**Technical Design Document: Graph-based Flight Network Construction**

This document details the design, data structures, algorithms, and complexity analysis for the construction of the flight network graph as required by the GlobalAir Hackathon.

**1. System Architecture and Flow**

The system is designed with a modular approach, separating configuration, data structures, and data loading logic for maintainability and clarity.

**Component Diagram**

The application is broken down into four distinct Python modules that work together to build the flight network.

+----------------+ +------------------+ +----------------+

| config.py |<---- | data\_loader.py |----> | graph.py |

| (Settings) | | (Builds Graph) | | (Defines Graph)|

+----------------+ +------------------+ +-------+--------+

^ |

| |

+-----+-----+ |

| main.py |-------------------+

| (Entry Pt)|

+-----------+

**Data Flowchart**

The process of constructing the graph follows a linear data flow, from reading raw files to producing a populated graph object.

START

|

V

Read Configuration (from config.py)

|

V

Pre-process Route Frequencies (from routes.dat)

|

V

Initialize Empty Graph Object

|

V

Load Airports as Nodes (from airports.dat)

|

V

Load Direct Flights as Weighted Edges (from routes.dat)

|

V

Return Populated Graph Object

|

V

END

**2. Data Structure Selection**

The primary data structure for representing the flight network is an

**Adjacency List**.

* **Structure**: The graph is implemented as a dictionary where keys are the IATA codes of source airports. The value for each key is a list of tuples, with each tuple containing the destination airport's IATA code and a dictionary of edge weights ((destination\_iata, {'cost': ..., 'delay': ...})).
* **Justification**: A flight network is inherently a **sparse graph**; most airports are not directly connected to each other.
  + An adjacency list is highly memory-efficient for sparse graphs, as it only stores existing connections. Its space complexity is O(V + E), where V is vertices (airports) and E is edges (routes).
  + Iterating over all direct flights from an airport is efficient, as it's a simple list traversal.
* **Trade-offs vs. Adjacency Matrix**:
  + An **Adjacency Matrix** would represent the graph as a V x V matrix, where M[i][j] stores the edge weight.
  + **Advantage**: Checking for the existence of a direct flight between two airports would be a constant time O(1) operation.
  + **Disadvantage**: The space complexity is O(V²), which is prohibitively large and wasteful for a sparse graph with over 6,000 airports (6000² = 36 million entries). The adjacency list is the clear choice for this use case.

**3. Algorithm Design and Justification**

**Distance Calculation: Haversine Formula**

* **Justification**: To calculate the distance-based edge weight, the Haversine formula was used. This is the standard mathematical approach for computing the great-circle distance between two points on a sphere from their latitudes and longitudes. For global flight paths, it provides high accuracy and is superior to simpler, less accurate methods.

**Edge Weight Simulation: Intelligent Randomization**

* **Justification**: This was necessary due to the absence of real cost or average delay data in the OpenFlights dataset. The simulation adds realism by incorporating factors like airline type and route frequency, making the graph more useful for subsequent path-finding algorithms.
* **Alternative**: The alternative was using a simple, uniform random value, which would be less realistic. The chosen simulation creates a more complex and interesting problem space for the subsequent path-finding algorithms.

**4. Core Function Analysis**

**build\_flight\_network() in data\_loader.py**

* **Time Complexity**: O(A+R)
  + Where A is the number of airports and R is the number of routes in the data files. The function iterates through each file once in a linear fashion.
* **Space Complexity**: O(A+R)
  + The function creates and returns the

Graph object, which stores data for every airport and route.

**Graph.add\_node() in graph.py**

* **Time Complexity**: O(1)
  + On average. This operation involves two dictionary insertions, which are constant-time operations on average.
* **Space Complexity**: O(1)
  + It stores a fixed amount of data per call.

**Graph.add\_edge() in graph.py**

* **Time Complexity**: O(1)
  + On average. Appending an item to a Python list is an amortized constant-time operation.
* **Space Complexity**: O(1)
  + It stores a single tuple containing references to existing data.

**5. Pseudocode**

**build\_flight\_network Algorithm**

FUNCTION build\_flight\_network():

// Initialization

graph = new Graph()

route\_frequencies = calculate\_frequencies\_from\_file(config.ROUTES\_FILE)

// Step 1: Load Airports

airport\_file = open(config.AIRPORTS\_FILE)

FOR each row IN airport\_file:

IF row is valid THEN

airport = create\_airport\_from\_row(row)

graph.add\_node(airport)

END IF

END FOR

// Step 2: Load Routes

route\_file = open(config.ROUTES\_FILE)

FOR each row IN route\_file:

IF route is a direct flight (stops == '0') THEN

source\_iata, dest\_iata, airline\_code = extract\_data\_from\_row(row)

// Calculate Weights

distance = haversine\_distance(source, dest)

cost = simulate\_cost(distance, airline\_code, route\_frequencies)

delay = simulate\_delay(airline\_code)

weights = {distance: distance, cost: cost, delay: delay}

graph.add\_edge(source\_iata, dest\_iata, weights)

END IF

END FOR

RETURN graph

END FUNCTION

**6. Benchmarks and Optimization**

The primary performance bottleneck of this module is the one-time data loading and graph construction process, which is dominated by file I/O. For the given dataset size, this process is acceptable for a single run.

* **Key Optimization**: The most impactful optimization is **Graph Persistence**.
  + **Benchmark**:
    - **Initial Run (Building from scratch)**: Can take 10-20 seconds depending on the machine, as it involves parsing over 70,000 lines of text data.
    - **Subsequent Runs (Loading a pre-built graph)**: By saving the populated Graph object using a library like pickle, subsequent startups can be reduced to **under 1 second**.
  + **Rationale**: This avoids the costly file I/O and data processing steps entirely, making the application feel instantaneous after the first run. This is crucial for an interactive system or for development and testing.