Flight Route Planner: Algorithm and Design Justification

Author: Manus AI

Date: Nov 05, 2025

1. Introduction

This document provides a comprehensive justification for the design and architectural choices made in the **Flight Route Planner**, a Java 24 Maven console-based application. The project aims to solve the problem of finding optimal flight routes between airports by modeling the system as a graph and applying various search and sorting algorithms. This document outlines the rationale behind the selection of algorithms, data structures, and overall system design, along with an analysis of their performance characteristics.

2. Algorithm Choices and Reasoning

The core of this application lies in its ability to find optimal routes based on different criteria. The choice of algorithms was driven by the specific requirements of each optimization goal, balancing efficiency, complexity, and correctness.

2.1. Route Calculation Algorithms

Dijkstra's Algorithm

For finding the **cheapest**, **fastest**, and **fewest stopovers** routes, **Dijkstra's algorithm** was the ideal choice. It is a classic and highly efficient algorithm for finding the shortest path in a weighted, non-negative edge graph [1].

- **Cheapest Route**: By setting the edge weight to the `price` of each flight, Dijkstra's algorithm guarantees finding the path with the minimum cumulative

cost.

- **Fastest Route**: By using `duration` (plus a fixed stopover time) as the edge weight, the algorithm identifies the route with the minimum total travel time.
- **Fewest Stopovers**: By assigning a uniform weight of `1` to every flight, the shortest path corresponds to the route with the minimum number of flights, and therefore the fewest stopovers.

An efficient implementation using a `PriorityQueue` was chosen to ensure optimal performance, reducing the time complexity from $O(V^2)$ to $O(E + V \log V)$, where V is the number of vertices (airports) and E is the number of edges (flights).

Modified Depth-First Search (DFS)

For the **slowest route**, a standard shortest-path algorithm like Dijkstra's is unsuitable. Instead, a **modified Depth-First Search (DFS)** was implemented. The goal is to find the longest path in the graph, which is an NP-hard problem in general [2]. To make this computationally feasible, the search is constrained by a **depth limit** of 3 stopovers (4 flights). This heuristic prevents prohibitively long search times and avoids cycles while still providing a meaningful "slowest" route for practical purposes.

2.2. Sorting Algorithms

The application implements two distinct sorting algorithms to demonstrate different trade-offs in sorting a list of `Route` objects.

Merge Sort (Stable Sort)

Merge Sort was chosen as the **stable sorting** algorithm. Its key advantages are:

- **Stability**: It preserves the relative order of elements with equal keys. This is important when sorting by multiple criteria (e.g., sorting by price, then duration), as the initial order is maintained for routes with the same price.
- **Guaranteed Performance**: It has a predictable time complexity of **O(n log n)** in all cases (worst, average, and best), making it reliable for any dataset.

Quick Sort (Unstable Sort)

Quick Sort was implemented as the **unstable sorting** algorithm. It is widely used in practice due to its speed and efficiency.

- **In-Place Sorting**: It typically sorts in-place, requiring only O(log n) space for the recursion stack, which is more memory-efficient than Merge Sort’ s O(n) space requirement.
- **Fast Average Performance**: Its average-case time complexity is **O(n log n)**, making it one of the fastest general-purpose sorting algorithms.

While its worst-case performance is $O(n^2)$, this is rare in practice with a good pivot selection strategy.

3. Time and Space Complexity Analysis

The following table summarizes the complexity analysis for the key algorithms used in the application.

Algorithm	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Justification
Dijkstra’s	O(E + V log V)	O(E + V log V)	O(V + E)	With a `PriorityQueue`, efficient for sparse graphs like flight networks.
Modified DFS	$O(b^d)$	$O(b^d)$	O(d)	`b` is branching factor, `d` is depth limit. Feasible due to the low depth limit.
Merge Sort	O(n log n)	O(n log n)	O(n)	Predictable and stable, but requires extra memory for temporary arrays.
Quick Sort	O(n log n)	$O(n^2)$	O(log n)	Fast on average and memory-efficient, but unstable and has a poor worst case.
Linear Search	O(n)	O(n)	O(1)	Simple and sufficient for the current scale of flight data.

4. Overview of Each Class and Its Role

The project is organized into several packages, each with a distinct responsibility, promoting modularity and maintainability.

Class	Package	Role
`Airport`, `Flight`, `Route`	`model`	Define the core data structures. `Airport` is a node, `Flight` is an edge, and `Route` is a collection of flights.
`FlightGraph`	`graph`	Implements the graph data structure using an adjacency list, representing the network of airports and flights.
`DataLoader`	`util`	Handles all data ingestion and persistence, reading from and writing to CSV files.
`RouteFinder`	`algorithm`	Contains the core route-finding logic, including Dijkstra’s algorithm and the modified DFS for slowest route.
`RouteComparators`	`algorithm`	Provides `Comparator` implementations for sorting routes by price, duration, stopovers, and a combined criteria.
`RouteSorter`	`algorithm`	Implements the Merge Sort (stable) and Quick Sort (unstable) algorithms for sorting lists of `Route` objects.
`FlightSearch`	`search`	Implements linear search functionality to find flights based on various criteria like origin, airline, or flight number.
`FlightPlannerApp`	`menu`	The main application class, responsible for the interactive console menu and orchestrating the overall user experience.

5. Data Structure Rationale

The primary data structure for representing the flight network is an **Adjacency List**, implemented as a `Map<Airport, List<Flight>>`. This choice was made for several reasons:

- **Efficiency for Sparse Graphs**: Flight networks are typically sparse (the number of flights from an airport is much smaller than the total number of airports).

Adjacency lists are more space-efficient than adjacency matrices for sparse graphs.

- **Fast Neighbor Traversal**: It allows for efficient retrieval of all outgoing flights from a given airport, which is the most common operation in the route-finding algorithms.

For quick lookups, several `Map` structures are used:

- `Map<String, Airport>`: Allows for O(1) average-time access to `Airport` objects by their IATA code.
- `Map<Integer, Flight>`: Provides O(1) average-time access to `Flight` objects by their unique ID.

6. Handling Stopovers, Input Errors, and Assumptions

Stopover Handling

A minimum stopover time of **20 minutes** is automatically added between connecting flights when calculating the total duration of a route. This is a realistic assumption that accounts for the time required to deplane, transfer, and board the next flight.

Input Error Handling

The application is designed to be robust against invalid user input. The console menu uses `try-catch` blocks to handle `NumberFormatException` and other potential input mismatches. It provides clear error messages and prompts the user to enter valid data without crashing.

Assumptions

- **Time Zones**: For simplicity, time zones are ignored in this implementation. All times are treated as local.
- **Data Validity**: The application assumes that the data in the CSV files is generally well-formed. However, it includes basic validation to skip corrupted or incomplete lines, logging warnings to the console.

- **Static Data**: The flight and airport data is loaded once at the start and is treated as static during the application’s runtime.

References

[1] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[2] S. Sahni and T. Gonzalez, “P-complete approximation problems,” *Journal of the ACM*, vol. 23, no. 3, pp. 555-565, 1976.