NED UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science & Information Technology

Data Structure & Algorithm – CT-159 FSCS- Fall 2025

Group members:

- **Fozan Ahmad Khan (CT-24096) SEC - B**
- **Huzaifa Jawed (CT-24077) SEC - B**
- **Saim Uz Zaman (CT-24078) SEC - B**

Flight Graph Management: Optimized Air Routes Navigation System

Final Report

1. Introduction

Air transportation networks are critical components of national and international connectivity. With the rapid expansion of cities and increasing dependency on air travel, the need for intelligent route optimization systems has grown significantly. This project, **Flight Graph Management: Optimized Air Routes Navigation System**, focuses on analyzing air routes using graph algorithms and data structures to determine the shortest and most cost-efficient flight paths between destinations.

The system loads city nodes (airports) and directed flight routes, manages them through a Binary Search Tree (BST), and computes optimized routes using Dijkstra's shortest path algorithm. This lightweight yet powerful system demonstrates the practical combination of data structures, graph theory, and algorithm design to create an efficient navigation and route planning tool.

2. Objectives of the Project

The primary objectives of the Optimized Air Routes Navigation System are:

- To develop a system capable of reading and storing flight cities (airports) efficiently.
- To use a Binary Search Tree for fast city lookup using unique IDs.
- To represent flight paths as a weighted directed graph using adjacency lists.
- To compute the minimum flight cost between any two cities using Dijkstra's algorithm.
- To allow users to inspect cities, routes, direct connections, and optimized paths.
- To maintain a history stack for tracking user actions.
- To implement a clear menu-based console interface suitable for academic and practical demonstrations.

3. System Overview

The system works by reading two text files:

cities.txt
Contains unique IDs and city names (airports).

routes.txt
Contains directed flight routes in the form:
(SourceID → DestinationID → FlightCost)

After loading data, the system constructs:

- A vector of City objects
- A Binary Search Tree for searching by ID
- An adjacency list representing all flight connections
- A priority queue for Dijkstra's algorithm
- A stack for user action history

Users can view city lists, flight routes, direct connections, and shortest-cost paths through a friendly menu interface.

4. Data Structures Used

4.1 Vector

Stores city objects containing:
● City ID
● Airport name
Also maps each city to an index for graph traversal.

4.2 Binary Search Tree (BST)

The BST enables fast searching when loading routes or when a user inputs a city ID.

Each BST node stores:
● City ID
● City name
● Vector index
● Left and right nodes

BST searching reduces lookup complexity to O(log n) average.

4.3 Adjacency List (Graph Representation)

The graph uses an adjacency list where each entry stores:
● Destination city index
● Cost of flight
This format is memory-efficient and ideal for sparse graphs like flight networks.

4.4 Priority Queue (Min-Heap)

Used in Dijkstra's algorithm to efficiently extract the next closest unvisited city.
The priority queue stores pairs in the form:
(cost, cityIndex)

4.5 Stack (History Recording)

A stack records all user actions such as:
● Viewing cities
● Viewing routes
● Checking shortest path
● Checking direct connections

Users can display or clear the history at any time.

5. Methodology

Step 1 — Load Cities

The program reads each line from *cities.txt*, extracts:
● ID
● City/Airport name
Then stores them inside a vector and inserts them into the BST.

Step 2 — Load Flight Routes

Each route contains:
● Source city ID
● Destination city ID
● Flight cost

For each route:

1. IDs are located using the BST.
2. Converted to vector indices.
3. Added into adjacency lists as weighted edges.

Step 3 — Dijkstra's Algorithm

Dijkstra's algorithm is used to compute the optimal (minimum-cost) air route.

The algorithm:

1. Sets all distances to infinity.
2. Sets source city distance to 0.
3. Repeatedly selects the city with the smallest distance.
4. Updates distances of neighbors if a cheaper route is found.
5. Stores parent nodes for path reconstruction.

The result is:
- Minimum flight cost to the destination
- Complete optimized route

Step 4 — Reconstructing the Route

Once Dijkstra determines parent links, the system:
- Backtracks from the destination
- Uses a stack temporarily
- Prints the exact sequence of cities in the correct order

Step 5 — User Menu System

Users can:

1. View all cities
2. View all routes
3. Find the optimized flight path
4. View direct air connections
5. View/Delete action history
6. Exit the program

All options are interactive and recorded in the history stack.

6. Testing and Results

The system was tested using:
- 20 unique city IDs (random two-digit IDs)
- 23 directed flight routes

Testing achievements:
- Cities loaded successfully
- BST structure built correctly
- Routes mapped accurately to new two-digit IDs
- Dijkstra's algorithm computed shortest routes reliably
- Direct connections feature worked for all cities
- Action history successfully logged user steps
- Program handled invalid input gracefully
- No crashes or logical errors occurred after extensive tests

Example scenarios showed that the system could:
- Compute accurate flight-cost-optimized routes
- Handle asymmetric routes (A → B but not B → A)
- Trace paths through intermediate cities correctly

7. Advantages of the System

1. Efficient City Lookup: BST-based ID search improves performance significantly.
2. Accurate and Optimal Route Selection: Dijkstra ensures minimum travel cost.
3. Memory Efficient: Adjacency lists reduce unnecessary memory usage.
4. Expandable Architecture: Easy to add new cities and routes.
5. User Action Tracking: History stack supports debugging and analysis.
6. Practical Concept: Can evolve into a real airline route planning tool.

8. Limitations

1. BST is not self-balancing (like AVL), affecting worst-case performance.
2. Algorithm does not support negative weights.
3. Routes are static, not user-editable in runtime.
4. Only direct routes are stored; reverse routes must be added manually.
5. Text-based interface (no GUI).

9. Future Improvements

1. Replace BST with AVL/Red-Black Tree for balanced searching.
2. Add a full graphical interface (GUI).
3. Add bidirectional route detection.
4. Implement route editor for adding/removing flights.
5. Integrate airline-specific data:
   - Duration
   - Fuel cost
   - Seat availability
6. Use A* algorithm for larger datasets.

7. Provide automated reports and summaries.

10. Conclusion

The **Flight Graph Management: Optimized Air Routes Navigation System** successfully demonstrates how graph algorithms and data structures can solve real-world transportation optimization problems. Using a Binary Search Tree for city lookup and Dijkstra's algorithm for shortest-cost routing, the system performs efficiently and reliably.

This project showcases practical usage of:
- BSTs
- Priority queues
- Adjacency lists
- Stacks
- Graph theory

The system is well-structured, extendable, and fulfills all project objectives.

11. User Interface (UI) Overview & How the System Works

(This section is polished for Word formatting and readability.)

The system uses a **menu-based console interface**, designed to be simple, clean, and easy for any user to control. After the program loads all cities and routes, the main menu appears and remains active until the user chooses to exit.

Main Menu Options

1. View All Cities
Displays every airport with its two-digit ID and name.

2. View All Routes
Lists every stored flight route in the format:
Source → Destination → Cost

3. Find Optimized Flight Path
The user enters:
- Source city ID
- Destination city ID

The program runs Dijkstra's algorithm and displays the cheapest flight route step-by-step.

4. View Direct Connections
Shows all cities directly reachable from a selected city along with the flight cost.

5. View/Delete Action History
Every action the user performs is pushed onto a stack.
Users can:
- Display the entire action history
- Clear all records with one command

6. Exit
Safely closes the system.

How the UI Works

• The menu is always visible after each operation.
• All user inputs are validated to prevent errors.
• Every action is stored in the history stack.
• Results such as routes, shortest paths, and city lists are printed neatly.
• The system is completely keyboard-controlled—no mouse needed.

Why This UI Is Effective

• Simple and beginner-friendly
• Fast, clean navigation
• Minimal typing required
• Handles errors gracefully
• Perfect for academic demonstration and practical use