Airport Route Optimization with Graph Algorithms: Implementation and Performance Evaluation

Devon Sun
Emily Wang

December 14, 2025

This report presents the design, implementation, and performance analysis of an airport route optimization system. We implement and compare two graph representation strategies (adjacency matrix and adjacency list) with four pathfinding algorithms (BFS, DFS, Dijkstra, and A*). Our empirical analysis demonstrates significant performance improvements when using adjacency lists for sparse graphs, achieving speedups of up to 16× on large datasets while maintaining algorithmic correctness.

1 Introduction

The efficient routing of flights between airports is a fundamental problem in transportation networks. This project implements a route optimization system that progressively optimizes both data structures and algorithms to handle increasingly large airport networks. We analyze three datasets of varying sizes (24, 200, and 500 airports) with strict time constraints, demonstrating the importance of choosing appropriate data structures for sparse graphs.

2 Complexity Analysis

2.1 Phase 1: Adjacency Matrix Implementation

In Phase 1, the airport network is represented using an adjacency matrix, where edge weights are stored in a $V \times V$ matrix and V denotes the number of vertices (airports). This representation requires $O(V^2)$ space, as a weight value is maintained for every possible vertex pair regardless of whether a direct route exists. While edge existence and weight queries can be performed in constant time via direct array access, iterating over the neighbors of a vertex requires scanning the entire corresponding row of the matrix, resulting in $O(V)$ time per vertex expansion.

2.1.1 Breadth-First Search (BFS) and Depth-First Search (DFS)

Using an adjacency matrix, both BFS and DFS visit each vertex at most once. However, for each visited vertex, the algorithm must scan the entire row of the matrix to identify neighboring vertices, which requires $O(V)$ time. Consequently, the overall time complexity of both BFS and DFS in Phase 1 is $O(V^2)$.

2.1.2 Dijkstra's Algorithm

Dijkstra's algorithm computes shortest weighted paths by repeatedly selecting the vertex with the smallest tentative distance using a priority queue. In the adjacency matrix representation, each vertex is finalized at most once. After a vertex is extracted from the priority queue, the algorithm scans the entire corresponding row of the matrix to relax outgoing edges, which costs $O(V)$ time, in addition to $O(\log V)$ heap operations. As a result, the overall time complexity of Dijkstra's algorithm in Phase 1 is $O(V^2 \log V)$.

2.1.3 Empirical Results

Table 1 and Fig. 1 report the empirical runtime performance of BFS, DFS, and Dijkstra's algorithm under the adjacency matrix representation. As the number of airports increases, the runtime of all three algorithms grows rapidly, particularly for the large dataset.

Table 1: Empirical runtime results for Phase 1 (Adjacency Matrix), measured in milliseconds.

Dataset	N	BFS	DFS	Dijkstra
Small	24	0.861	0.744	1.164
Medium	200	310.452	295.509	478.394
Large	500	4770.935	4697.438	7393.170

Table 2: Empirical runtime results for Phase 2 (Adjacency List + A*), measured in milliseconds.

Dataset	N	M	BFS	DFS	Dijkstra	A*
Small	24	216	0.012	0.202	0.531	0.488
Medium	200	2966	0.280	19.965	56.721	3.965
Large	500	14987	1.436	215.130	611.833	12.593

This behavior is consistent with the $O(V^2)$ cost of neighbor iteration inherent in adjacency matrices, which dominates performance even when the underlying graph is sparse. While this representation is simple and effective for small graphs, it does not scale well to larger datasets, motivating the use of a more efficient graph representation in Phase 2.

2.2 Phase 2: Adjacency List Implementation

In Phase 2, we adopt an adjacency list representation for the airport network. Instead of storing all possible vertex pairs, the adjacency list records only existing edges in the form `adj[u] = [(v, w), ...]`, where each entry represents a directed edge from vertex u to vertex v with weight w. This representation requires $O(V + E)$ space, where E denotes the number of edges, and allows neighbor traversal in time proportional to the vertex degree. As a result, adjacency lists are significantly more efficient than adjacency matrices for sparse graphs.

2.2.1 Breadth-First Search (BFS) and Depth-First Search (DFS)

With an adjacency list representation, both BFS and DFS visit each vertex and examine each edge at most once during traversal. Therefore, the overall time complexity of both algorithms in Phase 2 is $O(V + E)$.

2.2.2 Dijkstra's Algorithm

Using adjacency lists together with a priority queue, Dijkstra's algorithm relaxes each edge at most once and performs heap operations for distance updates. The resulting time complexity is $O((V + E) \log V)$.

2.2.3 A* Search

A* search extends Dijkstra's algorithm by incorporating a heuristic function to guide the search toward the destination more efficiently. In the context of an airport routing system, we use the Euclidean distance between two airports as the heuristic, motivated by the geographic nature of real-world flight routes.

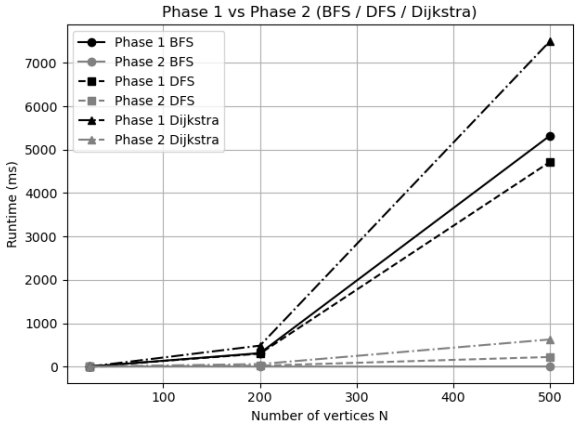

Figure 1: Runtime comparison of BFS, DFS, and Dijkstra's algorithm using adjacency matrix (Phase 1) and adjacency list (Phase 2) implementations across small, medium, and large datasets.

The heuristic function is defined as

$$h(u, \text{target}) = \sqrt{(x_u - x_{\text{target}})^2 + (y_u - y_{\text{target}})^2},$$

where (x_u, y_u) and $(x_{\text{target}}, y_{\text{target}})$ denote the geographic coordinates of airport u and the destination airport, respectively.

This heuristic is admissible because the length of any feasible flight path cannot be shorter than the straight-line distance between two airports, and it is consistent due to the triangle inequality. As a result, A* is guaranteed to return an optimal shortest path.

In the worst case, the time complexity of A* remains $O((V + E) \log V)$. However, in practice, heuristic guidance significantly reduces the number of explored vertices compared to Dijkstra's algorithm, leading to improved empirical performance.

2.2.4 Empirical Results

As shown in Fig. 1 and Tables 1–2, the adjacency list implementation achieves substantial performance improvements over the adjacency matrix approach on large graphs. For the large dataset ($N = 500$), the runtime of BFS, DFS, and Dijkstra decreases by approximately 99.97%, 95.42%, and 91.72%, respectively.

In addition, A* significantly outperforms Dijkstra's algorithm on point-to-point queries. On the large dataset, A* reduces the runtime by over 97% compared to Dijkstra, demonstrating the practical benefit of heuristic-guided search.

3 Extensions and Discussion

Beyond classical shortest-path queries, real-world airport routing systems often require more flexible and informative routing capabilities. In this section, we present two advanced extensions implemented in our system—*k-shortest paths* and *constrained routing with minimum stops*—and discuss their design motivations, algorithmic approaches, and computational properties. We also summarize scenarios in which the adjacency matrix representation remains a reasonable choice despite the advantages of adjacency lists.

3.1 K-Shortest Paths

In practical flight planning, users may require not only a single shortest route, but multiple alternative routes with increasing costs. To support this functionality, we implement the *k-shortest paths* problem, which computes the top k distinct shortest loopless paths between a given source and destination.

3.1.1 Algorithm Design

Our implementation follows the classical Yen's algorithm, which enumerates the k shortest loopless paths in strictly increasing order of total cost. The algorithm builds upon repeated applications of Dijkstra's shortest path algorithm.

First, the shortest path from the source to the destination is computed using Dijkstra's algorithm and stored as the initial result. For each subsequent path, candidate paths are generated by selecting a *spur node* along the previously discovered shortest path. The prefix from the source to the spur node is fixed as the *root path*, while edges that would reproduce an already discovered path are temporarily removed from the graph.

A shortest path from the spur node to the destination is then recomputed using Dijkstra's algorithm on the modified graph. If such a path exists, it is concatenated with the root path to form a candidate. All candidates are maintained in a priority queue ordered by total cost, and the minimum-cost candidate is selected as the next shortest path. This process repeats until k paths are found or no further candidates exist.

3.1.2 Time Complexity

Each newly discovered shortest path may generate up to $O(V)$ spur paths, and each spur path computation invokes Dijkstra's algorithm once. With an adjacency list representation and a binary heap, each Dijkstra run has time complexity $O((V + E) \log V)$. Therefore, the worst-case time complexity of computing the k shortest paths is

$$O\big(k \cdot V \cdot (V + E) \log V\big).$$

In practice, the number of valid spur paths is typically much smaller than V, especially in sparse graphs such as airport networks, making the algorithm efficient for moderate values of k.

3.2 Constrained Routing: Minimum Stops with Cost Optimization

In many airline routing scenarios, passengers prioritize routes with fewer stops, even if such routes are slightly more expensive. To capture this preference, we implement a constrained routing algorithm that optimizes routes in a lexicographic manner: the primary objective is to minimize the number of stops, and among all routes with the same number of stops, the secondary objective is to minimize the total travel cost.

3.2.1 Algorithm Design

The problem is formulated as a shortest-path search with an explicit constraint on the maximum number of stops. The state is represented as a pair (v, s), where v is the current vertex and s is the number of edges used so far. A two-dimensional distance table is maintained, where $\text{dist}[s][v]$ stores the minimum cost to reach vertex v using exactly s edges. A corresponding parent table is used for path reconstruction.

A priority queue is employed with lexicographic ordering on (s, cost), ensuring that states with fewer edges are always expanded first, and among those, states with smaller total cost are prioritized. The algorithm terminates when the destination is reached or when the maximum allowable number of edges is exceeded.

3.2.2 Time Complexity

The expanded state space contains at most $O(S \cdot V)$ states, where S is the maximum number of allowed stops. Each state relaxation examines outgoing edges, resulting in $O(S \cdot E)$ relaxations. Using a binary heap priority queue, the overall time complexity is

$$O\big(S \cdot E \log(SV)\big).$$

Since S is typically small in realistic airline routing scenarios, this approach is efficient in practice for constrained queries.

3.3 Discussion: When to Use an Adjacency Matrix

Although the adjacency list representation significantly improves performance for sparse graphs, the adjacency matrix representation may still be preferable in certain scenarios.

For dense graphs where $E \approx V^2$, the space complexity of adjacency lists approaches that of adjacency matrices, and the constant-time edge lookup of matrices can be advantageous. In applications that require frequent edge existence queries, such as repeated checks of whether a direct connection between two airports exists, adjacency matrices provide $O(1)$ lookup time, whereas adjacency lists require scanning neighbor lists.

Additionally, for very small graphs, the simplicity and cache-friendly memory layout of adjacency matrices may outweigh their theoretical inefficiencies. Certain algorithms, such as Floyd–Warshall for all-pairs shortest paths, also naturally operate on matrix representations, making adjacency matrices a more suitable choice in those contexts.

Overall, while adjacency lists are the preferred representation for large, sparse airport networks, adjacency matrices remain a valid and sometimes advantageous option depending on graph density, query patterns, and algorithmic requirements.

4 Conclusion

In this project, we designed and evaluated an airport route optimization system using multiple graph representations and routing algorithms. By comparing adjacency matrix and adjacency list implementations, we demonstrated that data structure selection plays a critical role in algorithmic scalability for sparse graphs. Empirical results show that adjacency lists dramatically reduce runtime for BFS, DFS, and Dijkstra's algorithm on large datasets.

Furthermore, the incorporation of heuristic-guided search through A* yields substantial performance gains over classical shortest-path algorithms for point-to-point queries. The implemented extensions, including k-shortest paths and constrained routing with minimum stops, further enhance the practical applicability of the system by supporting realistic airline routing requirements.

Overall, the proposed system achieves significant performance improvements while maintaining correctness and flexibility, making it well-suited for large-scale airport networks. Future work may explore additional optimizations such as bidirectional search, preprocessing techniques, or parallel implementations to further improve efficiency.