

Table of Contents

1. PROBLEM STATEMENT	2
2.METHODOLOGY.....	3
2.1 Data Structure.....	3
3.RESULT.....	5
4.ANALYSIS AND DISCUSSION	6
4.1 Time Complexity.....	6
4.2Limitations.....	7
4.3Overcoming our Limitations	8

1. PROBLEM STATEMENT

The flight booking app aims to simplify the process of finding available flights and booking flights by providing a user-friendly interface and using efficient data structures and algorithms to find all possible flight routes between airports in India.

The FlightSystem class allows users to add airports to the system with their corresponding codes, names, latitudes and longitudes. Flights between different airports can be added, specifying the origin airport, destination airport and the cost of the flight.

The main functionality of the app is to find flight paths between a given origin and destination airport. The find_all_paths method uses the BFS traversal algorithm to explore all possible flight paths, and returns a list of flight paths from the origin to the destination airport. Each flight path includes the sequence of airports to visit and the total cost of the path.

The app also provides a method to calculate the distance between two airports based on their latitude and longitude coordinates.

The problem that this app solves is the need for an efficient and convenient way to search for available flights and book flights between different airports. By using the FlightSystem class and its implementation of a linked list, the app can efficiently find all possible flight routes, allowing users to compare costs and make informed decisions when booking flights. The inclusion of distance calculation also provides additional information for users who consider distance as a factor in their flight selection.

2.METHODOLOGY

2.1 Data Structure

In this project of Flight attendance system, two main data structures are used: a linked list for storing flight information and a queue for performing breadth-first search (BFS) traversal.

I. Linked List:

The Flight class represents a linked-list, where, each node contains the information about a flight's destination and the cost. The linked list data type is used to store multiple flights from a source airport. It allows efficient insertion of new flights at the end, as well as traversing in the order in which they were added.

Advantages:

- **Dynamic Size:** Linked lists have a dynamic size, allowing for efficient insertion and removal of elements at any position in the list.
- **Easy Modification:** It is relatively easy to modify the linked list structure by adding or removing nodes, making it suitable for managing flight information that can change frequently.
- **Efficient Memory Utilization:** Linked lists utilize memory efficiently by allocating memory for each.

Disadvantages:

- **Sequential Access:** Unlike arrays, linked lists do not provide direct access to elements at arbitrary positions. To access a specific node, the list needs to be traversed from the beginning.
- **Memory Overhead:** Linked lists require additional memory allocation for storing the pointers/references to the next nodes, which can result in increased memory overhead.
- **Slower Search:** Searching for an element in a linked list is less efficient compared to arrays, as it requires traversing the list from the beginning until the desired element is found.

II. Queue:

The queue data type is used to perform the breadth-first search (BFS) traversal of flight paths in the 'find_all_paths' method. The queue data structure follows the FIFO (First In First Out) principle, which is suitable for BFS algorithms. It allows efficient adding of the paths to the end of the queue and retrieval of the paths from the front.

Advantages:

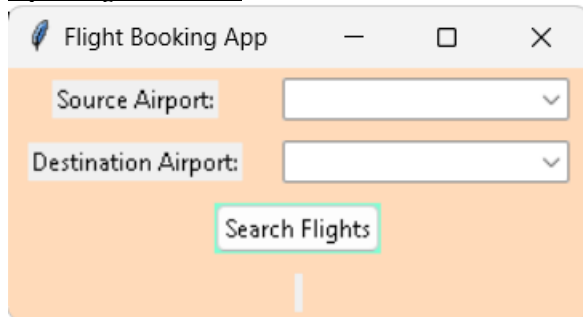
- **FIFO Behavior:** The queue follows the First-In-First-Out (FIFO) principle, ensuring that elements are processed in the same order they were inserted. This property is crucial for performing BFS traversal, as it explores nodes in a level-by-level manner.
- **Efficient Insertion and Removal:** Queue operations, such as enqueue and dequeue, have a time complexity of $O(1)$, making them efficient for adding and removing elements.
- **Simple Implementation:** Queues have a straightforward implementation and are widely used in various algorithms due to their simplicity.

Disadvantages:

- **Limited Access:** Unlike arrays, queues do not provide direct access to elements at arbitrary positions. The front and rear elements are accessible, but accessing other elements requires dequeuing elements until the desired element is reached.
- **Fixed Size:** In some implementations, queues may have a fixed size limit, leading to potential issues when the capacity is exceeded.
- **Inefficient Search:** Searching for an element in a queue requires dequeuing elements until the desired element is found, which can be inefficient compared to other data structures like binary search trees.

3.RESULT

Opening Window:




Flight Booking App

Source Airport:

Destination Airport:

Search Flights

Searching flights after choosing desired Source and Destination:



Flight Booking App

Source Airport:

Destination Airport:

Search Flights

Available Flights:

Path: BLR -> BOM (Shortest distance and Least cost)
Total Cost: 2500
Total Distance: 845.3183856559485 km

Path: BLR -> DEL -> BOM
Total Cost: 10653
Total Distance: 2903.356053663414 km

Path: BLR -> MAA -> BOM
Total Cost: 5847
Total Distance: 1323.270537240802 km

Path: BLR -> HYD -> BOM
Total Cost: 6407
Total Distance: 1121.4560909730383 km

Path: BLR -> ATQ -> BOM
Total Cost: 11920
Total Distance: 3504.3267735872096 km

Path: BLR -> AMD -> BOM
Total Cost: 6599
Total Distance: 1677.024520137638 km

Path: BLR -> LKO -> BOM
Total Cost: 8344
Total Distance: 2776.5409292285767 km

Path: BLR -> PNQ -> BOM
Total Cost: 5725
Total Distance: 855.3792445541131 km

Path: BLR -> SXR -> BOM
Total Cost: 15349
Total Distance: 4044.0635326488236 km

Note that the Shortest distance and Least cost is highlighted to help users to choose the ideal travel route.

4.ANALYSIS AND DISCUSSION

4.1 Time Complexity

Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time taken for each statement in the program to complete. The Best case is the function which performs the minimum number of steps on input data of n elements. Worst case is the function which performs the maximum number of steps on input data of size n . We will be considering both these outcomes and calculate the overall time complexity of our program.

Let us look at the time complexity of our program by the time complexity of the given code can be analyzed by examining the complexity of each operation and loop.

- ✓ The import statements and class definitions have constant time complexity.

FlightSystem class methods:

- ✓ `add_airport`: Adding an airport to the airports dictionary has an average time complexity of $O(1)$ because it uses a dictionary for constant-time lookup.
- ✓ `add_flight`: Adding a flight to the flights dictionary has an average time complexity of $O(1)$ because it uses a dictionary for constant-time lookup.
- ✓ `FlightSystem.add_flight()`: This method adds a flight to the flight system. In the worst case, it traverses the linked list of flights for a specific source airport to find the last flight and then adds the new flight.
Therefore, the worst-case time complexity is $O(n)$, where n is the number of flights from the source airport.
In the best case, when there are no flights from the source airport, the time complexity is $O(1)$ as it simply adds the flight.
- ✓ `get_flights_from_airport`: This method retrieves the flights from a given source airport. The time complexity is $O(1)$ in the best case when there are no flights from the source airport. In the worst case, it traverses the linked list of flights for the source airport, which has a time complexity of $O(n)$, where n is the number of flights from the source airport.
- ✓ `calculate_distance`: This method calculates the distance between two airports using the Haversine formula. It performs a series of mathematical operations that have constant time complexity. Therefore, the time complexity is $O(1)$.
- ✓ `find_all_paths`: This method utilizes a breadth-first search (BFS) algorithm to find all possible paths between two airports. The time complexity depends on the number of flights and airports in the flight system. The time complexity of BFS algorithm is $O(V+E)$, since in the worst case, BFS algorithm explores every node and edge; Where $|V|$ is the number of vertices (airports) and $|E|$ is the number of edges (flights) in the flight system.
- ✓ `get_distance`: This method retrieves the distance between two airports. It internally calls the `calculate_distance()` method, which has a constant time complexity of $O(1)$. Therefore, the time complexity of this method is also $O(1)$.

FlightBookingApp class methods:

- ✓ `create_widgets``: This method initializes the flight system, adds airports, and adds flights. The time complexity of this method depends on the number of airports and flights being added. Since the number of airports and flights being added is fixed in the code, the time complexity is constant, i.e., $O(1)$.

CLASS	METHOD	CORRESPONDING TIME COMPLEXITY
FlightSystem class method	<code>add_airport</code>	$O(1)$
	<code>Add_flight</code>	$O(1)$
	<code>FlightSystem.add_flight()</code>	$O(n)$
	<code>Get_flights_from_airport</code>	$O(n)$
	<code>Calculate_distance</code>	$O(1)$
	<code>find_all_paths</code>	$O(V+E)$
FlightBookingApp class	<code>Create_widgets</code>	$O(1)$
OVERALL TIME COMPLEXITY	$O(n+V+E)$	

4.2 Limitations

The program build does have its own limitations. Few of the identified limitations are described as follows:

- ✓ Lack of Error Handling:

The code does not have strong error handling mechanisms. It assumes valid inputs and may not handle unexpected errors or edge cases gracefully. Error handling for scenarios like invalid airport codes or unavailable flights is missing.

- ✓ Single source airport:

The code is designed to find flight paths from a single source airport to a destination airport. It does not support finding paths involving multiple source airports or allowing multiple destinations.

- ✓ Limited optimization:

The code utilizes the basic Breadth-First Search (BFS) algorithm to find flight paths. While BFS is suitable for finding the shortest path, it may not be the most optimized algorithm for larger datasets. Implementing more advanced algorithms like Dijkstra's algorithm could improve performance.

- ✓ Lack of user interface:

The code lacks a user interface or interaction mechanism. It does not provide a way for users to input queries or view results directly. Integration with a graphical or command-line interface would enhance usability.

4.3 Overcoming our Limitations

The ways to tackle the found limitations in such a way where the code could be improved to handle real time data to transform the project into a more comprehensive and user-friendly flight path finder, providing a richer experience for users. Some of the ways to handle the limitations are:

- ✓ Improved error handling:

Implement robust error handling mechanisms to handle invalid inputs, missing data, or unexpected errors. This can involve input validation, exception handling, and appropriate error messages to guide the user.

- ✓ Support for multiple sources and destinations:

Modify the code to handle multiple source airports and multiple destination airports. This could enable users to find flight paths involving multiple cities or plan complex itineraries.

- ✓ Advanced pathfinding algorithms:

Explore more advanced pathfinding algorithms like Dijkstra's algorithm or A* search to optimize the search for flight paths. These algorithms can provide more efficient and optimized results, especially for larger datasets.

- ✓ User interface enhancements:

Develop a user interface that allows users to interact with the program more easily. This can include input forms for selecting airports, date selection for flights, and a visually appealing representation of the flight paths.