Flight Interconnection Analysis Using Apache Spark and GraphFrames

Juan Gonzalo Quiroz Cadavid[1], Priit Peterson[1], Shivam Maheshwari[1], and Venkata Narayana Bommanaboina[1]

[1]University of Tartu
juangonzalo@ut.ee, shivammahe21@gmail.com, bvnarayana515739@gmail.com, priit.petersonest@gmail.com

April 13, 2025

Graph Construction Methodology

Before we could analyze the flight data with graph algorithms, we had to convert it into a GraphFrames-compatible representation using Apache Spark. The graph comprises two main components:

- **Vertices:** Representing unique airports. These were extracted from both origin and destination airport fields.
- **Edges:** Representing individual flights. Each flight record contributes a directed edge from the origin to the destination airport, enriched with relevant metadata.

The resulting graph supports analyzing connectivity, influence, and clustering.

Query 0: Graph Construction

Vertices and Edges Definition

The dataset contains flight records with origin and destination airport codes. From these, we constructed:

- **Vertices:** Each unique airport from either the ORIGIN or DEST column forms a vertex, identified by its airport code.
- **Edges:** Each flight record creates a directed edge from the origin airport to the destination airport. The edge also includes metadata like flight date, distance, and cancellation status.

```
flight_vertex_df = (flight_df
    .filter(F.col("ORIGIN").isNotNull() & F.col("DEST").isNotNull())
    .select(F.col("ORIGIN").alias("id"))
    .union(flight_df.select(F.col("DEST").alias("id")))
    .distinct())
```

Listing 1: Creating Vertices from ORIGIN and DEST

```
flight_edges_df = (flight_df
    .filter(F.col("ORIGIN").isNotNull() & F.col("DEST").isNotNull())
    .select(
        F.col("ORIGIN").alias("src"),
        F.col("DEST").alias("dst"),
        F.col("FL_DATE"),
        F.col("CANCELLED"),
        F.col("ARR_TIME"),
        F.col("DISTANCE")
    ))
```

Listing 2: Creating Edges from Flight Records

```
from graphframes import GraphFrame
flight_graph = GraphFrame(flight_vertex_df, flight_edgs_df)
```

Listing 3: Constructing the Graph

Graph Overview

The resulting graph contains 296 vertices and over 2.7 million directed edges. This representation enables us to apply advanced network analysis techniques such as centrality, connected components, and PageRank.

```
GraphFrame(v:[id: string],
           e:[src: string, dst: string,
              FL_DATE, CANCELLED,
              ARR_TIME, DISTANCE])
```

We used this graph as the basis for all subsequent queries and analysis.

Query 1: Computing Graph Statistics

To understand the flight network's structure, we looked at how airports connect by computing a few metrics, these metrics help identify major hub airports and measure how interconnected the network is. We focused on:

- **In-Degree**: Number of incoming flights to each airport.
- **Out-Degree**: Number of outgoing flights from each airport.
- **Total Degree**: Sum of in-degree and out-degree, representing overall connectivity.
- **Triangle Count**: Number of distinct triangles, where three airports form a closed loop of direct flights.

We used both built-in GraphFrames functions and our own DataFrame operations to compute these metrics, helping us verify results and showcase flexible implementation options.

In-Degree Computation

We computed the in-degree using pattern matching to identify destination nodes.

```
from pyspark.sql.types import IntegerType
in_edges_df = (flight_graph .find("()-[edge]->(a)")
.groupBy("a.id") .count() .withColumn("inDegree", F.col("count").cast(IntegerType()))
.select(F.col("id"), F.col("inDegree")) )
```

Listing 4: In-degree calculation

Out-Degree Computation

The out-degree measures the number of flights departing from each airport:

```
outgoing_edges_df = (flight_graph .find("(a)-[edge]->()").groupBy("a.id") .count()
.withColumn("outDegree", F.col("count").cast(IntegerType()))
.select(F.col("id"), F.col("outDegree")) )
```

Listing 5: Out-degree calculation

Total Degree Computation

The total degree is computed by summing the values of the in-degree and out-degree:

```
degrees_df = (outgoing_edges_df .join(in_edges_df, "id") .withColumn("degree", F.col("outDegree")
+ F.col("inDegree")) .select(F.col("id"), F.col("degree")) )
```

Listing 6: Total degree calculation

These degree metrics help rank airports by overall activity and connectivity.

Triangle Count Computation

To identify triangular routes (cyclic paths of three distinct airports), we used an efficient self-join method on undirected edges. This method eliminates directionality and ensures each triangle is counted only once by enforcing `v1 < v2 < v3` ordering.

```
undirected_edges = (flight_edgs_df .select( F.when(F.col("src") < F.col("dst"), F.col("src")).
  otherwise(F.col("dst")).alias("v1"),
F.when(F.col("src") < F.col("dst"), F.col("dst")).otherwise(F.col("src")).alias("v2") ).
  dropDuplicates())
e1 = undirected_edges.alias("e1") e2 =
undirected_edges.alias("e2") e3 = undirected_edges.alias("e3")
triangle_candidates = e1.join(e2, F.col("e1.v2") == F.col("e2.v1"))
triangles = triangle_candidates.join( e3, (F.col("e1.v1") == F.col("e3.v1")) & (F.col("e2.v2") == F.
  col("e3.v2")) )
triangles = triangles.filter( (F.col("e1.v1") < F.col("e1.v2")) & (F.col("e1.v2") < F.col("e2.v2"))
  )
triangle_vertices = (triangles.select(F.col("e1.v1").alias("id"))
.union(triangles.select(F.col("e1.v2").alias("id")))
.union(triangles.select(F.col("e2.v2").alias("id"))))
vertex_triangle_counts = triangle_vertices.groupBy("id").agg(F.count("\*").alias("triangle_count"))
```

Listing 7: Triangle count calculation

Counting triangles reveals locally dense areas of the network, highlighting clusters of airports with frequent mutual connections.

Query 2: Total Number of Triangles in the Graph

In this query, we calculated the total number of triangles present in the flight graph indicating closely interlinked airports that can form alternative routes if one segment is disrupted.

Triangle Count Computation

Using the triangle detection logic established in Query 1, we computed the global triangle count by counting the resulting triangle structures:

```
triangle_total_count = triangles.count()
print("Total␣number␣of␣triangles␣(overall):", triangle_total_count)
```

Listing 8: Total triangle count calculation

Results

```
Total number of triangles (overall): 16015
```

This value indicates that over 16,000 triangular subgraphs exist within the network, showing many alternate routing options for passengers between key nodes. This is important for understanding both robustness and route planning in aviation networks.

Query 3: Eigenvector Centrality

Eigenvector centrality measures how influential each airport is by factoring in the influence of the airports it connects to. It is effective in identifying influential hubs that are well-integrated within the network. It can be very useful in air traffic analysis often derive importance not just from volume, but also from their strategic positioning in the network.

Computation Method

We implemented the power iteration method using Spark DataFrame operations. Each airport (vertex) starts with a rank of 1.0 and updates its value iteratively based on the rank of its inbound neighbors. The process continues until convergence or a defined maximum number of iterations.

```
eigen_ranks = flight_vertex_df.select("id").withColumn("rank", F.lit(1.0))

max_iter = 10
epsilon = 1e-4

for i in range(max_iter):
    contribs = flight_edgs_df.join(eigen_ranks, flight_edgs_df.src == eigen_ranks.id) \
        .groupBy("dst") \
        .agg(F.sum("rank").alias("contribution"))

    new_ranks = flight_vertex_df.join(contribs, flight_vertex_df.id == contribs.dst, "left") \
        .select(flight_vertex_df.id, F.coalesce(F.col("contribution"), F.lit(0.0)).alias("new_rank")
)

    total_rank = new_ranks.agg(F.sum("new_rank")).first()[0]
    new_ranks = new_ranks.withColumn("new_rank", F.col("new_rank") / total_rank)

    diff = eigen_ranks.join(new_ranks, "id") \
        .withColumn("diff", F.abs(F.col("rank") - F.col("new_rank")))
    max_diff_value = diff.agg(F.max("diff")).first()[0]
    print(f"Iteration {i + 1}, Max Rank Change: {max_diff_value}")

    if max_diff_value < epsilon:
        break

    eigen_ranks = new_ranks.withColumnRenamed("new_rank", "rank")

eigen_ranks.orderBy(F.col("rank").desc()).show(10, truncate=False)
```

Listing 9: Eigenvector centrality calculation

Results

```
+---+--------------------+
| id|                rank|
+---+--------------------+
|ATL| 0.03753374418630729|
|ORD| 0.03420041245905938|
|DFW|0.029195453879788374|
|DEN| 0.02846066233421956|
|LAX|0.027571457107331145|
|PHX| 0.02469136002748945|
|LAS|0.023324443940525116|
|IAH|0.020739376543623195|
|SFO| 0.01988898809777415|
|BOS| 0.01879929142273175|
+---+--------------------+
```

Airports such as ATL, ORD, and DFW scored highly, confirming their centrality in the U.S. flight network. Eigenvector centrality highlights not just busy airports, but those connected to other well-connected hubs—making it effective for identifying strategic nodes.

Query 4: PageRank Implementation

PageRank also measures airport importance but adds a probability that a traveler might randomly jump to any other airport, making it well-suited to modeling real passenger behavior.

Computation Method

We implemented the PageRank algorithm manually in Spark using a damping factor of 0.85, a standard value that reflects the idea that travelers typically follow connections 85% of the time and choose a random airport 15% of the time.

```python
damping = 0.85
max_iter = 10
epsilon = 1e-4

out_degrees = flight_edgs_df.groupBy("src").agg(F.count("dst").alias("out_degree"))
page_ranks = flight_vertex_df.select("id").withColumn("rank", F.lit(1.0))

for i in range(max_iter):
    edge_contribs = flight_edgs_df \
        .join(page_ranks, flight_edgs_df.src == page_ranks.id) \
        .join(out_degrees, flight_edgs_df.src == out_degrees.src) \
        .select(flight_edgs_df.dst, (F.col("rank") / F.col("out_degree")).alias("contrib"))

    summed_contribs = edge_contribs.groupBy("dst").agg(F.sum("contrib").alias("total_contrib"))

    new_page_ranks = flight_vertex_df \
        .join(summed_contribs, flight_vertex_df.id == summed_contribs.dst, "left") \
        .select(flight_vertex_df.id,
                ((1 - damping) + damping * F.coalesce(F.col("total_contrib"), F.lit(0.0))).alias("
new_rank"))

    diff = page_ranks.join(new_page_ranks, "id") \
        .withColumn("diff", F.abs(F.col("rank") - F.col("new_rank")))
    max_diff_value = diff.agg(F.max("diff")).first()[0]
    print(f"Iteration {i + 1}, Max Rank Change: {max_diff_value}")

    if max_diff_value < epsilon:
        break

    page_ranks = new_page_ranks.withColumnRenamed("new_rank", "rank")

page_ranks.orderBy(F.col("rank").desc()).show(10, truncate=False)
```

Listing 10: PageRank calculation

Results

```
+---+------------------+
|id |rank              |
+---+------------------+
|ATL|18.905010413236592|
|ORD|12.92711042270102 |
|DFW|11.735613068886858|
|DEN|9.998483451765958 |
|LAX|7.726101869874977 |
|IAH|7.159305038125105 |
|PHX|7.0653715625994264|
|SLC|7.038754134167003 |
|DTW|7.019968536748528 |
|SFO|5.904248175402836 |
+---+------------------+
```

Similar to eigenvector centrality, the PageRank results also point to ATL, ORD, and DFW as key airports in the network. However, by incorporating random jump behavior, PageRank better models user navigation patterns and can assist with travel demand forecasting or influence propagation in route optimization.

Note: GraphFrames also supports a built-in PageRank function. We also manually implemented PageRank to deepen our understanding of the algorithm's behavior and convergence.

Query 5: Connected Components

To identify the most interconnected regions within the flight network, we computed connected components using a custom label propagation approach. Each connected component represents a group of airports where every airport is reachable from every other airport in the group through direct or indirect flights.

Computation Method

We implemented a Union-Find style label propagation algorithm. This approach is efficient and scalable for distributed data processing.

- All flight edges were converted into undirected form by including both directions (i.e., source-to-destination and destination-to-source).
- Each airport (vertex) was initially labeled with its own ID, forming its own component.
- In each iteration, a vertex updated its component label to the smallest value among itself and its neighbors.
- The process continued until no component labels changed between iterations, signaling convergence.

```
undirected_edges = (flight_edgs_df.select("src", "dst")
    .union(flight_edgs_df.select(F.col("dst").alias("src"), F.col("src").alias("dst")))
    .dropDuplicates())
```

Listing 11: Creating Undirected Edges for Connectivity Analysis

```
vertices_df = flight_vertex_df.withColumn("component", F.col("id"))
```

Listing 12: Initializing Vertices with Component Labels

```
max_iter = 10
iteration = 0
converged = False

while not converged and iteration < max_iter:
    neighbor_components = undirected_edges.join(vertices_df, undirected_edges.dst == vertices_df.id, "left") \
        .groupBy("src") \
        .agg(F.min("component").alias("min_neighbor_component"))

    updated_vertices = vertices_df.join(neighbor_components, vertices_df.id == neighbor_components.src, "left") \
        .withColumn("new_component", F.when(F.col("min_neighbor_component").isNotNull(),
                                            F.least(F.col("component"), F.col("min_neighbor_component")))
                    .otherwise(F.col("component"))) \
        .select("id", "new_component")

    diff_count = updated_vertices.filter(F.col("new_component") != vertices_df.component).count()
    if diff_count == 0:
        converged = True
    else:
        vertices_df = updated_vertices.withColumnRenamed("new_component", "component")
        iteration += 1
```

Listing 13: Label Propagation to Identify Connected Components

```
components_group = vertices_df.groupBy("component").count().orderBy("count", ascending=False)
components_group.show()
```

Listing 14: Final Component Groups

Results and Interpretation

The algorithm revealed that all 296 airports belong to a single connected component:

```
+---------+-----+
|component|count|
+---------+-----+
|ABE      |  296|
+---------+-----+
```

The result shows that all 296 airports form one large connected component. Any airport is reachable from any other airport through a series of flights