this might not seem like an improvement, but in practice, it is significantly more efficient. The major difference in this approach is that it no longer depends on the capacity value of any edge in the flow graph, which is crucial. The Edmonds-Karp algorithm can also be thought of as a method of augmentation, which repeatedly finds the shortest augmenting path from source to destination in terms of the number of edges used in each iteration. Using BFS to find augmenting paths ensures that the shortest path is found in every iteration.

○ ADJACENCY MATRIX

The first step in our algorithm implementation involved creating an adjacency matrix from the final, filtered flights dataset. Our file, ‘adjacency_matrix.py’, contains a class named Graph that initializes the matrix and includes two methods: one for adding an edge to the graph, and the other for retrieving edges. The class is initialized with a list of vertices, representing the city names in our flight network. Given our extensive dataset, which required us to account for numerous flights and connections, we had to devise an innovative solution to organize our matrix.

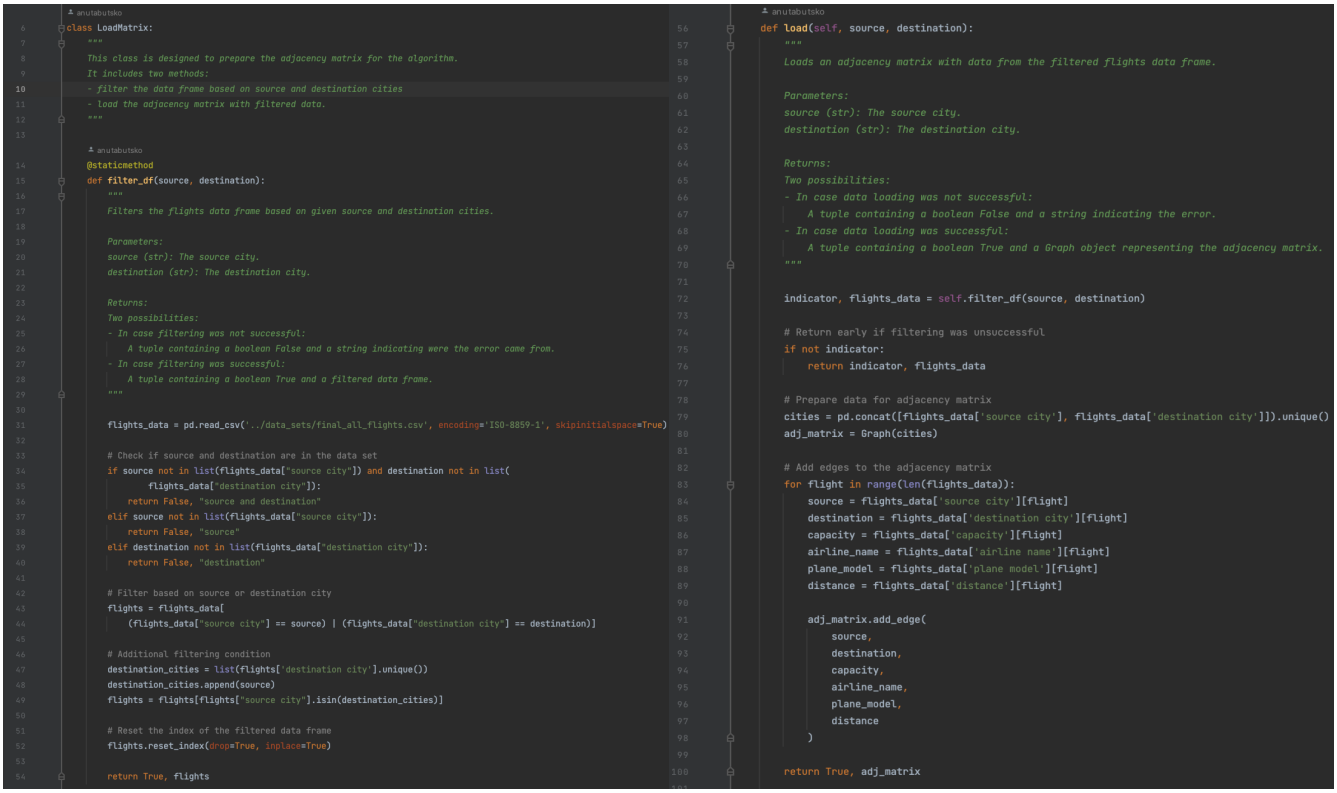
```python
class Graph:
    def __init__(self, vertices):
        """
        Initializes the graph with the given vertices.

        Parameters:
        vertices (list): A list of vertices (city names) in the graph.
        """

        # A dictionary mapping each vertex to its index in the adjacency matrix.
        self.vertex_index = {node: i for i, node in enumerate(vertices)}
        # A dictionary mapping each index in the adjacency matrix to the vertex name.
        self.vertex_city = {i: node for i, node in enumerate(vertices)}

        # Initialize the graph as a matrix of empty lists
        n = len(vertices)
        self.graph = [[[] for _ in range(n)] for _ in range(n)]

    def add_edge(self, source, destination, capacity, airline_name, plane_model, distance):
        """
        Adds an edge to the graph between two vertices (cities) with specified attributes.

        Parameters:
        source (str): The source vertex of the edge.
        destination (str): The destination vertex of the edge.
        capacity (int): The capacity of the edge.
        airline_name (str): The name of the airline for this edge.
        plane_model (str): The model of the plane for this edge.
        distance (int): The distance between source and destination for this edge.
        """

        # Add an edge with attributes between source and destination vertices
        source_idx = self.vertex_index[source]
        destination_idx = self.vertex_index[destination]
        weight_info = {'capacity': capacity,
                       'airline name': airline_name,
                       'plane model': plane_model,
                       'distance': distance}
        self.graph[source_idx][destination_idx].append(weight_info)

    def get_edges(self, source, destination):
        """
        Retrieves all edges between two specified vertices (cities).

        Parameters:
        source (str): The source vertex.
        destination (str): The destination vertex.

        Returns:
        list: A list of dictionaries, each representing an edge with attributes from the source to the destination.
        """

        source_idx = self.vertex_index[source]
        destination_idx = self.vertex_index[destination]

        return self.graph[source_idx][destination_idx]
```

Considering the multiple flights available between the same two cities, each operated by different airlines with various plane models and consequently differing capacities, we needed a systematic way to organize the matrix. It was essential to keep it both organized and readable, without losing critical information about each city connection. Therefore, instead of using integers as weights in the graph, we opted for Python lists that contained all necessary information. Consequently, an edge between city A and city B would have a list, encompassing all potential flights between these two cities. Each flight's relevant details are encapsulated in a dictionary within this list.

For each edge, the weight is a list of dictionaries, with each dictionary representing a possible flight between the two vertices. Every dictionary includes comprehensive information about the specific flight: the capacity of each flight, plane model, carrier, and the distance between the cities. This structure was chosen because we aimed to convey more than just the flight capacity. Ultimately, our goal was to provide a detailed report on each flight, including specifics about the airline, plane model, etc., to offer a complete picture of the passenger transport options from city A to city B.

○ LOAD ADJACENCY MATRIX

The second step involved loading data into the matrix from the final “flights” dataset. However, due to the large size of the file and the extensive amount of information it contained, we first needed to trim the data. Our initial attempt to load the entire final data frame into the matrix resulted in excessively long processing times when running the algorithm, necessitating a more efficient approach.



```python
class LoadMatrix:
    """
    This class is designed to prepare the adjacency matrix for the algorithm.
    It includes two methods:
    - filter the data frame based on source and destination cities
    - load the adjacency matrix with filtered data.
    """

    @staticmethod
    def filter_df(source, destination):
        """
        Filters the flights data frame based on given source and destination cities.

        Parameters:
        source (str): The source city.
        destination (str): The destination city.

        Returns:
        Two possibilities:
        - In case filtering was not successful:
            A tuple containing a boolean False and a string indicating were the error came from.
        - In case filtering was successful:
            A tuple containing a boolean True and a filtered data frame.
        """

        flights_data = pd.read_csv('../data_sets/final_all_flights.csv', encoding='ISO-8859-1', skipinitialspace=True)

        # Check if source and destination are in the data set
        if source not in list(flights_data["source city"]) and destination not in list(
                flights_data["destination city"]):
            return False, "source and destination"
        elif source not in list(flights_data["source city"]):
            return False, "source"
        elif destination not in list(flights_data["destination city"]):
            return False, "destination"

        # Filter based on source or destination city
        flights = flights_data[
            (flights_data["source city"] == source) | (flights_data["destination city"] == destination)]

        # Additional filtering condition
        destination_cities = list(flights['destination city'].unique())
        destination_cities.append(source)
        flights = flights[flights["source city"].isin(destination_cities)]

        # Reset the index of the filtered data frame
        flights.reset_index(drop=True, inplace=True)

        return True, flights
```

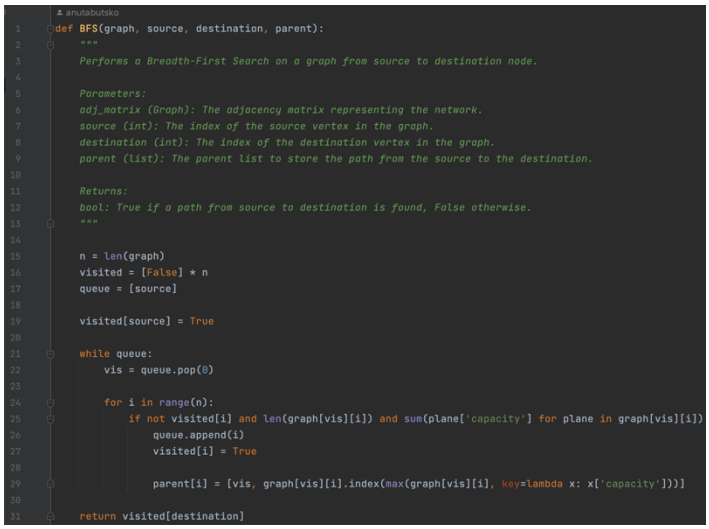
```python
def load(self, source, destination):
    """
    Loads an adjacency matrix with data from the filtered flights data frame.

    Parameters:
    source (str): The source city.
    destination (str): The destination city.

    Returns:
    Two possibilities:
    - In case data loading was not successful:
        A tuple containing a boolean False and a string indicating the error.
    - In case data loading was successful:
        A tuple containing a boolean True and a Graph object representing the adjacency matrix.
    """

    indicator, flights_data = self.filter_df(source, destination)

    # Return early if filtering was unsuccessful
    if not indicator:
        return indicator, flights_data

    # Prepare data for adjacency matrix
    cities = pd.concat([flights_data['source city'], flights_data['destination city']]).unique()
    adj_matrix = Graph(cities)

    # Add edges to the adjacency matrix
    for flight in range(len(flights_data)):
        source = flights_data['source city'][flight]
        destination = flights_data['destination city'][flight]
        capacity = flights_data['capacity'][flight]
        airline_name = flights_data['airline name'][flight]
        plane_model = flights_data['plane model'][flight]
        distance = flights_data['distance'][flight]

        adj_matrix.add_edge(
            source,
            destination,
            capacity,
            airline_name,
            plane_model,
            distance
        )

    return True, adj_matrix
```

We decided to refine the dataframe based on user input for source and destination cities. This meant filtering the matrix to include only those flights that matched the user's input for the source city and filtering by destination flights that matched the user's destination city input. To further reduce the size of our dataframe, we took the list of all destination flights that could be reached from the user's chosen source city and mapped it to the list of source flights that could reach the user's chosen destination city. We retained only those flights that matched between these two lists, effectively eliminating flights that could never reach the user's destination of interest from their chosen source.

For instance, if a user plans to travel from New York to San Francisco, and there's a flight from New York to Moscow but no available flight from Moscow to San Francisco, we will disregard the New York to Moscow flight. This is because it could never ultimately transport the user to their desired destination. By applying this method of data trimming, we managed to significantly streamline our dataset, making the algorithm more efficient and focused on relevant flight paths.

○ BREADTH FIRST SEARCH ALGORITHM IMPLEMENTATION

Having previously discussed our rationale for choosing the Breadth-First Search (BFS) algorithm over the Depth-First Search (DFS), let's delve into the specifics of the BFS implementation. The key functionality is encapsulated in the following code segment:



```
anutabutsko
def BFS(graph, source, destination, parent):
    """
    Performs a Breadth-First Search on a graph from source to destination node.

    Parameters:
    adj_matrix (Graph): The adjacency matrix representing the network.
    source (int): The index of the source vertex in the graph.
    destination (int): The index of the destination vertex in the graph.
    parent (list): The parent list to store the path from the source to the destination.

    Returns:
    bool: True if a path from source to destination is found, False otherwise.
    """

    n = len(graph)
    visited = [False] * n
    queue = [source]

    visited[source] = True

    while queue:
        vis = queue.pop(0)

        for i in range(n):
            if not visited[i] and len(graph[vis][i]) and sum(plane['capacity'] for plane in graph[vis][i])
                queue.append(i)
                visited[i] = True

                parent[i] = [vis, graph[vis][i].index(max(graph[vis][i], key=lambda x: x['capacity']))]

    return visited[destination]
```

This BFS implementation explores the graph from a source to a destination vertex. It uses a visited list to track explored vertices and a queue to manage the traversal order. The algorithm iterates through the graph, checking each vertex and its connections. For each connection, if it's unvisited and has a non-empty, positive capacity, the vertex is marked as visited and added to the queue. The parent list keeps track of the path, storing the predecessor vertex and the flight with maximum capacity. The function returns True if a path to the destination exists, otherwise False.

```
class MaxCapacity:
    """
    This class is designed to analyze the capacity of flight paths using the Edmond-Karp algorithm.
    It operates on an adjacency matrix representation of flight routes and capacities.
    """
    anutabutsko
    def edmond_karp(self, adj_matrix, source_city, destination_city):
        """
        Executes the Edmond-Karp algorithm to find the maximum flow (capacity) between two cities.

        Parameters:
        adj_matrix (Graph): The adjacency matrix representing the network.
        source_city (str): The source city.
        destination_city (str): The destination city.

        Returns:
        A tuple containing two parameters:
        - list of dictionaries with details about each connection between vertices (cities):
            layover city
            layover airline
            layover plane model
            layover capacity
            layover distance
            destination city
            destination airline
            destination plane model
            destination capacity
            destination distance
            maximum capacity
        - total maximum capacity.
        """

        result = []
        graph = adj_matrix.graph
        source_idx = adj_matrix.vertex_index[source_city]
        destination_idx = adj_matrix.vertex_index[destination_city]
        n = len(graph)
        parent = [[-1, -1] for _ in range(n)]
        max_capacity = 0

        while BFS(graph, source_idx, destination_idx, parent):
            current_capacity = float("Inf")
            curr_node_index = destination_idx
            report = {'source city': source_city.title()}

            while curr_node_index != source_idx:
                parent_node_index, edge_id = parent[curr_node_index]
                current_capacity = min(current_capacity, graph[parent_node_index][curr_node_index][edge_id]['capacity'])
                capacity, airline_name, plane_model, distance = self.extract_flight_info(graph, parent_node_index,
                                                                                          curr_node_index, edge_id)

                if adj_matrix.vertex_city[parent_node_index] != source_city:
                    report.update(self.create_layover_report(adj_matrix,
                                                             parent_node_index,
                                                             capacity,
                                                             airline_name,
                                                             plane_model,
                                                             distance))

                curr_node_index = parent_node_index

            report.update(self.create_destination_report(destination_city,
                                                         capacity,
                                                         airline_name,
                                                         plane_model,
                                                         current_capacity,
                                                         distance))
            result.append(report)
            # print(current_capacity)
            max_capacity += current_capacity
            self.update_graph_capacity(graph, parent, source_idx, destination_idx, current_capacity)
```

○ EDMOND KARP ALGORITHM IMPLEMENTATION

Following our foundational work on the data transformation and BFS implementation, we proceeded to the main part of our project: implementing the MaxCapacity class, which leverages the Edmond-Karp algorithm to analyze flight path capacities.

The MaxCapacity class is a cornerstone of our project, designed to analyze the maximum flow or capacity of flight paths using the Edmond-Karp algorithm. This algorithm operates on an adjacency matrix representation of our flight routes and capacities.

The `edmond_karp` method within this class executes the algorithm to find the maximum flow (capacity) between two cities, represented as vertices in our graph. It returns a detailed report of the connections between cities and the total maximum capacity. The report includes each connection's layover and destination city, airline, plane model, capacity, and distance, along with the maximum capacity found.

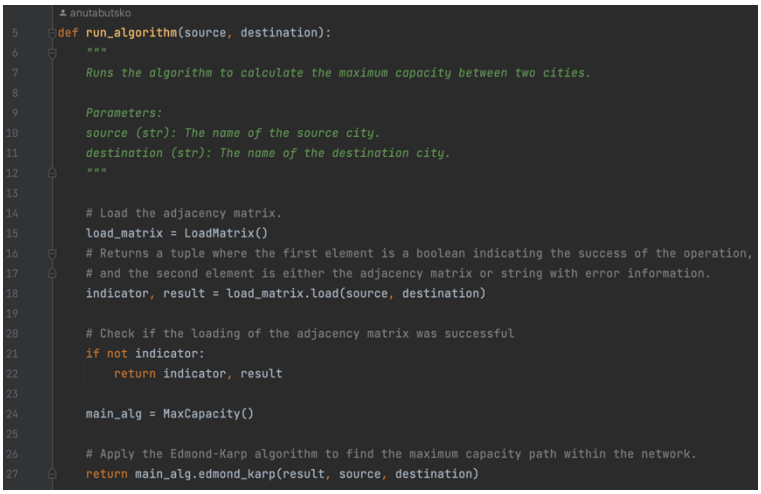
In each iteration, the BFS function identifies a path with available capacity from the source to the destination. We then calculate the minimum capacity along this path, which is considered the current flow.

The `extract_flight_info` helper method gathers detailed information about each flight segment. For layovers and the final destination, the `create_layover_report` and `create_destination_report` methods generate comprehensive reports, respectively. These include details such as the airline name, plane model, and individual segment capacities.

```
anutabutsko
def extract_flight_info(self, graph, parent_node_index, curr_node_index, edge_id):
    """
    Helper method to extract flight information from the graph.
    """

    capacity = graph[parent_node_index][curr_node_index][edge_id]['capacity']
    airline_name = graph[parent_node_index][curr_node_index][edge_id]['airline name']
    plane_model = graph[parent_node_index][curr_node_index][edge_id]['plane model']
    distance = graph[parent_node_index][curr_node_index][edge_id]['distance']

    return capacity, airline_name, plane_model, distance

anutabutsko
def create_layover_report(self, adj_matrix, parent_node_index, capacity, airline_name, plane_model, distance):
    """
    Helper method to create a layover report.
    """

    return {
        'layover': adj_matrix.vertex_city[parent_node_index].title(),
        'layover airline': airline_name,
        'layover model': plane_model,
        'layover capacity': capacity,
        'layover distance': distance,
    }

anutabutsko
def create_destination_report(self, destination_city, capacity, airline_name, plane_model, current_capacity,
                              distance):
    """
    Helper method to create a destination report.
    """

    return {
        'destination city': destination_city.title(),
        'destination airline': airline_name,
        'destination model': plane_model,
        'destination capacity': capacity,
        'maximum capacity': current_capacity,
        'destination distance': distance,
    }

anutabutsko
def update_graph_capacity(self, graph, parent, source_idx, destination_idx, current_capacity):
    """
    Helper method to update the graph capacities after each iteration of the algorithm.
    """

    curr_node_index = destination_idx
    while curr_node_index != source_idx:
        parent_node_index, edge_id = parent[curr_node_index]
        graph[parent_node_index][curr_node_index][edge_id]['capacity'] -= current_capacity
        reverse_edge = {'capacity': current_capacity}
        graph[curr_node_index][parent_node_index].append(reverse_edge)
        curr_node_index = parent_node_index
```

The `update_graph_capacity` method is critical for adjusting the capacities in the graph after each flow augmentation. This step ensures that the algorithm accounts for the current flow in the graph, thus updating the remaining capacities and adding reverse edges to represent the possibility of flow adjustment in future iterations.

By implementing the MaxCapacity class and the Edmond-Karp algorithm, we could analyze complex flight network data, determining the most efficient routes and capacities between any two given cities. This approach has provided us with a powerful tool for understanding and optimizing flight route capacities, aligning with our project's overarching goals.



```
anutabutsko
def run_algorithm(source, destination):
    """
    Runs the algorithm to calculate the maximum capacity between two cities.

    Parameters:
    source (str): The name of the source city.
    destination (str): The name of the destination city.
    """

    # Load the adjacency matrix.
    load_matrix = LoadMatrix()
    # Returns a tuple where the first element is a boolean indicating the success of the operation,
    # and the second element is either the adjacency matrix or string with error information.
    indicator, result = load_matrix.load(source, destination)

    # Check if the loading of the adjacency matrix was successful
    if not indicator:
        return indicator, result

    main_alg = MaxCapacity()

    # Apply the Edmond-Karp algorithm to find the maximum capacity path within the network.
    return main_alg.edmond_karp(result, source, destination)
```

RUNNING THE ALGORITHM

The ‘run_algorithm()’ function, located within the main.py file, serves as the primary driver for executing the Edmond-Karp algorithm.

The function accepts two parameters: source and destination, which are the names of the source and destination cities, respectively. The first step in this function is to load the adjacency matrix, which represents our network of flight routes and capacities.

Once the adjacency matrix is successfully loaded, the function instantiates the MaxCapacity class and then calls its ‘edmond_karp’ method, passing the loaded adjacency matrix, along with the source and destination cities.

The ‘run_algorithm()’ function is a crucial component of our system, orchestrating the overall process of analyzing flight path capacities. It seamlessly integrates the data loading process with the core algorithmic analysis, providing an effective and user-friendly way to compute the optimal paths and capacities between any two cities in our flight network.

○ OUTPUT

The algorithm outputs a comprehensive report in the form of a list of dictionaries. Each dictionary contains detailed route information between the source and destination. Let's visualize the output for possible flights from Tampa to Dubai:

Because there are so many possible flights and connections, the output looks quite complicated, however this format is useful for preserving the data and for the future output in out dashboard. Here are the first three lines of this output for better visualization:


```
{
  "source city": "Tampa",
  "layover": "London",
  "layover airline": "Emirates",
  "layover model": "Airbus A380-800",
  "layover capacity": 853,
  "layover distance": 3398,
  "destination city": "Dubai",
  "destination airline": "American Airlines",
  "destination model": "Boeing 777",
  "destination capacity": 388,
  "maximum capacity": 388,
  "destination distance": 4416
}
```

```
{
  "source city": "Tampa",
  "layover": "London",
  "layover airline": "Emirates",
  "layover model": "Airbus A380-800",
  "layover capacity": 853,
  "layover distance": 3414,
  "destination city": "Dubai",
  "destination airline": "Finnair",
  "destination model": "Boeing 777",
  "destination capacity": 388,
  "maximum capacity": 388,
  "destination distance": 4416
}
```

```
{
  "source city": "Tampa",
  "layover": "London",
  "layover airline": "Qantas",
  "layover model": "Airbus A380-800",
  "layover capacity": 853,
  "layover distance": 3398,
  "destination city": "Dubai",
  "destination airline": "British Airways",
  "destination model": "Boeing 777",
  "destination capacity": 388,
  "maximum capacity": 388,
  "destination distance": 4416
}
```

DASHBOARD IMPLEMENTATION

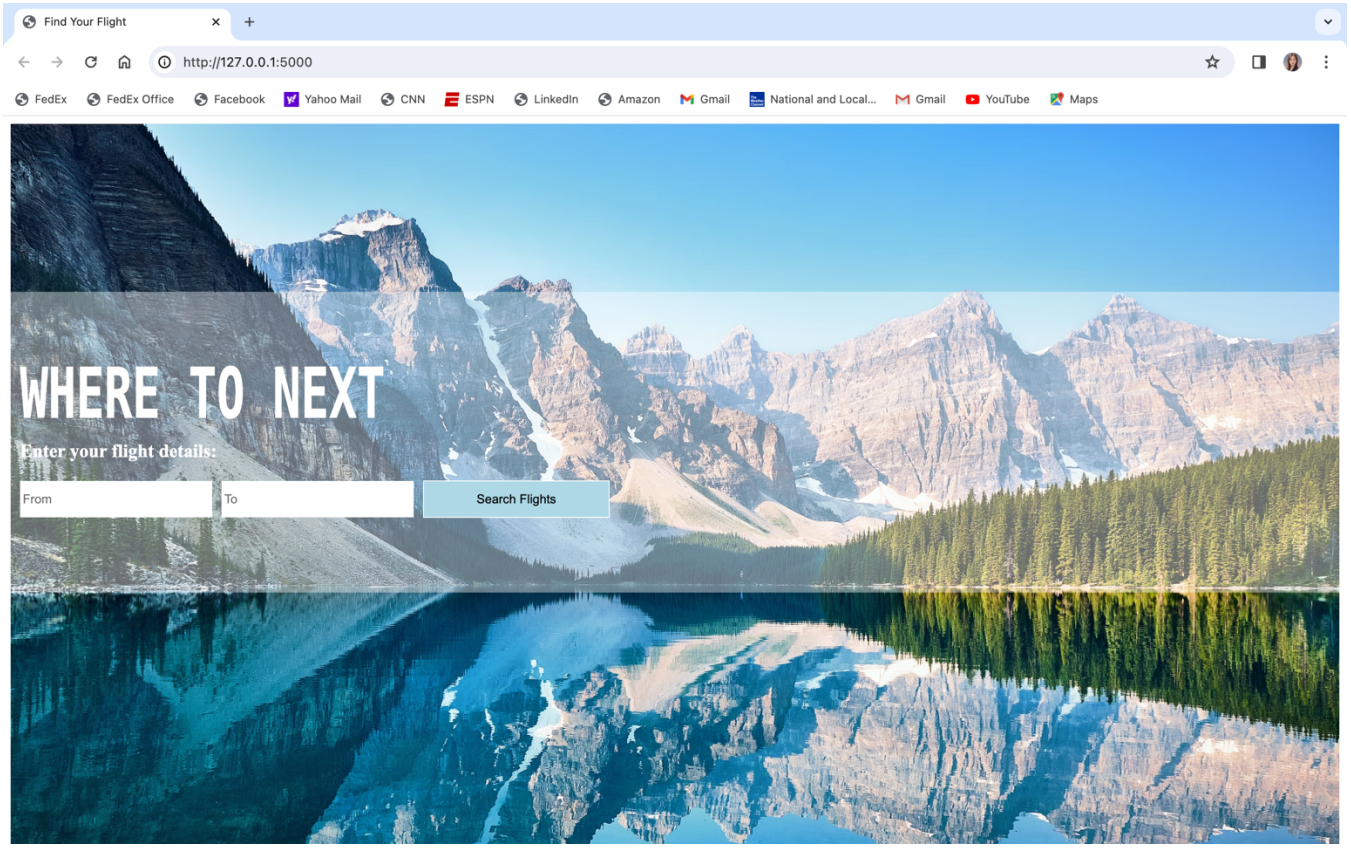
○ FLASK APP INITIALIZATION

For our dashboard, we developed a web service application using Flask. The server directs users to the home page, where they are prompted to input their desired source and destination cities. Upon submission, the application uses a GET method to request the results and then redirects the user to a new page called 'result'. This page employs a POST method to display all available flights based on the user's inputs.

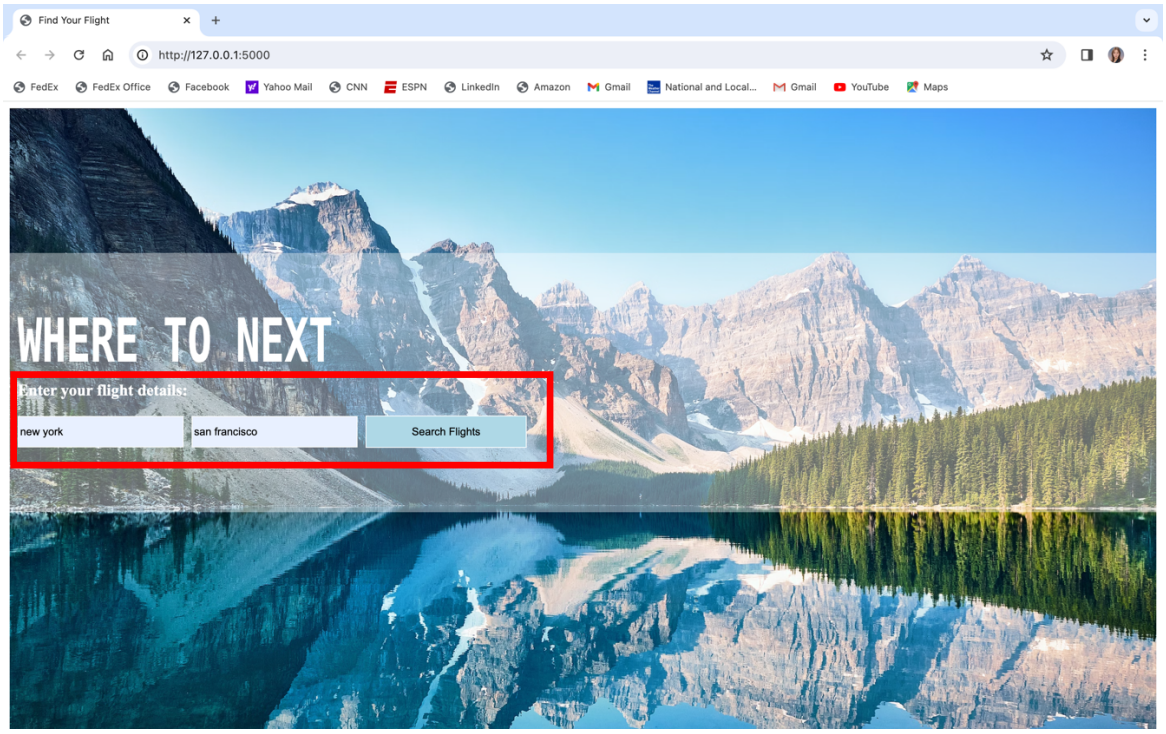
```python
app = Flask(__name__)


# Home page
anutabutsko
@app.route("/", methods=["GET"])
def home():
    # Render the home page for GET requests
    return render_template("home.html")  # home.html is the page with the input form


# Results page
anutabutsko +1
@app.route("/result", methods=["POST"])
def result():
    source = request.form['source']
    destination = request.form['destination']
    source = source.lower()
    destination = destination.lower()
```

○ WEBPAGE

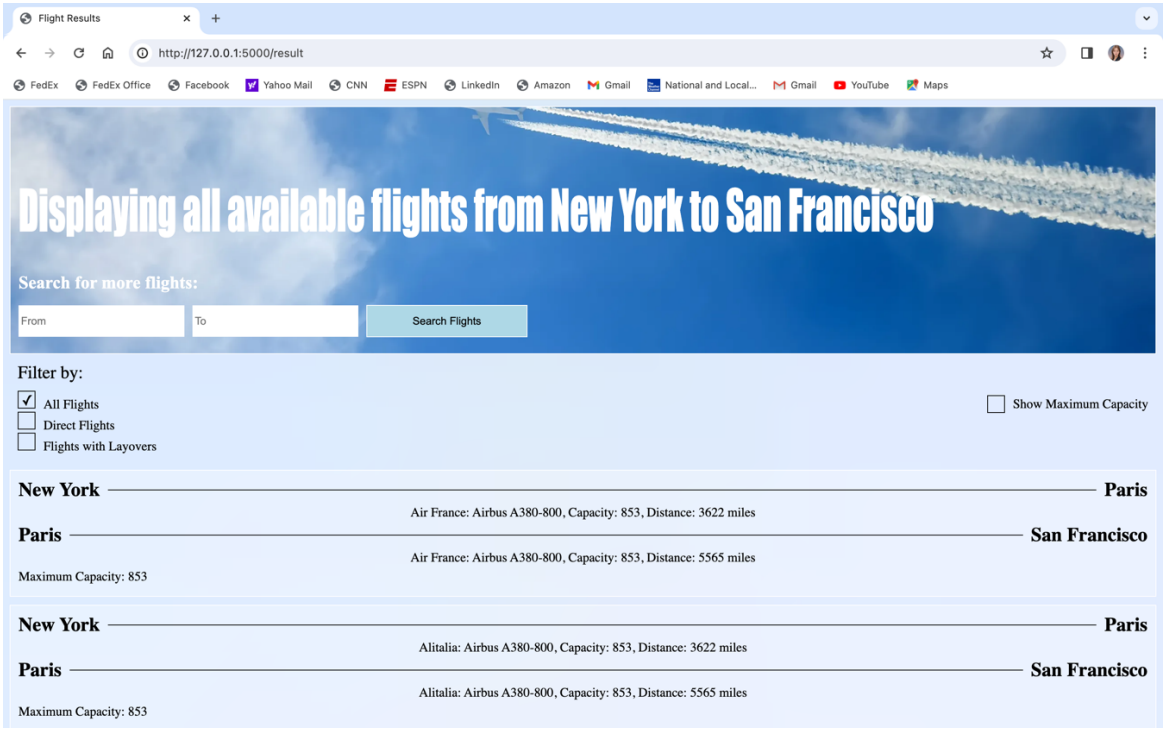
When the Flask app is started, it will launch a local host homepage designed as follows:



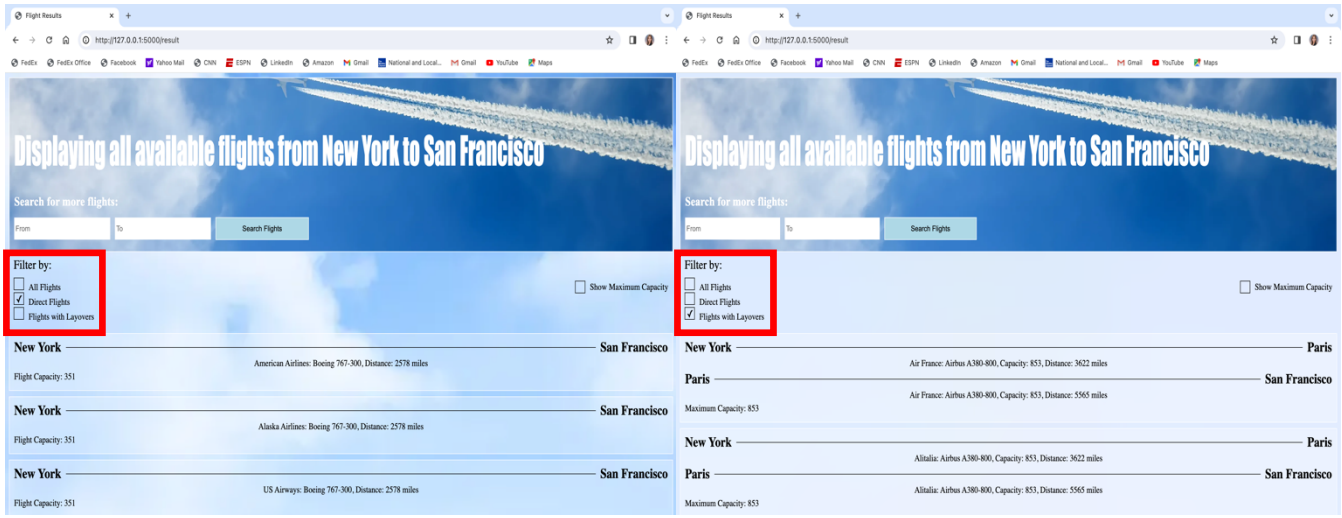
It prompts the user to enter source and destination city names, and then redirects them to a new results page displaying the algorithm's output. Let us input “new york” for the source city and “san francisco” for the destination city:



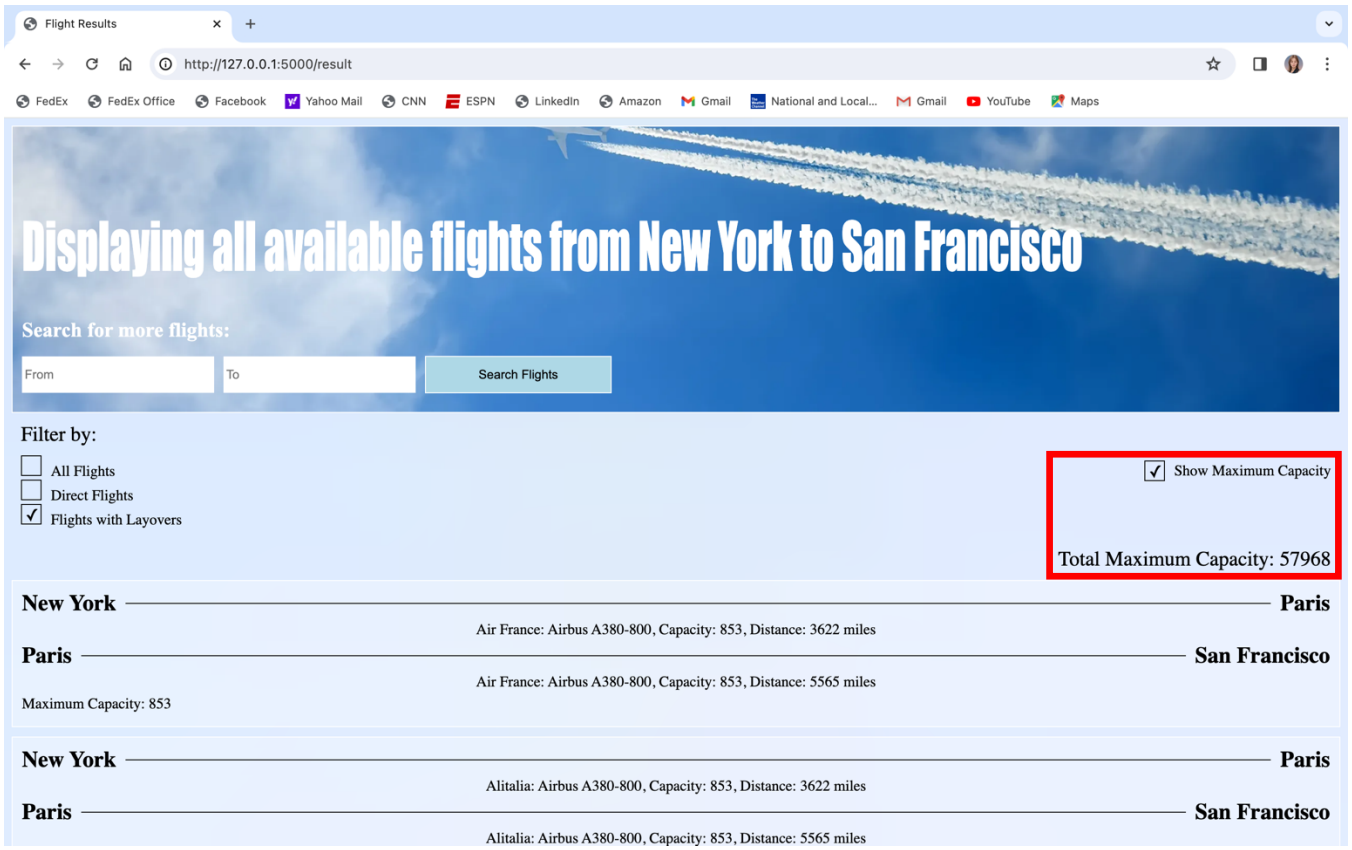
The output looks as follows:



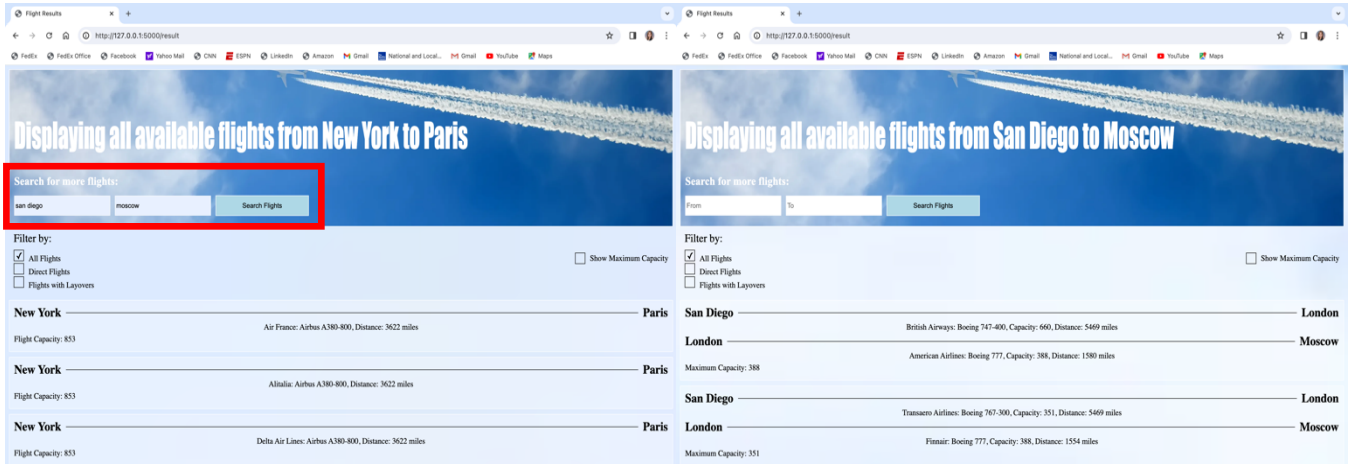
On this page, the user can scroll through all available flights for the source and destination cities selected. Moreover, the user has the option to sort the flights based on whether they are direct or have layovers.



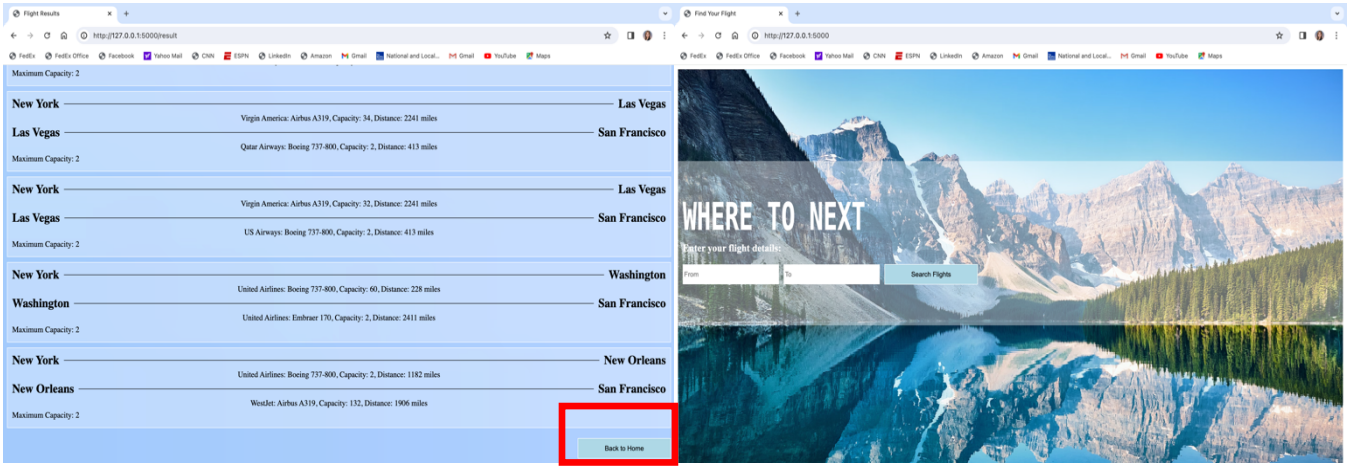
For the final feature, users can select the 'Show Maximum Capacity' option to visualize the maximum capacity between the two cities they have chosen. This functionality aligns with the primary objective of our project: to efficiently determine and display the maximum capacity for flights between selected destinations.



From this point, users are presented with two options: they can either input new source and destination cities to generate additional flights of interest while remaining on the same page,



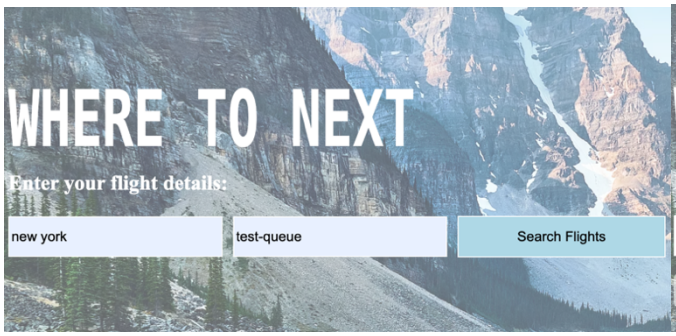
or they can scroll down to the bottom of the results page and click the 'Back Home' button to return to the home page.

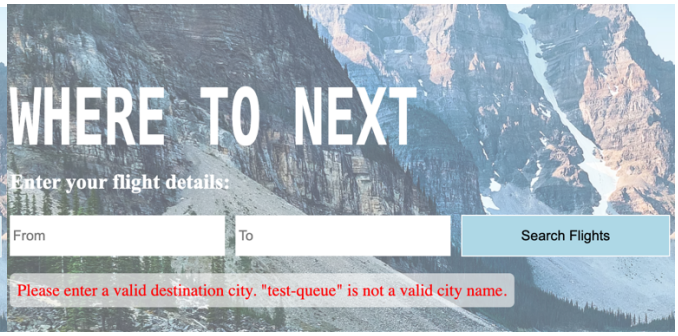


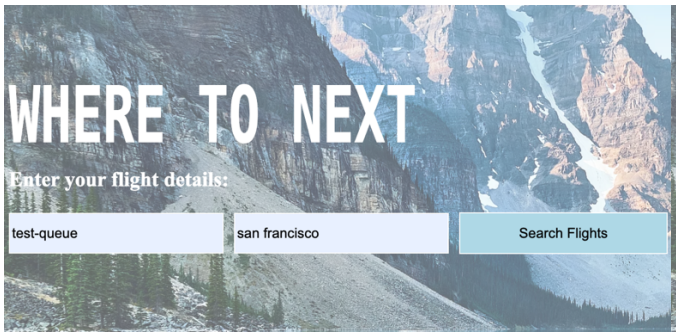
TESTING

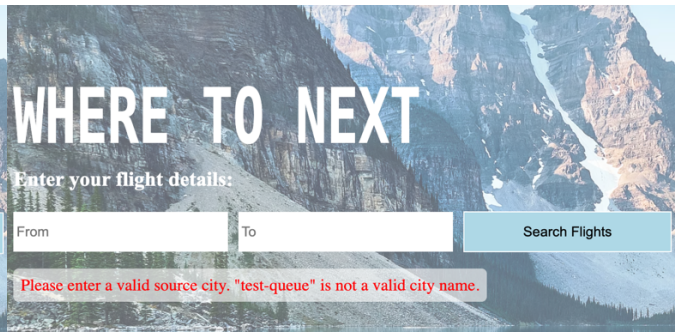
The algorithm underwent extensive testing through various methods, ensuring its robustness and efficiency. In the following section, we will delve into the key aspects we accounted for and the significant improvements we implemented as a result of these tests.

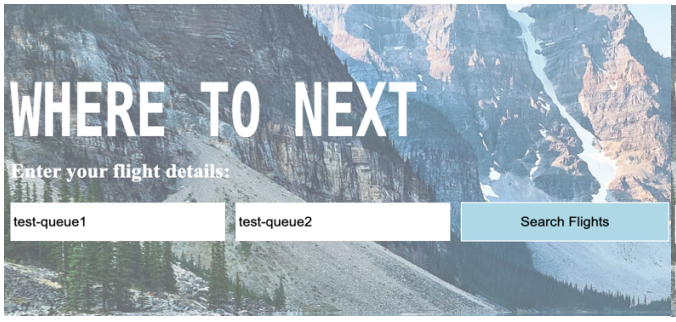
- If a user inputs a non-existent city name, they will receive an error message indicating that the city does not exist. The system will then prompt the user to input a valid city name.

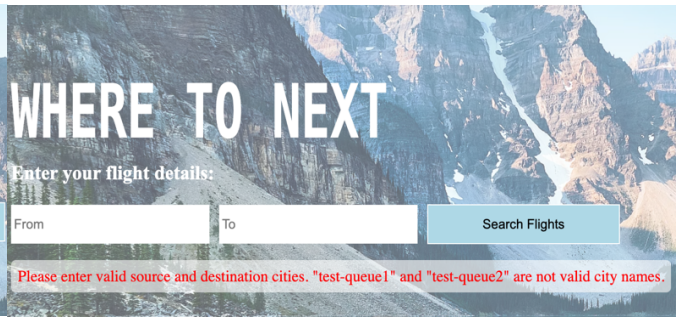




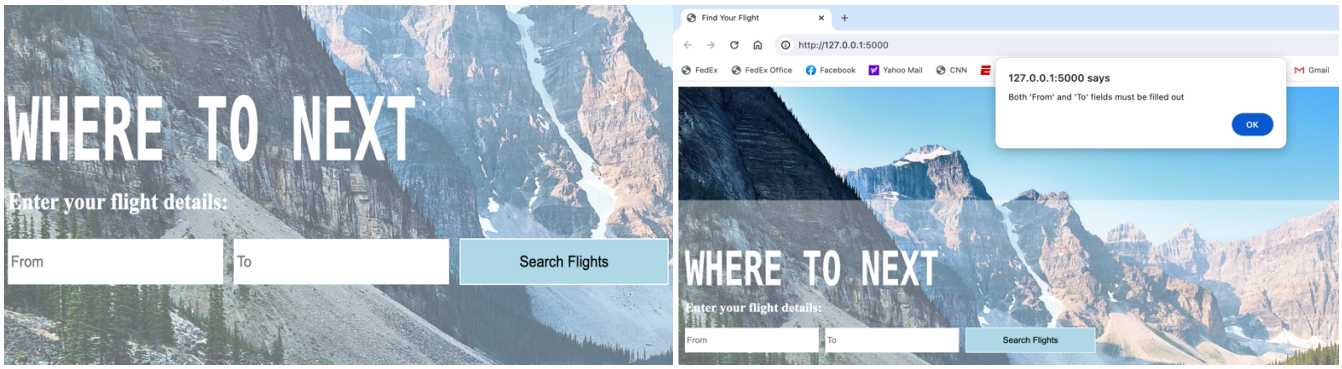




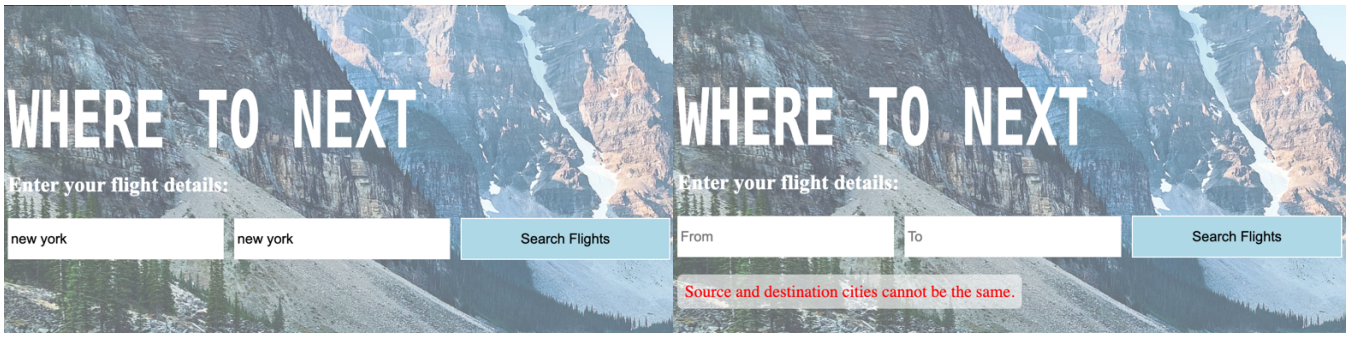




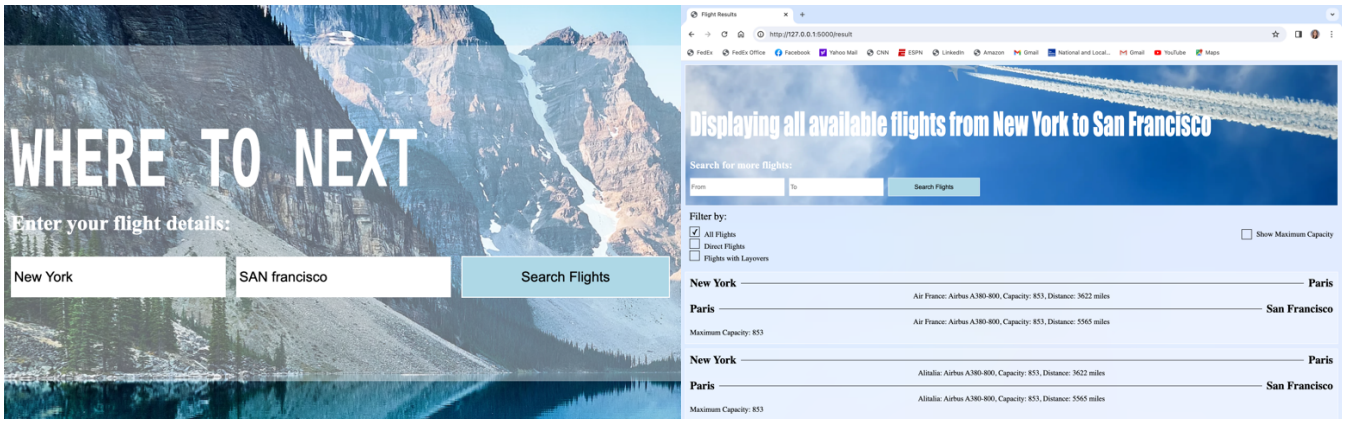
- If a user does not provide any input, an error message will pop up, prompting them to enter the required information.



- If a user inputs two cities with the same name, an error message will arise, indicating the need for distinct source and destination cities.



- The input fields are designed to accommodate city names in any text format, whether in lowercase, uppercase, or a combination of both, ensuring user convenience and flexibility.



EXTENSIONS

Possible extensions for this project:

1. *Refinement of the Adjacency Matrix Structure:*
   - Rearrange the adjacency matrix to store only capacity values and corresponding hash codes.
   - Develop a separate data structure, potentially a hash table, where detailed flight information linked to each hash code is stored, streamlining the matrix, and improving data retrieval efficiency.

2. *Integration of a Key-Value Database:*
   - Transition from CSV files to a key-value database for improved data management and performance.
   - This shift will enhance scalability, allow real-time data processing, and ensure greater data reliability and concurrency control.

3. *Web Application Scalability:*
   - Explore methods to transition the website from a local host to a public web server, thereby enhancing its accessibility and scalability.

4. *Interactive Dashboard Features:*
   - Implement interactive features on the dashboard, such as dynamic graphs or maps, to visualize flight routes and capacities more engagingly.

5. *User Preferences:*
   - Add a sorting option to display flights sorted by capacity or distance between selected cities.
   - Introduce functionality for users to set preferences, such as preferred airlines, plane models, airports, or layover cities.

6. *Algorithm Optimization and Expansion:*
   - Continuously refine the underlying algorithm for enhanced performance and accuracy.

7. *API Integration:*
   - Develop an API to retrieve flight capacity data, moving away from manual data collection.

LINKS

○ GITHUB REPOSITORY

The entire project is available on GitHub in our repository, which is open to the public. Contributions and extensions are welcome.

You can access it here: https://github.com/rbidini/Algorithms_Project_2

REFERENCES

SeatGuru by TripAdvisor LLC. 2001 – 2023. Retrieved from https://www.seatguru.com

Shubham Kumar Shukla. January 7, 2022. EDMONDS-KARP AND DINIC'S ALGORITHMS FOR MAXIMUM FLOW. Retrieved from https://www.topcoder.com/thrive/articles/edmonds-karp-and-dinics-algorithms-for-maximum-flow

William Fiset. October 19, 2018. Edmonds Karp Algorithm | Network Flow | Graph Theory. Retrieved from https://www.youtube.com/watch?v=RppuJYwlcI8