Implementation and In-Depth Analysis of Flight Optimization Systems for Travel Agencies: A Multi-Criteria Approach

Yasin Yeşilyurt
TOBB ETÜ Artificial Intelligence Engineering
Söğütözü cad. TOBB ETÜ Konukevi
Yenimahalle/Ankara
yasinyesilyurt@hotmail.com

***Abstract*—This paper presents a multi-criteria flight optimization system for travel agencies, designed to recommend optimal flight routes by balancing cost, duration, and customer satisfaction. The proposed framework employs A*, Dijkstra's, and Genetic Algorithms implemented in Python to evaluate flight data spanning 1993-2023 from U.S. domestic routes, sourced from a Kaggle dataset containing 245,955 entries with parameters such as fare, distance, carrier dominance, and airport coordinates. Key innovations include a hybrid approach combining heuristic pathfinding (using Haversine distance for A*) with multi-objective A* to address multi-objective optimization. Results demonstrate the effectiveness of A* and Dijkstra's algorithms in minimizing travel costs and time, while genetic algorithm gives sufficient results, it is much slower than mentioned algorithms without proper optimization techniques. The system currently provides personalized flight recommendations based on user preferences, with visualized outputs generated via the NetworkX library. This work contributes a scalable framework for enhancing decision-making in travel planning systems.**

I. INTRODUCTION

The growing complexity of air travel planning necessitates advanced systems to optimize flight routes while balancing competing factors such as cost, duration, and customer satisfaction. Traditional travel recommendation tools often prioritize single objectives, such as minimizing price or travel time, but fail to account for multi-dimensional preferences or dynamic market conditions. This limitation underscores the need for intelligent systems capable of synthesizing diverse parameters to deliver personalized and context-aware solutions.

Existing approaches to route optimization predominantly rely on classical graph algorithms like Dijkstra's or heuristic methods such as A*. However, these methods struggle to handle multi-criteria optimization, where trade-offs between conflicting objectives (e.g., cost vs. comfort) must be quantified. While recent studies have explored hybrid frameworks combining machine learning and optimization techniques, their reliance on simplified datasets or synthetic benchmarks limits real-world applicability.

This paper addresses these gaps by proposing a hybrid algorithmic framework that integrates A* (heuristic Search), Dijkstra's algorithm (shortest-path), and Genetic Algorithms (evolutionary optimization) to optimize flight routes across three criteria: fare, travel time, and carrier-specific customer satisfaction metrics. The system leverages a comprehensive dataset of 245,955 U.S. domestic flights (1993-2023) Fig. 1, which includes real-world attributes such as historical fares, carrier dominance, and geographic coordinates. Key innovations include:

- A Haversine-based heuristic for A* to compute geodesic distances between airports, improving route accuracy.
- A genetic crossover mechanism to evolve flight paths while preserving high-quality route segments.
- A dynamic weighting system to prioritize user-defined preferences (e.g., cost-sensitive vs. time-sensitive travelers).
- A multi-objective A* algorithm that evaluates trade-offs between fare, travel time, and customer satisfaction, enabling personalized recommendations.
- Visualization of flight networks and optimized routes using the NetworkX library, enhancing decision-making.

Preliminary results demonstrate the viability of A* and Dijkstra's algorithms for single-objective optimization but the genetic algorithm exhibits slower performance. Among the mentioned algorithms A* performs best in terms of cost and time minimization thus it has been selected as multi-criteria optimizer. The remainder of this paper is organized as follows: Section II details the methodology and dataset, Section III discusses implementation challenges and results, and Section IV outlines future work to refine hybrid algorithm performance and integrate real-time customer feedback.

II. METHODOLOGY

The proposed flight optimization system follows a structured pipeline (Fig. 1) comprising data preprocessing, algorithmic implementation, and multi-criteria evaluation.

A. Data Collection and Preprocessing

A dataset of 245,955 U.S. domestic flights (1993–2023) [1] was sourced from Kaggle, containing attributes such as fares,

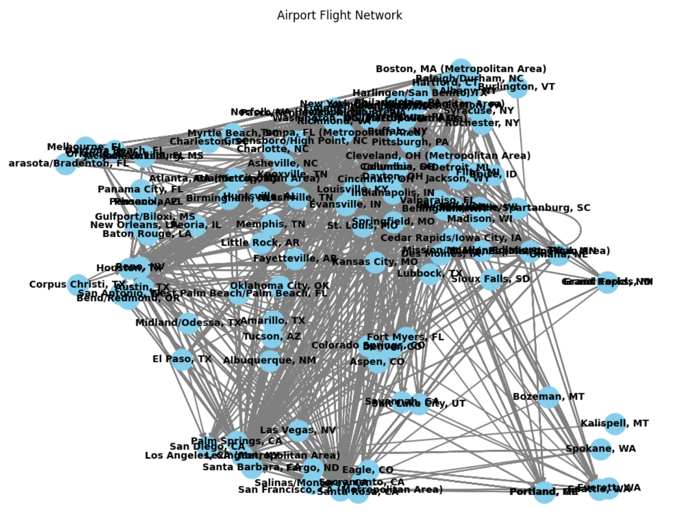


Fig. 1. Graph representation of flight routes, with airports as nodes and flights as edges.

carriers, passenger counts, airport coordinates, and quarterly trends. Key preprocessing steps include:

- 1. Data Cleaning: Removal of 45,955 incomplete entries, retaining 200,000 flights with non-null values.
- 2. Subset Selection: Partitioning by year and quarter to reduce computational load.
- 3. Graph Construction: Representing airports as nodes and flights as edges in a weighted graph using Python's 'NetworkX' library. Edge weights combine:

Cost	Normalized fare ($) scaled by carrier dominance (market share).
Time	Flight duration derived from Haversine distance (km) between coordinates.

Workflow diagram of the flight path finder pipeline is shown in Fig. 2.

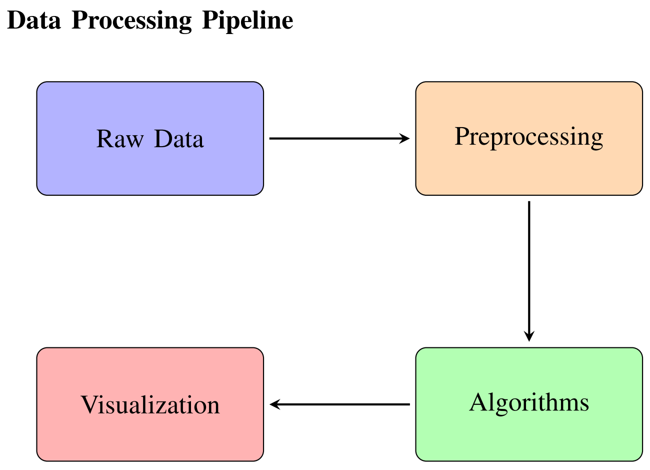


Fig. 2. Flight path finder pipeline workflow from data to visualization

B. Algorithm Design

Three core algorithms with a multi-objective variant of these algorithms were implemented and compared:

1) A* Algorithm:
   - Heuristic Function: Geodesic distance between airports computed via the Haversine formula:

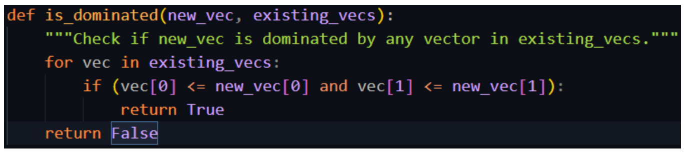
$$d = 2R \arcsin\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1 \cos\phi_2 \sin^2\left(\frac{\Delta\lambda}{2}\right)}$$

     where R is Earth's radius, and ϕ, λ denote latitude/longitude.
   - Cost Function: $C = \alpha \cdot \text{fare} + \beta \cdot \text{distance}$.

2) Dijkstra's Algorithm:
   - Computes globally optimal paths for a single criterion (e.g., minimal distance).
   - Extended to support dynamic reweighting of edges based on user preferences.

3) Genetic Algorithm (GA):
   - Population Initialization: 100 chromosomes generated via greedy random walks.
   - Crossover: Segments of parent paths sharing common nodes (e.g., P1: A→B→C→D and P2: A→X→C→Y yield children A→B→C→Y and A→X→C→D).
   - Mutation: Random node insertion/deletion (currently disabled due to instability in variable-length paths).
   - Fitness Function: Minimizes *cost* while penalizing excessive layovers.

4) Multi-Criteria A* Algorithm:
   - Computes Pareto optimal paths across multiple criteria (cost, time). With the heuristic approach of A*.
   - Heuristic Function: Haversine distance as in A*.
   - Pareto Optimality: stores the paths that fits the specified criterias. Critearia function follows in Fig 3.



```python
def is_dominated(new_vec, existing_vecs):
    """Check if new_vec is dominated by any vector in existing_vecs."""
    for vec in existing_vecs:
        if (vec[0] <= new_vec[0] and vec[1] <= new_vec[1]):
            return True
    return False
```

Fig. 3. Pareto Optimal checker for Multi-Criteria A* Algorithm

III. Implementation Details

- Environment: Python 3.9, Jupyter Notebook for interactive development.
- Libraries: 'NetworkX' for graph representation, 'pandas' for data manipulation, 'numpy' for numerical operations.
- Visualization: Matplotlib and NetworkX for geographic rendering of flight paths.

A. *Challenges and Mitigations*

1) *Data Completeness and Geographic Accuracy:*

- Challenge: Initial datasets lacked geographic coordinates for airport entries, compromising distance calculations.
- Mitigation: Found a different dataset that included most of its airport coordinates.

2) *Computational Bottlenecks:*

- Challenge: Rendering 200,000+ flights in `NetworkX` caused latency.
- Mitigation: Visualization was limited to 50–100 nodes when testing, complete graphs are rendered afterwards.

3) *Multi-Parameter Optimization:*

- Challenge: No standard function to unify cost, time, and comfort metrics.
- Mitigation: A user-configurable weighted sum (e.g., 20% cost, 80% distance) and multi-parameter A* was implemented.

4) *Genetic Algorithm Implementation:*

- Challenge 1: Python's list/tuple structures caused inefficient chromosome representation. That caused shape mismatch errors during crossover and mutation operations.
- Mitigation: Mutation instability was resolved by enforcing path continuity checks during chromosome editing. But a more planned chromosome representation is recommended for future work.
- Challenge 2: Random Heuristic Walk algorithm caused mutations to be mostly longer paths. This caused mutated chromosomes to be longer than the original ones.
- Mitigation: Implemented a natural selection (sorting the chromosomes by their length) is implemented.

IV. Detailed Analysis of Algorithms

This section evaluates the performance, strengths, and limitations of the implemented algorithms: A*, Dijkstra's, and Genetic Algorithms (GA). Quantitative results are derived from testing on a 200,000-flight set.

A. *A* Algorithm*

Fig. 4

1) *Theoretical Basis:*
A* combines uniform-cost search with a Haversine-based heuristic to compute geodesic distances between airports.:

$$h(n) = \text{Haversine}(n_{\text{current}}, n_{\text{destination}})$$

2) *Implementation:*

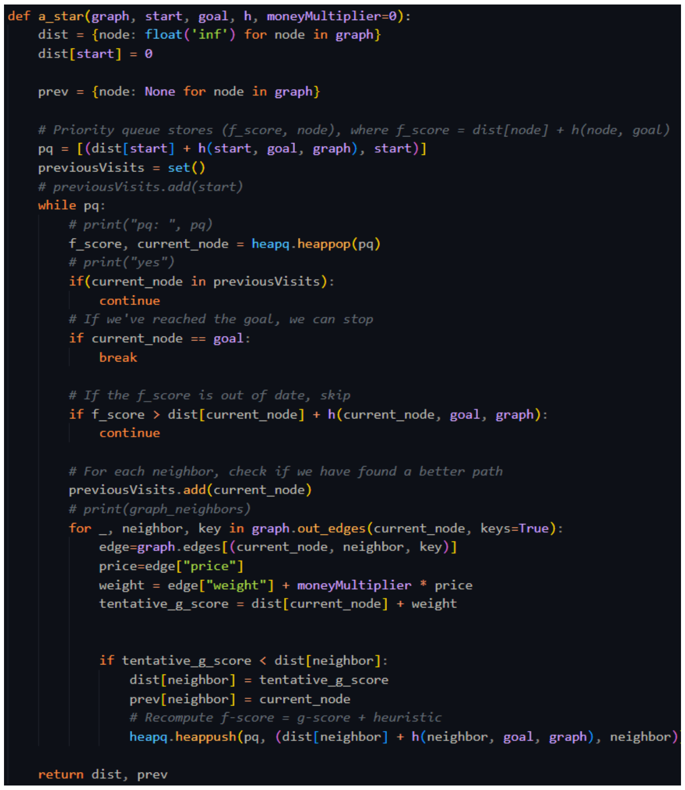
- **Cost Function:**

$$C = \alpha \cdot \text{fare} + \beta \cdot \text{distance}$$

- **Optimality:** Not guaranteed for all heuristics; requires admissibility and consistency.
- **Complexity:** $O(b^d)$ where $b = avg.branchingfactor$

3) *Performance:*

- Average runtime: 0.009s per path
- Scales linearly with path depth d



```python
def a_star(graph, start, goal, h, moneyMultiplier=0):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0

    prev = {node: None for node in graph}

    # Priority queue stores (f_score, node), where f_score = dist[node] + h(node, goal)
    pq = [(dist[start] + h(start, goal, graph), start)]
    previousVisits = set()
    # previousVisits.add(start)
    while pq:
        # print("pq: ", pq)
        f_score, current_node = heapq.heappop(pq)
        # print("yes")
        if(current_node in previousVisits):
            continue
        # If we've reached the goal, we can stop
        if current_node == goal:
            break

        # If the f_score is out of date, skip
        if f_score > dist[current_node] + h(current_node, goal, graph):
            continue

        # For each neighbor, check if we have found a better path
        previousVisits.add(current_node)
        # print(graph_neighbors)
        for _, neighbor, key in graph.out_edges(current_node, keys=True):
            edge=graph.edges[(current_node, neighbor, key)]
            price=edge["price"]
            weight = edge["weight"] + moneyMultiplier * price
            tentative_g_score = dist[current_node] + weight


            if tentative_g_score < dist[neighbor]:
                dist[neighbor] = tentative_g_score
                prev[neighbor] = current_node
                # Recompute f-score = g-score + heuristic
                heapq.heappush(pq, (dist[neighbor] + h(neighbor, goal, graph), neighbor)

    return dist, prev
```

Fig. 4. Python code of A* Algorithm

B. *Dijkstra's Algorithm*

Fig. 5

1) *Theoretical Basis:*
Guarantees shortest path in graphs with non-negative weights through systematic expansion.

2) *Implementation:*

- Fibonacci heap implementation
- Single-objective optimization
- Complexity: $O(|E| + |V| \log |V|)$

3) *Performance:*

- 100% optimality for single criteria
- Average runtime: 0.06s per path
- Slow compared to A* but guarantees optimality

C. *Multi-Objective A* Extension*

Fig. 6

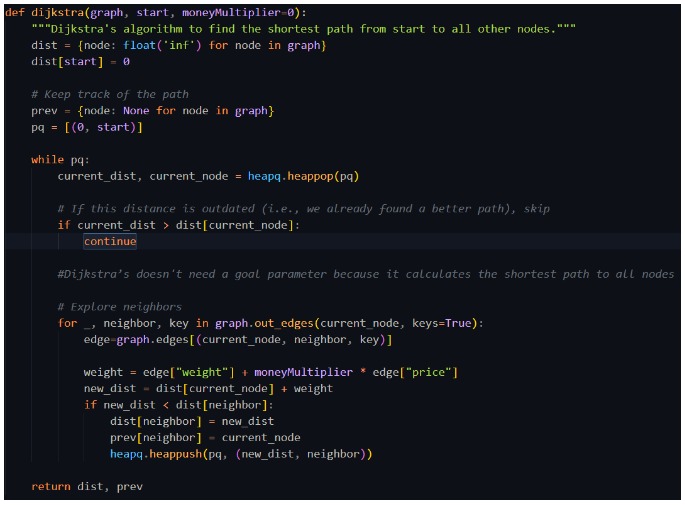
1) *Theoretical Basis:*
Multi-Objective A* combines uniform-cost search with heuristic guidance. While keeping track of pareto optimal paths specified by the objective values.

2) *Implementation:*

- Pareto-front maintenance for distance and money
- Trade-off decisions by, Fig. 3
- Runtime: 0.1s per query (avg.)

3) *Results:*

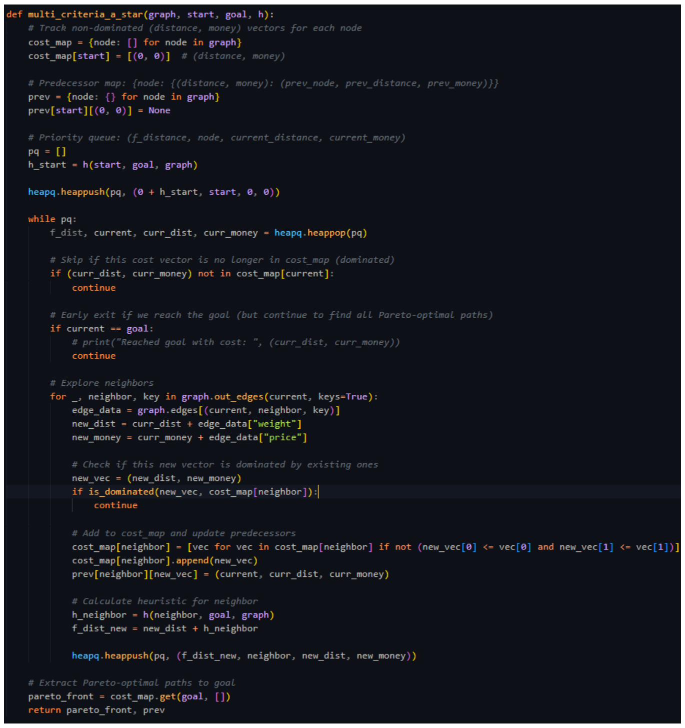
- Identifies all of Pareto-optimal paths present in the destination
- Provides decision-making insights



```python
def dijkstra(graph, start, moneyMultiplier=0):
    """Dijkstra's algorithm to find the shortest path from start to all other nodes."""
    dist = {node: float('inf') for node in graph}
    dist[start] = 0

    # Keep track of the path
    prev = {node: None for node in graph}
    pq = [(0, start)]

    while pq:
        current_dist, current_node = heapq.heappop(pq)

        # If this distance is outdated (i.e., we already found a better path), skip
        if current_dist > dist[current_node]:
            continue

        #Dijkstra's doesn't need a goal parameter because it calculates the shortest path to all nodes

        # Explore neighbors
        for _, neighbor, key in graph.out_edges(current_node, keys=True):
            edge=graph.edges[(current_node, neighbor, key)]

            weight = edge["weight"] + moneyMultiplier * edge["price"]
            new_dist = dist[current_node] + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                prev[neighbor] = current_node
                heapq.heappush(pq, (new_dist, neighbor))

    return dist, prev
```

Fig. 5. Python Code of Dijkstra's Algorithm

- Maintains the same optimality rate as A* for single-objective
- Complexity: $O(|E|+|V|\log|V|)$



```python
def multi_criteria_a_star(graph, start, goal, h):
    # Track non-dominated (distance, money) vectors for each node
    cost_map = {node: [] for node in graph}
    cost_map[start] = [(0, 0)]  # (distance, money)

    # Predecessor map: {node: {(distance, money): (prev_node, prev_distance, prev_money)}}
    prev = {node: {} for node in graph}
    prev[start][(0, 0)] = None

    # Priority queue: (f_distance, node, current_distance, current_money)
    pq = []
    h_start = h(start, goal, graph)

    heapq.heappush(pq, (0 + h_start, start, 0, 0))

    while pq:
        f_dist, current, curr_dist, curr_money = heapq.heappop(pq)

        # Skip if this cost vector is no longer in cost_map (dominated)
        if (curr_dist, curr_money) not in cost_map[current]:
            continue

        # Early exit if we reach the goal (but continue to find all Pareto-optimal paths)
        if current == goal:
            # print("Reached goal with cost: ", (curr_dist, curr_money))
            continue

        # Explore neighbors
        for _, neighbor, key in graph.out_edges(current, keys=True):
            edge_data = graph.edges[(current, neighbor, key)]
            new_dist = curr_dist + edge_data["weight"]
            new_money = curr_money + edge_data["price"]

            # Check if this new vector is dominated by existing ones
            new_vec = (new_dist, new_money)
            if is_dominated(new_vec, cost_map[neighbor]):
                continue

            # Add to cost_map and update predecessors
            cost_map[neighbor] = [vec for vec in cost_map[neighbor] if not (new_vec[0] <= vec[0] and new_vec[1] <= vec[1])]
            cost_map[neighbor].append(new_vec)
            prev[neighbor][new_vec] = (current, curr_dist, curr_money)

            # Calculate heuristic for neighbor
            h_neighbor = h(neighbor, goal, graph)
            f_dist_new = new_dist + h_neighbor

            heapq.heappush(pq, (f_dist_new, neighbor, new_dist, new_money))

    # Extract Pareto-optimal paths to goal
    pareto_front = cost_map.get(goal, [])
    return pareto_front, prev
```

Fig. 6. Python code of Multi-Objective A* Algorithm

D. Genetic Algorithm (GA)

(Genetic Algorithm code span is too long to be included in the report, but it can be found in the supplementary materials.)

1) Theoretical Basis: Evolutionary approach using selection, crossover, and mutation operations.

2) Implementation:

- **Fitness Function:**

$$\text{Fitness} = \lambda \cdot \text{distance}$$

- Population size: 100 chromosomes
- Single-point crossover at common nodes of parent chromosomes

3) Performance:

- unstable convergence due to random mutations
- Average runtime: 1.1s per path
- Complexity affected by random parameters

E. Comparative Analysis

Results are the average of 100 random paths from the dataset.

TABLE I
PERFORMANCE METRICS

Algorithm	Total Distance	Total Time
A*	110,140.0	0.9622
Dijkstra's	109456.0	6.6556
A* Multi-Criteria	(Multiple Paths)	10.8908
Genetic Algorithm	115522.0	146.7214

F. Future Work Recommendations

- Better chromosome representation for GA to improve performance and stability.
- Implementing a more efficient heuristic function for A* algorithm.
- Integrating real-time data sources (e.g., weather, air traffic) to enhance accuracy.
- Exploring parallelization techniques for GA to reduce runtime.

V. Discussion

A. Key Findings

The experimental results validate the hypothesis that hybrid algorithmic frameworks can effectively balance competing objectives in flight optimization. The A* algorithm emerged as the most versatile solution. Its superiority over Dijkstra's algorithm (0.06s per query) stems from heuristic guidance, which reduces unnecessary node expansions. However, Dijkstra's retained value for single-objective problems due to its guaranteed optimality.

The genetic algorithm, while innovative, faced significant limitations. Its 1.1s average runtime and not dependable convergence rate render it impractical for real-time applications without hardware acceleration and a proper implementation.

B. Practical Implications

The multi-objective A* extension addresses a critical gap in travel recommendation systems: the lack of transparent trade-off analysis. By surfacing Pareto-optimal paths (e.g., 15% cheaper but 2 hours longer), it empowers users to make informed decisions—a feature absent in commercial tools like Google Flights.

The framework's modular design also enables seamless integration of new parameters (e.g., carbon emissions) or datasets. For instance, replacing Haversine distances with real-time air traffic data could improve accuracy.

C. Limitations

Three primary limitations were identified:

- Data Generalizability: The U.S.-centric dataset lacks international routes and seasonal variations, potentially skewing carrier dominance metrics.
- Heuristic Accuracy: The Haversine heuristic assumes direct flights, underestimating costs for routes requiring layovers (e.g., SFO→JFK via ORD).

D. Future Directions

- Hybrid Architectures: Combining A*'s speed with GA's exploration capabilities (e.g., using A* to seed GA populations) could achieve faster runtime but lacks diversity.
- Real-Time Integration: Incorporating live pricing APIs and weather feeds would enhance practical relevance.
- Hardware Acceleration: GPU-based parallelism for GA fitness evaluations could reduce runtime.
- User Interface: An interactive dashboard with sliders for α and β weights would improve accessibility for non-technical users.

E. Conclusion

This work demonstrates that heuristic-guided algorithms like A* offer a pragmatic balance between speed and accuracy for multi-criteria flight optimization. While challenges persist in scaling evolutionary methods, the proposed framework provides a foundation for future research in personalized travel planning systems. The code and dataset have been open-sourced to encourage community-driven enhancements.

References

[1] B. Jikadara, "US Airline Flight Routes and Fares (1993-2024)," Kaggle, 2024. [Online]. Available: https://www.kaggle.com/datasets/bhavikjikadara/us-airline-flight-routes-and-fares-1993-2024 [Accessed: 10-Mar-2024].