AIRPORT ROUTE FINDER USING GRAPH ALGORITHMS

CSL2020 PROJECT DEMONSTRATION

Leveraging graph data structures and algorithms to enhance flight schedule management, visualization, and decision-making for travelers and planners

PROBLEM STATEMENT

- In large airport networks, figuring out the most efficient route between two locations is difficult.
- With many airports and routes, manual decision-making isn’t practical or reliable.
- We need a method that can quickly determine the best possible route based on distance or most cost efficient route based on pricing.

OUR SOLUTION

- We represent all airports and routes as a graph, where each connection (edge) has distance and cost weights.
- To determine the best travel route, we apply two powerful algorithms — Dijkstra and Bellman-Ford — both designed to find the most efficient path through the network.
- Dijkstra’s algorithm explores all possible routes, guaranteeing the most accurate shortest path.
- Bellman-Ford is another standard shortest path algorithm used in various applications.
- The project provides an easy-to-use React-based interface where users can add airports, draw routes, and interact with the airport network(graph).

KEY DATA STRUCTURES USED

Data structure	Usage
HashMaps	Adjacency list, dashboard
Queues	BFS, connected componenets
Graphs	Representation of airport map
Classes / Objects	Graphs
Heaps / Priority queues	Dijkstra’s algorithm
Set	Unique components, centrality
Array	Graphs nodes, data storage

ALGORITHMS USED

Algorithm	Data Structure used	Reasoning behind choice
Dijkstra’s	Adjacency list (Map), Array, Min-heap/priority queue	Fast and efficient for finding shortest or cheapest paths with non-negative edge weights graphs
Bellman-Ford	Adjacency list (Map), Array	simple and general single-source shortest path algorithm
BFS	Adjacency list (Map), Array, Queue, Set	Guarantees finding the path with the fewest number of edges (“stops”)
Centrality measures	Adjacency list (Map), Array, Set	Identifies important “hub” nodes and subnetworks; helps in network analysis and visualization

DIJKSTRA

- Uses a priority queue to always expand the closest unvisited node.
- Avoids unnecessary work by exploring only the most promising direction first.
- Does not relax edges blindly; it updates only when a shorter path is discovered.
- Much faster in practice: grows outward like ripples instead of scanning all edges.
- Requires non-negative weights, but gives optimal shortest paths efficiently.

BELLMAN FORD

- Relaxes every edge, V−1 times to gradually improve distance estimates.
- Checks all possible paths, even ones that are not useful.
- Works even when edges have negative weights.
- Can detect negative cycles by checking if any distance still improves after V−1 rounds.
- Slower because it repeatedly scans the entire edge list.
- Useful when graphs may contain negative edges or require cycle detection.

GRAPH IMPLEMENTATION

1. The class represents a full Graph structure
The code defines a Graph class that stores airports as nodes and routes as edges, forming the backbone of the project's path-finding system.
2. It supports adding and removing nodes (airports)
Functions like addAirport(code) and removeAirport(code) let you dynamically modify the graph's nodes.
3. It supports adding and removing edges (routes)
addRoute(src, dest, distance, cost) and removeRoute(src, dest) manage connections between airports.
4. It exposes utility functions to access graph data
Methods like getAirports() and getNeighbors(node) allow algorithms (like Dijkstra or A*) to read the graph efficiently.
5. It includes JSON import/export functions
toJSON(), fromJSON(data), and fromData(airports, routes) allow the graph to be saved, loaded, and reconstructed—important for file handling and persistence.

```
1   export class Graph {
2   >   constructor() {…
4       }
5   >   addAirport(code) {…
9       }
10  >   addRoute(src, dest, distance, cost) {…
25      }
26  >   removeAirport(code) {…
37      }
38  >   removeRoute(src, dest) {…
49      }
50  >   getAirports() {…
52      }
53  >   getNeighbors(node) {…
55      }
56  >   printGraph() {…
69      }
70  >   static fromData(airports, routes) {…
75      }
76  >   toJSON() {…
88      }
89  >   static fromJSON(data) {…
94      }
95  }
```

TEAM MEMBERS AND CONTRIBUTIONS

01

NIHAR KUCHANKAR

Frontend / Backend
Algorithm implementation
Graph functions

02

JAENIL PAREKH

Frontend / backend
website UI
Algorithm implementation

03

KRISH PATEL

PPT
Code review
Edge case testing

04

YASHVABHAV BHARDWAJ

PPT
UI functionality
Data collection

JAENIL
PAREKH
B24CS1117

KRISH PATEL
B24CS1054

NIHAR
KUCHANKAR
B24CS1102

YASHVABHAV
BHARDWAJ
B24EE1100

THANK YOU