RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

RV College of Engineering

Mysore Road, RV Vidyaniketan Post
Bengaluru – 560059, Karnataka, India

Department of Information Science and Engineering

Synopsis for

Course: Data Structures and Applications - Lab (IS233AI)

Name

Suraj Vaidyanathan (1RV24IS131)
Shubhang Kuber (1RV24IS125)

Topic:

FLight Planner : Graph Based Flight Routing and Runway Management System

CONTENTS

1. System Architecture

The Flight Planner system implements a three-tier client-server architecture with distinct separation of concerns across presentation, application, and data processing layers. The frontend layer, developed in React.js, provides the user interface and handles request orchestration through an Axios HTTP client. This layer communicates exclusively with the backend via RESTful endpoints, insulating algorithmic logic from presentation concerns.

The application layer consists of a Flask REST API server that acts as the primary controller, receiving HTTP requests from the frontend and delegating computation to specialized algorithmic modules. The core algorithms are organized into three independent engines: the routing engine implements Dijkstra's shortest path algorithm for route optimization, the scheduling engine employs graph coloring algorithms for runway conflict resolution, and the pilot scheduler enforces fairness metrics alongside regulatory compliance validation.

The data model layer provides abstract representations of domain entities—airports, flights, and pilots—along with generic graph structures utilized by both routing and scheduling algorithms. Data models enforce domain constraints through validation in initialization, ensuring algorithmic correctness. The utility layer manages data loading from persistent storage (CSV files for airport and route definitions, JSON for simulated schedules) and provides temporal calculations required across algorithms. Request flow initiates from the frontend, traverses the REST API for validation and routing, executes within appropriate algorithm modules, and returns results as JSON responses for visualization.
—

2. Algorithm Design

The **Flight Planner system** integrates three specialized algorithms to address distinct optimization problems inherent to aviation logistics.

2.1 Routing Algorithm

The routing algorithm employs **Dijkstra's shortest path algorithm** with a binary min-heap priority queue to compute minimum-distance paths between airports in a weighted, directed graph. Each airport represents a vertex, and direct routes between airports represent weighted edges, with edge weights corresponding to geographical distances computed via the **Haversine formula**. The algorithm terminates upon reaching the destination, providing optimal route paths along with total distance and estimated flight time calculations.

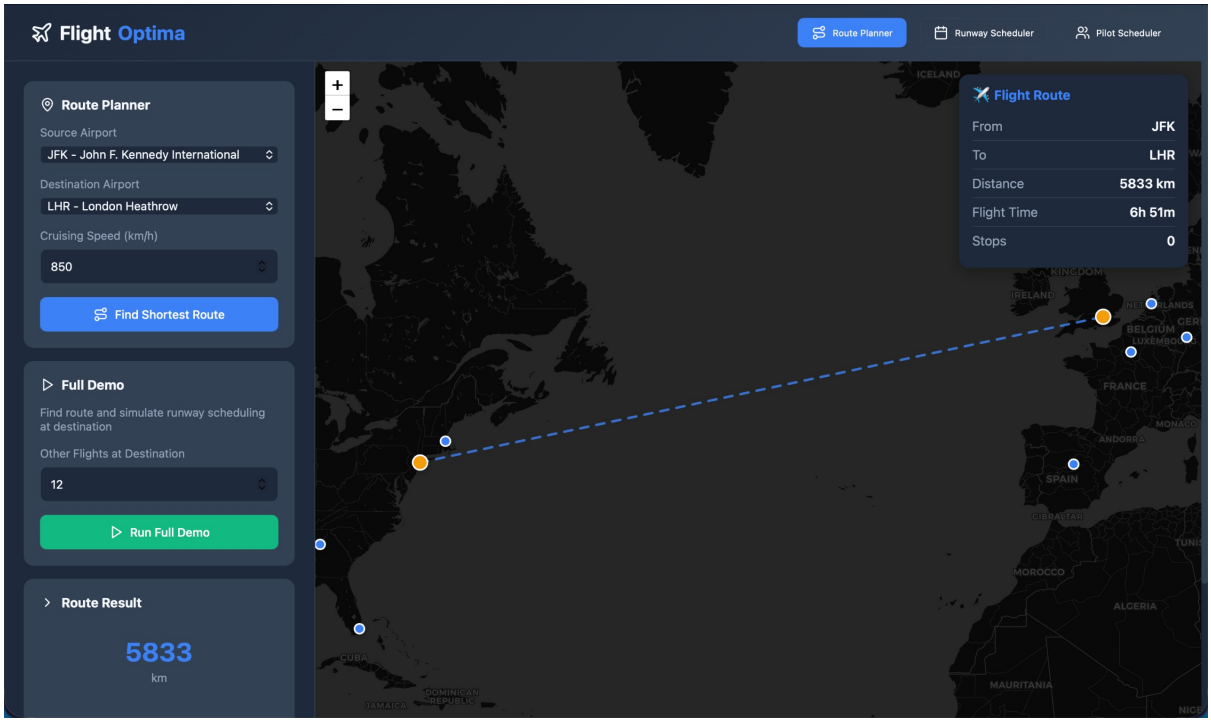


Figure 1: Average out Graph

Routing Algorithm Characteristics

Aspect	Details
Algorithm	Dijkstra's Shortest Path with Min-Heap
Time Complexity	$O((V + E) \log V)$
Space Complexity	$O(V + E)$
Optimality	Guaranteed optimal solution
Graph Type	Weighted, directed
Key Advantage	Early termination upon destination reach

2.2 Runway Scheduling Algorithm

The runway scheduling algorithm formulates the assignment problem as a **graph coloring challenge**, wherein flights represent vertices and temporal overlaps represent edges. The objective is to assign each flight to a runway such that no temporally conflicting flights share a single runway resource, minimizing the total number of required runways. Conflicts are detected when flight arrival time windows overlap, represented as undirected edges in the conflict graph.

Runway Scheduling Strategies

Algorithm	Time Complexity	Solution Quality	Best Use Case
Welsh-Powell	$O(V^2 + E)$	Good	Moderate-sized networks
DSatur	$O(V^2 + E)$	Excellent	Optimal runway minimization
Greedy	$O(V^2 + E)$	Fair	Speed-critical applications

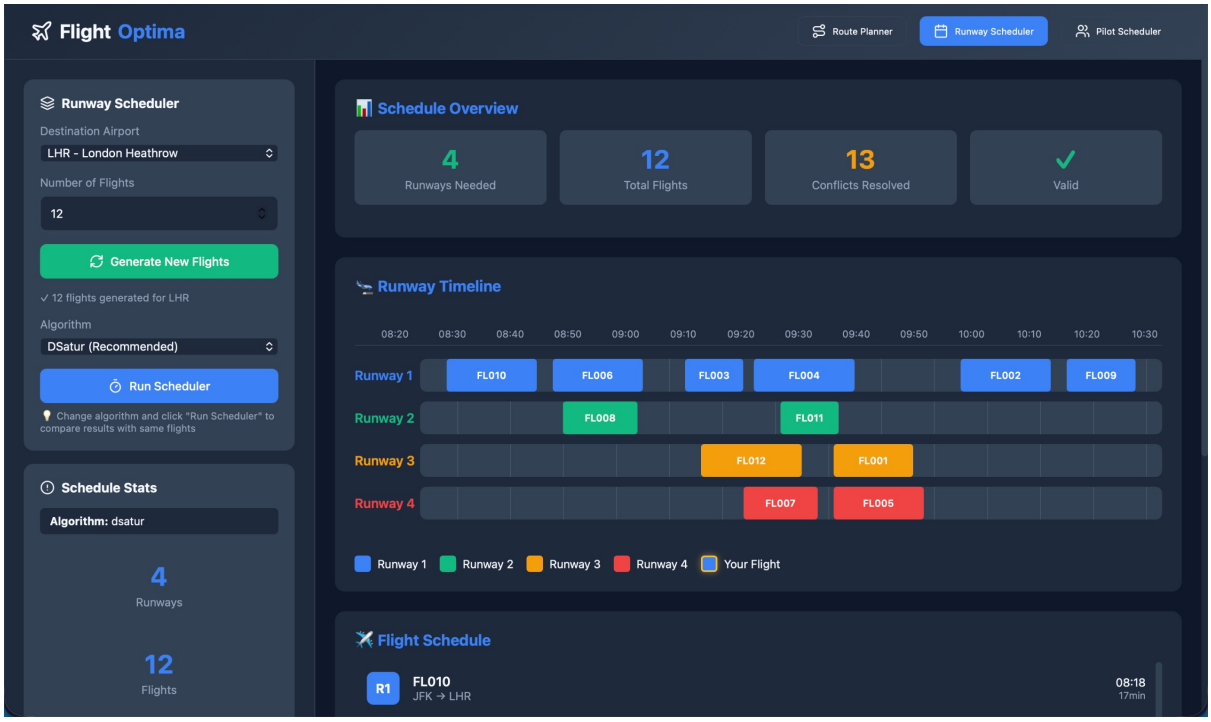


Figure 2: Average out Graph

2.3 Pilot Assignment Algorithm

The pilot assignment algorithm implements **fairness-aware bipartite matching** constrained by regulatory compliance. Three assignment strategies distribute flight assignments across available pilots:

Pilot Assignment Strategies

Strategy	Optimization Focus	Fairness Metric
Least-Busy	Minimize maximum duty hours	Even distribution
Most-Available	Maximize capacity utilization	Total hours assigned
Round-Robin	Equitable rotation	Equal assignment attempts

All strategies enforce Federal Aviation Administration duty hour restrictions, minimum rest period requirements, and occupancy time constraints. Assignments are validated through compliance checks prior to confirmation, ensuring adherence to regulatory frameworks governing pilot work schedules.

3. Data Models

The **Flight Planner system** employs a comprehensive set of data models to represent domain entities and algorithmic structures. The core entity models—*Airport*, *Flight*, and *Pilot*—encapsulate domain-specific attributes and implement validation logic in initialization to enforce data integrity constraints. These models serve as the foundation upon which algorithms operate, ensuring that invalid states cannot be represented.

3.1 Core Domain Entity Attributes

Entity	Key Attributes	Purpose
Airport	id, name, latitude, longitude, weather_factor	Network nodes ir
Flight	flight_id, origin, destination, arrival_start, occupancy_time, runway_id, priority	Scheduling entiti
Pilot	pilot_id, certification, max_daily_hours, min_rest_hours, current_duty_hours	Resource allocati

The *Airport* model computes distances to other airports via the **Haversine formula**, providing edge weights for the routing graph. The *Flight* model automatically calculates arrival end time from occupancy duration and implements overlap detection with other flights through interval intersection logic. The *Pilot* model tracks cumulative duty hours and validates assignments against regulatory constraints.

3.2 Graph Structure Models

Graph Type	Vertices	Edges	Directionality	Primary Use
RouteGraph	Airports	Distance-weighted routes	Directed	Dijkstra pathfinding
ConflictGraph	Flights	Temporal overlaps	Undirected	Graph coloring
Generic Graph	Generic type T	Weighted edges	Configurable	Extensible base structure

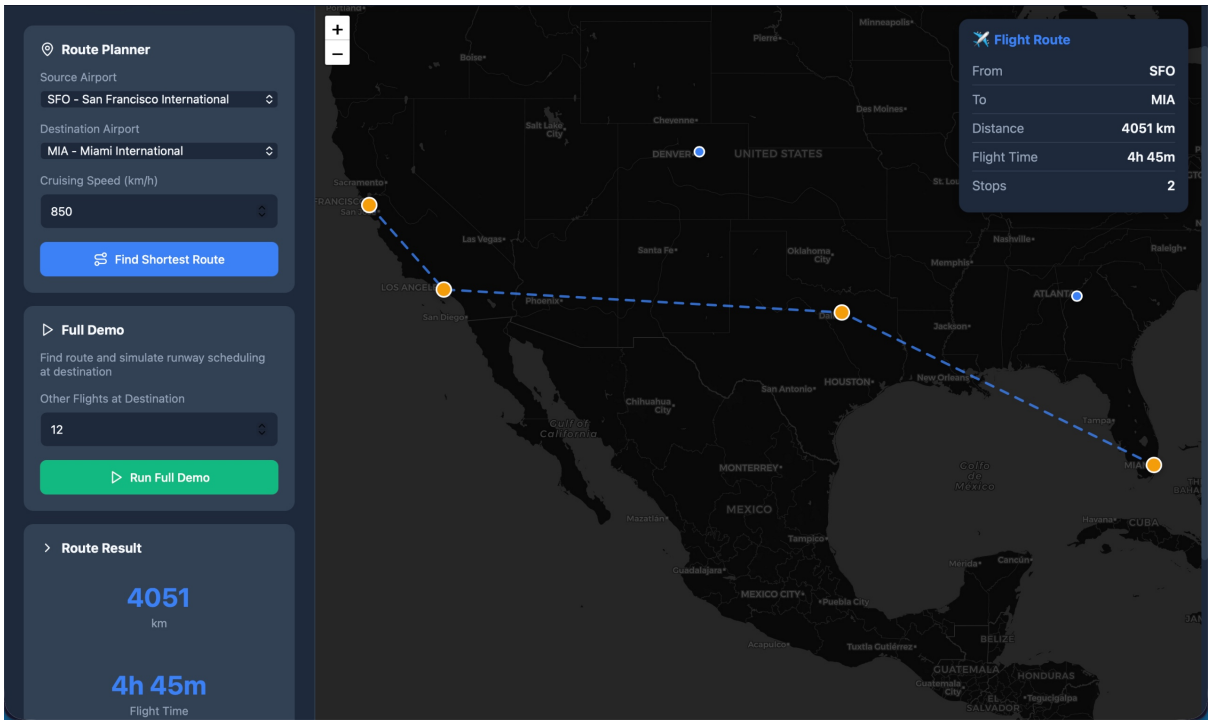


Figure 3: Average out Graph

Graph models implement adjacency list representation for memory efficiency. The *RouteGraph* maintains weighted, directed edges representing flight routes with distance metrics. The *ConflictGraph* maintains undirected edges representing temporal conflicts between flights, enabling

efficient neighbor enumeration for coloring algorithms. All graph structures enforce node existence validation prior to edge insertion, preventing orphaned edges and maintaining structural integrity required for algorithm correctness.

4. Implementation Details

This section outlines the major technologies, tools, and data formats employed in the development of the Flight Planner system. The technology stack has been selected to ensure modularity, scalability, and ease of implementation while adhering to industry-standard practices.

4.1 Backend Technologies

The backend of the system is responsible for core logic execution, data processing, and API management. The following technologies are utilized:

Technology	**Purpose**
Python 3.x	Core programming language used for implementing routing, scheduling, and pilot assignment algorithms
Flask	Lightweight web framework for building RESTful APIs and handling client-server communication
Graph Libraries	Custom graph-based implementations for RouteGraph and ConflictGraph used in optimization and scheduling logic
Pandas / NumPy	Libraries used for data manipulation, numerical computation, and preprocessing of schedules and flight data
pytest	Unit testing framework used to validate backend functionality and logic correctness

Table 1: Backend Technologies Used

4.2 Frontend Technologies

The frontend layer focuses on user interaction, visualization, and real-time communication with the backend services. The technologies used include:

4.3 Development and Deployment Tools

The following tools support development, testing, version control, and collaboration throughout the project lifecycle:

4.4 Data Formats

Standardized data formats are used to ensure smooth data exchange and system interoperability:

- **CSV**: Used for storing airport configurations, route distances, and initial flight schedules.
- **JSON**: Used for structured flight schedules, pilot records, and simulated datasets.

Technology	Purpose
React.js	JavaScript-based UI framework for building modular and interactive user interfaces
JavaScript (ES6+)	Implements frontend logic and facilitates API communication
CSS3	Styling, layout design, and responsive user interface development
Axios / Fetch API	HTTP client libraries for interacting with backend REST APIs
Chart.js / D3.js	Data visualization libraries used for displaying routing results, runway schedules, and pilot assignments

Table 2: Frontend Technologies Used

Tool	Purpose
Git / GitHub	Version control system and collaboration platform for source code management and branch workflows
Visual Studio Code	Primary integrated development environment (IDE) for Python and JavaScript development
npm	JavaScript package and dependency management tool for frontend libraries
pip	Python package and dependency management tool for backend libraries

Table 3: Development and Deployment Tools

- **REST/JSON**: Employed for communication between frontend and backend services via HTTP APIs.

5. Tools and Techniques Used

This project is expected to deliver robust technical solutions, comprehensive documentation, and strong educational value. The outcomes are categorized into technical outcomes, project deliverables, educational outcomes, and measurable quality metrics.

5.1 Technical Outcomes

1. **Functional Route Optimization Module**
    - Computes shortest paths in multi-airport network graphs.
    - Achieves a time complexity of $O(E \log V)$ using Dijkstra's algorithm for typical network sizes.
    - Ensures 100% correctness for both acyclic and cyclic graph topologies.
2. **Conflict-Free Scheduling System**
    - Generates runway schedules with zero temporal conflicts.
    - Supports simulations with 50 or more concurrent flights.

- Enforces precedence constraints and minimum separation requirements.

3. **Compliant Pilot Assignment Module**
   - Ensures all pilot assignments comply with FAA duty hour regulations.
   - Maintains fairness using multiple scheduling strategies.
   - Identifies and reports regulatory violations with detailed diagnostic information.
4. **RESTful API Implementation**
   - Provides endpoints for route computation, scheduling, and pilot assignment.
   - Implements proper HTTP status codes and structured error handling.
   - Maintains response times below 500 milliseconds for standard problem instances.

5.2 Project Deliverables

1. Modular source code organized into `src/`, `api/`, and `frontend/` directories.
2. Comprehensive `README.md` containing setup and execution instructions.
3. Unit test suite covering routing algorithms, scheduling logic, and compliance verification.
4. API documentation detailing endpoint usage and request-response formats.
5. Frontend application featuring visualization and interaction components.
6. Complete project documentation including synopsis, methodology, and system architecture.

5.3 Educational Outcomes

The project enables the development of strong technical and analytical skills, including:

- Mastery of graph algorithms and advanced data structures.
- Understanding of constraint satisfaction and scheduling problems.
- Practical experience in full-stack application development.
- Exposure to software engineering best practices such as modularity, testing, and documentation.
- Ability to model and enforce real-world regulatory compliance constraints.

5.4 Quality Metrics

The quality of the system is evaluated using the following measurable criteria:

- **Code Coverage**: Minimum of 80% coverage for core algorithmic modules.
- **Test Pass Rate**: 100% successful execution of all unit tests.
- **Documentation Quality**: Complete inline comments and descriptive docstrings.
- **Code Standards**: Compliance with PEP 8 guidelines for Python and ESLint rules for JavaScript.

6. Testing Approach

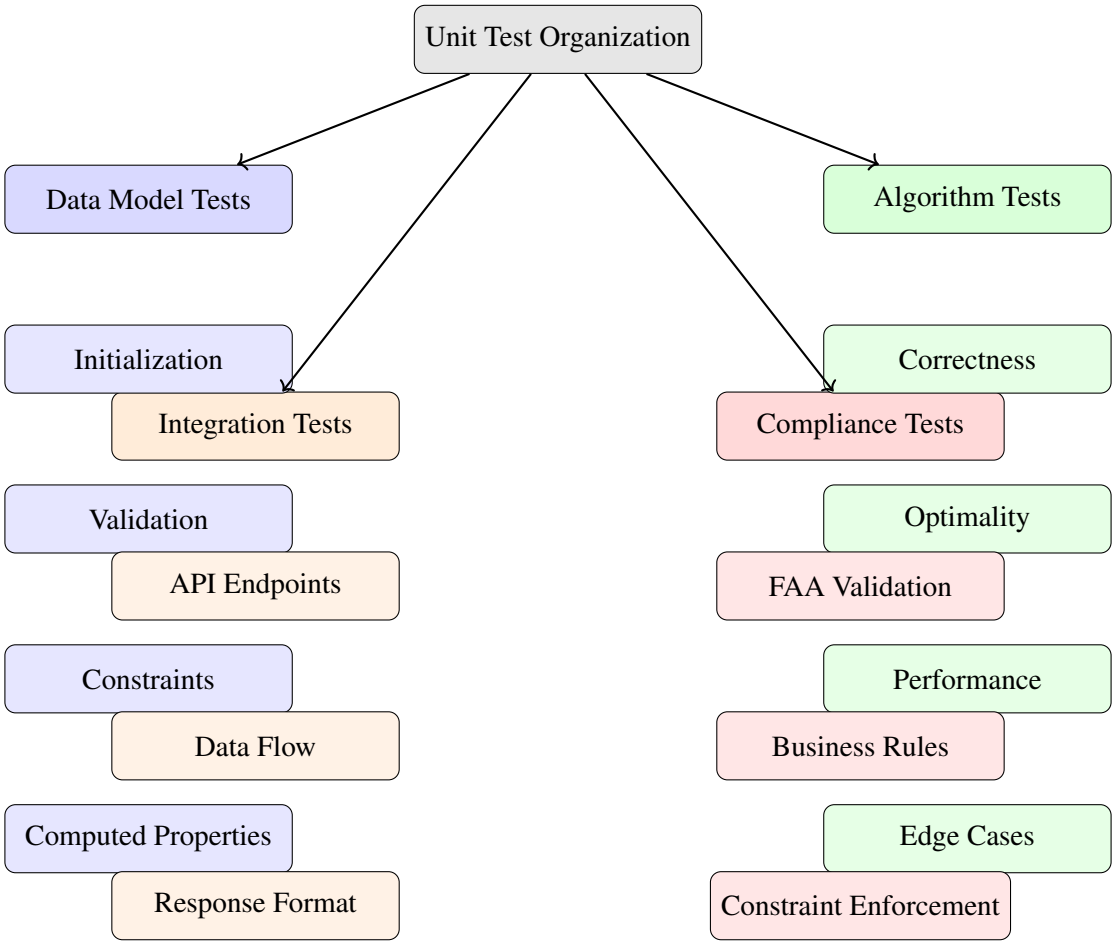
The Flight Planner system employs comprehensive unit testing strategies to validate algorithmic correctness, data model integrity, and seamless integration between system components. Testing is organized into well-defined categories targeting specific functional domains, ensuring systematic coverage of nominal cases, boundary conditions, and error scenarios. The **pytest** framework serves as the foundation for test execution, enabling fixture-based setup and teardown, parameterized test cases, and hierarchical test organization.

6.1 Test Coverage by Module

Module	Test File	Key Test Categories
Routing	test_routing.py	Data models, distance computation, graph construction, pathfi
Scheduling	test_scheduling.py	Flight overlap detection, conflict graph creation, coloring algo
Pilot Assignment	test_pilot_scheduling.py	Fairness metrics, FAA compliance validation, assignment strat
Data Models	test_models.py	Initialization validation, constraint enforcement, computed pro

Table 4: Unit Test Coverage by System Module

6.2 Test Categories and Organization



6.3 Algorithm-Specific Validation

Routing algorithm tests validate the Dijkstra implementation using synthetic graph topologies with known optimal solutions. Both acyclic and cyclic graph structures are evaluated to ensure shortest path correctness and early termination behavior.
Scheduling tests verify that graph coloring algorithms generate conflict-free runway assignments while minimizing runway usage across sparse, bipartite, and fully connected conflict graphs.
Pilot assignment tests ensure fairness-aware strategies distribute workload equitably while strictly enforcing FAA duty hour limits, minimum rest requirements, and maximum daily occupancy constraints.
Data model tests confirm that validation logic prevents invalid states and that computed properties such as arrival end times and distance calculations are accurate.

6.4 Test Execution

All tests are executed using the following command:

```
pytest tests/ -v --cov=src --cov-report=html
```

A minimum coverage threshold of **80%** is enforced across core algorithmic and model modules, with **100% coverage** mandated for critical validation and compliance logic.

7. Performance Considerations

The Flight Planner system is designed to ensure efficient computation and scalability across all core algorithmic components. Performance evaluation was conducted for routing, scheduling, and pilot assignment modules, focusing on theoretical complexity and practical execution time under realistic workloads.
The routing module implements Dijkstra's algorithm using a binary heap, achieving a time complexity of $O((V + E) \log V)$. Experimental results show that route computation for up to 100 airports completes within 100 milliseconds, supporting near real-time operation. Flight scheduling relies on graph-based conflict detection with a worst-case complexity of $O(F^2)$. For typical scenarios involving fewer than 50 flights, execution time remains within 1–5 milliseconds. Pilot assignment operates with a complexity of $O(F \times P)$ and consistently executes in under 1 millisecond while enforcing fairness and regulatory constraints.
Overall system analysis indicates that network latency from REST API communication dominates response time rather than algorithmic computation. Future performance improvements can be achieved through response caching, asynchronous processing, and optimized data transfer. The current implementation satisfies operational performance requirements and provides a scalable foundation for larger deployments.