Graph Gurus 26 demo instructions

This exercise from Graph Gurus 26 demonstrates both unweighted and weighted shortest path algorithms.

These instructions accompany the video recording of Graph Gurus 26.
These instructions should be distributed along with a GraphStudio solution file (export_GG26_path.tar.gz) and a data folder containing 4 files. Optionally, it may include several queries; however, the queries are included in the solution file.

Install the solution file, which bundles the schema, data, loading jobs, and queries used in this exercise.

(1) From the main page of GraphStudio. Import the solution file export_GG26_jan20.tar.gz.
(2) Go to the Map Data to Graph Page.
    (a) In the top menu, click on the Add Data File icon. One at a time, click the (+) icon to add each of the 4 data files. Within the Add Data File panel, click each file name, select the Has header checkbox, and click ADD at the bottom right, to load the data file.
    (b) Note that this will place a duplicate file icon for each file on the schema diagram. You can delete these duplicates (they ones which are not linked to a vertex or edge type).
    (c) Finally, click the Publish Data Mapping up-arrow button.
(3) Go to the Load Data page. Wait a few seconds for the Load (right arrow) button to become active. Click it and wait for the data to be loaded.
(4) Go to the Write Queries page. Click the Install All Queries button, to the left of the "GSQL Queries" heading.

Dataset:
Global commercial airline routes from https://openflights.org/data.html. Did some minor cleanup (replaced "\n" with "", which had represented NULL, and added a header row.)

- 7,698 airports (airports.dat, cleaned up and renamed airports.csv)
- 67,664 direct flight routes (cleaned up and renamed routes.csv)

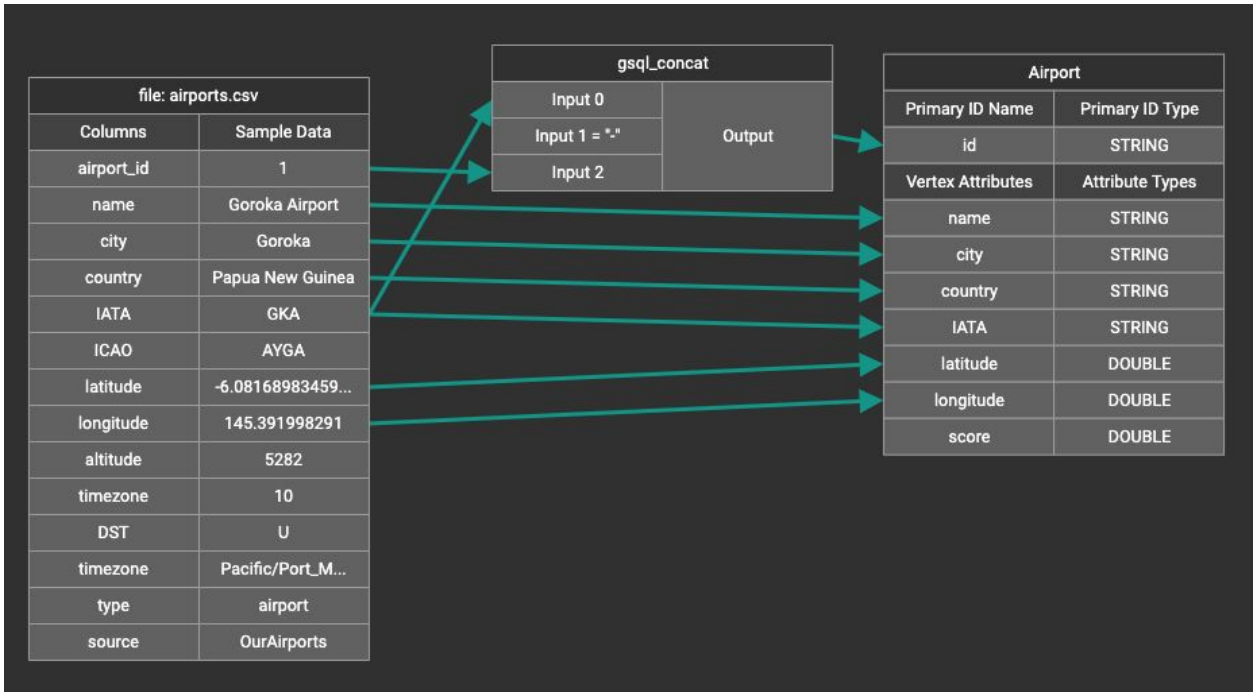
Also manually created a separate small graph of fake airports and routes, for a small-scale demo (airports-path-demo.csv and routes-path-demo.csv)

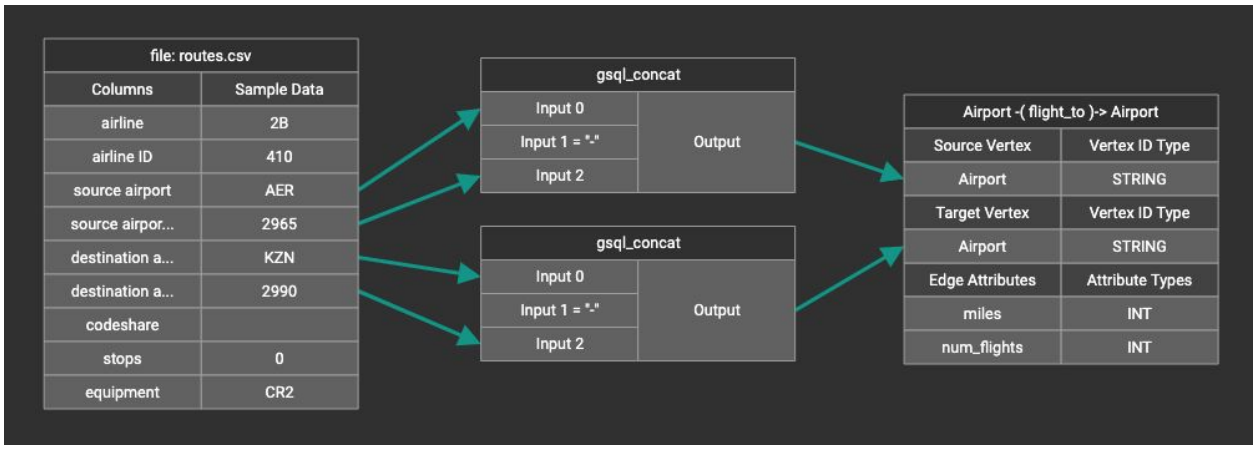
Schema:

```
VERTEX Airport(PRIMARY_ID id STRING, name STRING, city STRING, country STRING, IATA
STRING, latitude DOUBLE, longitude DOUBLE, score DOUBLE)

UNDIRECTED EDGE flight_route (FROM Airport, TO Airport, miles INT)

//DIRECTED EDGE   flight_to    (FROM Airport, TO Airport, miles INT, num_flights INT)
             WITH REVERSE_EDGE reverse_flight_to
```

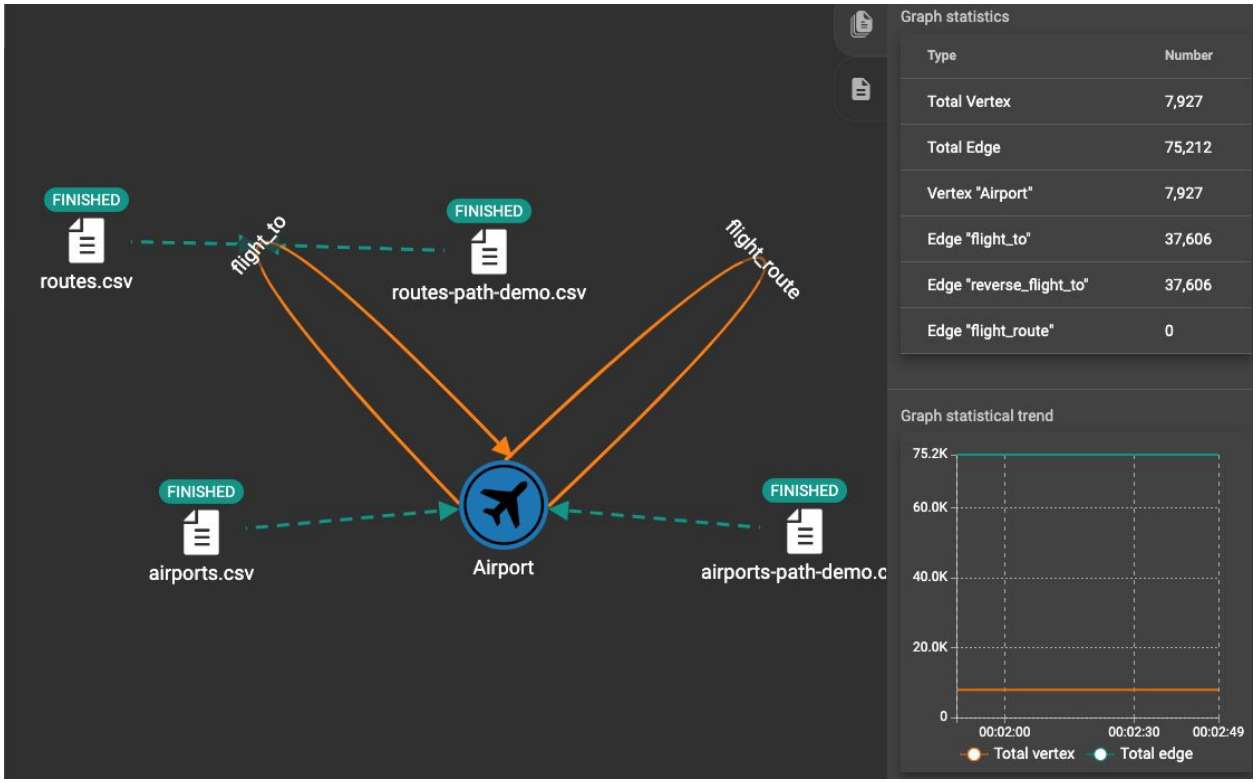
Data Mapping and Loading:

These figures show how we mapped the CSV data to the vertex and edge types. For the Airports, notice how we constructed the ID by concatenating the IATA code with the airport_id (openflight.org's numbering). We did this because while the IATA is the most recognizable (e.g., SFO, LAX, JFK), some smaller airports in the dataset do not have an IATA code.





Here are the vertex and edge counts after loading. We have fewer airports than expected. Some of the datalines must have had formatting or data value problems, but since this is just a demo, we didn't bother to trace the cause. There are significantly fewer routes than lines in the routes.csv file. This is because the data file sometimes has multiple routes between two cities (e.g. LAX to JFK), but we counted each route only once.



Data Preparation before Querying

The data from openflights.org does not tell us the distance between airports, but we can compute this from the latitude and longitude information, using the Haversine formula. We wrote a query to apply the Haversine formula for each edge and then store the result in the the edge's *miles* property. Notice how similar the GSQL code in addWeights looks to an implementation in Python.

Shortest Path Queries

We used two algorithms which are in our GSQL Graph Algorithm Library:
https://github.com/tigergraph/gsql-graph-algorithms

Note: We used our older versions of algorithms which start in a template format
https://github.com/tigergraph/gsql-graph-algorithms/tree/master/algorithms/templates
and then you run an installer.sh script to specify vertex types, edge types, relevant attributes, and output type.
The algorithms you find here have already been customized for the Airport Route dataset.

We suggest you try the new schema-free versions
https://github.com/tigergraph/gsql-graph-algorithms/tree/master/algorithms/schema-free
which don't need to be customized in advance. You simply specify the vertex types and edge types when you run it. (Currently, the schema-free algorithms are only available for algorithms which do not refer to a specific attribute.)

shortest_ss_no_wt:
The name is an abbreviation for "Shortest Path, Single Source, No Weights." This means it computes the shortest paths from a single source vertex to EVERY OTHER VERTEX in the graph, and the edges are unweighted. Since they are unweighted, "shortest path" is the same as "fewest hops". The algorithm is Breadth First Search, with some additions to record the path and number of hops. With our airline route dataset, this query tells you how to get from a given airport to any other airport, with the fewest stops.

The query takes two parameters, the source vertex and whether to display the results visually or not. GraphStudio currently has a limit of displaying about 5000 edges.

shortest_ss_pos_wt:
The name is an abbreviation for "Shortest Path, Single Source, Positive Weights." The conventional algorithm would be Dijkstra's, but Dijkstra's requires sequential execution. We want to take advantage of GSQL's parallelism, so we used the Bellman-Ford Algorithm (without the extra check for negative length loops, because we don't have negative weights).

To improve the output, we modified the algorithm, **creating shortest_ss_pos_wt_limits**. We added two parameters:

- maxHops: the maximum number of hops. We don't want airport itineraries with more than say 3 or 4 hops. This parameter is used in a modified WHILE loop.
- maxDest: the maximum number of destination vertices in the output. We added a SQL-like block which uses ORDER BY and LIMIT to sort and limit the output:

```
total = SELECT s # Sort VSET
        FROM total:s
        ORDER BY s.@minPath.top().dist ASC
        LIMIT maxDest;   #Limit the size of the output set
```

Bellman-Ford does take more steps than Dijkstra's, so running it will take longer on very large graphs. Use maxHops judiciously.