Project Statement for Milestone 4

Airline Search Engine

CPTS-415_BMW

Brian Joo, Noah Waxman, Mitchell Kolb

Overview:

At this stage of the project, teams should have implemented and tested their algorithms on the reduced data set. The algorithms should include at least 2 built-in algorithms (using database queries and/or Hadoop/Spark functions), and at least 2 custom algorithms (built from ground up). Team may have worked on a plan for implementing the user interface of the application.

In this report, the teams are expected to report on their algorithms and results. They are also expected to describe their plan for the user interface.

Report Topics:

The report should cover the following subtopics and answer the questions listed:

1. Algorithm Description:
    a. Give a formal description of each of the algorithms you have implemented. It should consist of (1) input; (2) output, and (3) (computing) operations.

__2 Built-in Algorithms:__

1st built-in Algorithm: Which country (or) territory has the highest number of Airports

Input:

- Airport: This table contains data for all things airports.

Output:

- 1 row with 2 columns labeled CountryName and NumberOfAirports

Operations

- MATCH (a:Airport)
- RETURN a.country AS Country, count(a) AS NumberOfAirports
- ORDER BY NumberOfAirports DESC
- LIMIT 1

2nd built-in Algorithm: Top K cities with most incoming/outgoing airlines

Input:

- Tables: Airports and Routes
- Created edges Departs_from and Arrives_at which are matched from Airport and Route by the “iata” attribute

Output:

- If K is 5 for example this will return the top 5 cities. You can change the limit to any value but in each K value row it will return two columns, city and numberOfRoutes.

Operations:

- MATCH (a:Airport)-[:DEPARTS_FROM]->(r:Route)<-[:ARRIVES_AT]-(d:Airport)
- WITH a.city AS City, count(r) AS NumberOfRoutes
- RETURN City, NumberOfRoutes
- ORDER BY NumberOfRoutes DESC
- LIMIT K

1st Custom Algorithm:

Breadth-First Search (BFS) algorithm:

Input:

- city: A string representing the starting city (node) for the BFS traversal.
- d: An integer representing the maximum depth (distance) to which the BFS should explore in the graph.

Output:

- A list of lists, where each inner list represents a path from the starting city to a city at depth d or less in the graph

Operations:

- BFS algorithm is executed in a Neo4j Cypher query
- MATCH (start:Airport {city: $city})
  - This part of the query matches the node labeled as "Airport" with the property "city" equal to the value of the city parameter, which serves as the starting point for the BFS traversal.
- CALL apoc.path.spanningTree(...) YIELD path
  - The apoc.path.spanningTree procedure is called to find a spanning tree in the graph starting from the specified node. This procedure allows you to specify the minimum and

        - maximum levels (depths) to traverse, and it uses the "FLIGHT>" relationship type as a filter
    - RETURN [node in nodes(path) | node.city] as cities
        - This part of the query returns a list of cities (nodes) in the path as "cities," extracting the "city" property from each node
    - The result of the Cypher query is a set of paths in the form of nodes' cities
    - We extract these paths from the Cypher result and return them as a list of lists, where each inner list represents a path from the starting city to a city at depth d or less

2nd Custom Algorithm:

Shortest Paths Between All Pairs of Vertices

Input:

- The weighted graph is my input as represented in a 2D array/matrix. Each row and column should represent the shortest distance from vertex A to vertex B for example

Output:

- A matrix that is the shortest paths between each pair of vertices

Operations:

- Initialize a shortest paths matrix using the weights of the edges in the graph.
    - To get information from the db I use this query to neo4j
    - MATCH (origin:Airport)-[r:DISTANCE]->(destination:Airport)
        - This finds all the pairs of airports nodes that are connected in the DISTANCE relationship that I made for this algo
    - WHERE origin.id < destination.id
        - This excludes pairs where the id of the origin is not less that the id of the destinations so I don't get copies of node pairs
    - RETURN origin.id AS origin_id, destination.id AS destination_id, r.distance AS distance
        - This is what I want to return.
- It iterates over all pairs of vertices in the graph. For each pair, it updates the shortest distance between them by considering all other vertices in the graph as potential intermediate vertices.
- I use the pyspark dataframe as shown in lecture to store the results after the step in the algorithm.

b. Discuss any optimization techniques you have implemented.

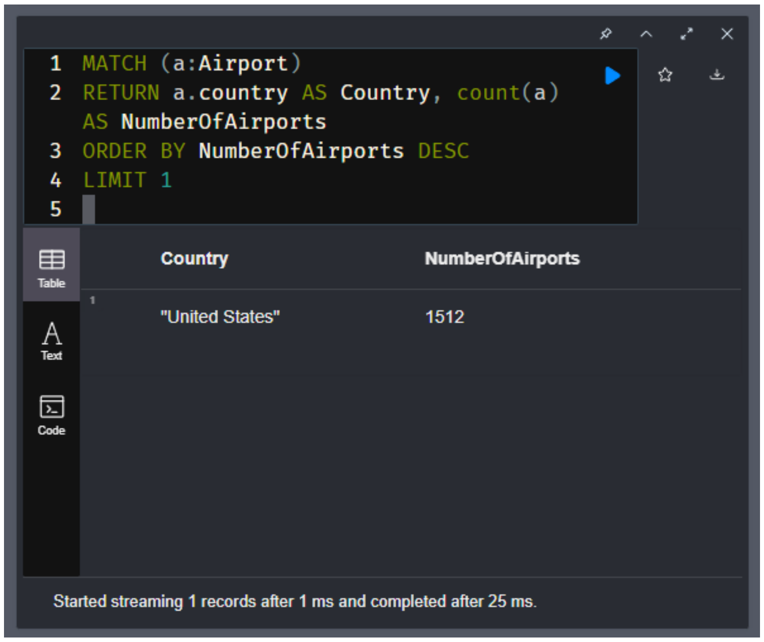
- For the two built in algorithms we have determined that without rearranging the way that our relationships are the optimization will be minimal.
- For the 2 custom algorithms we use efficient data structures to store and manipulate the data that is returned to our local systems. For the 2nd custom algorithm it is using the shortest path between two pairs algorithm or as known as the Floyd-Warshall technique. This way of manipulating the data is as optimized at our scale as we can get it. That algorithm runs at O(n^3) where n is the number of vertices in our whole graph.

2. Algorithm Results:

a. Provide algorithm results (output) using sample data (inputs). Also, include the performance metrics (execution time, accuracy, etc.) for the sample data.

<u>2 Built-in Algorithms:</u>

1st built-in Algorithm: Which country (or) territory has the highest number of Airports



```
MATCH (a:Airport)
RETURN a.country AS Country, count(a)
AS NumberOfAirports
ORDER BY NumberOfAirports DESC
LIMIT 1
```

	Country	NumberOfAirports
1	"United States"	1512

Started streaming 1 records after 1 ms and completed after 25 ms.

Input: Airports table

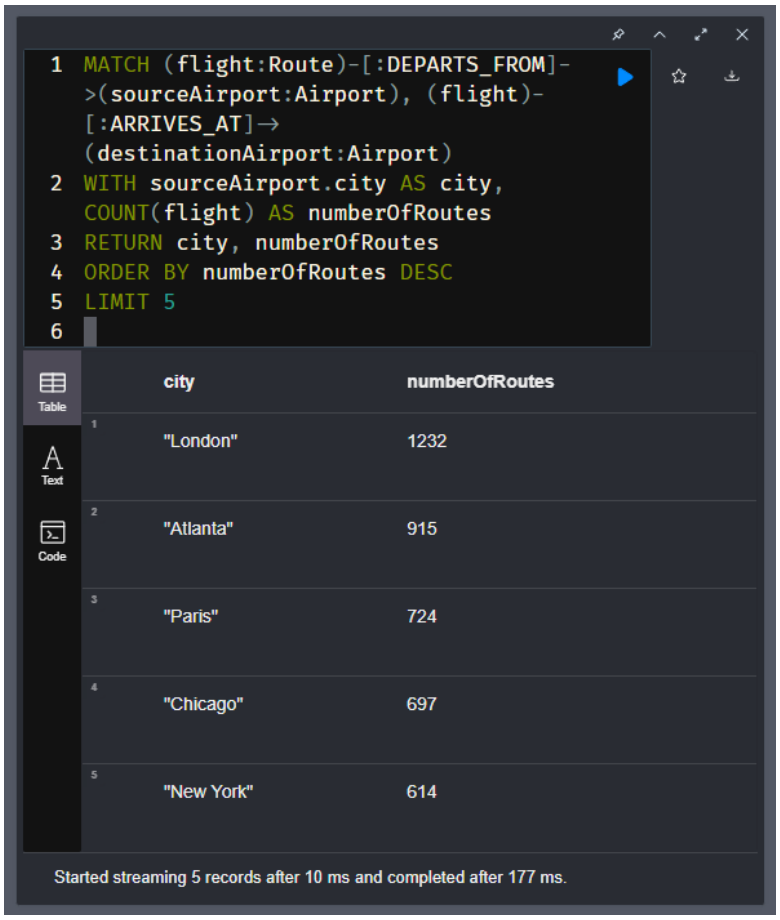
Output:

Country: United States

NumberOfAirports: 1512

Execution time: 25ms

2nd built-in Algorithm: Top K cities with most incoming/outgoing airlines



```
MATCH (flight:Route)-[:DEPARTS_FROM]-
>(sourceAirport:Airport), (flight)-
[:ARRIVES_AT]→
(destinationAirport:Airport)
WITH sourceAirport.city AS city,
COUNT(flight) AS numberOfRoutes
RETURN city, numberOfRoutes
ORDER BY numberOfRoutes DESC
LIMIT 5
```

	city	numberOfRoutes
1	"London"	1232
2	"Atlanta"	915
3	"Paris"	724
4	"Chicago"	697
5	"New York"	614

Started streaming 5 records after 10 ms and completed after 177 ms.

- **Input**: Airports node table, Routes node table, “Arrive_at” relationship, “Departs_from” relationship
- **Output:**

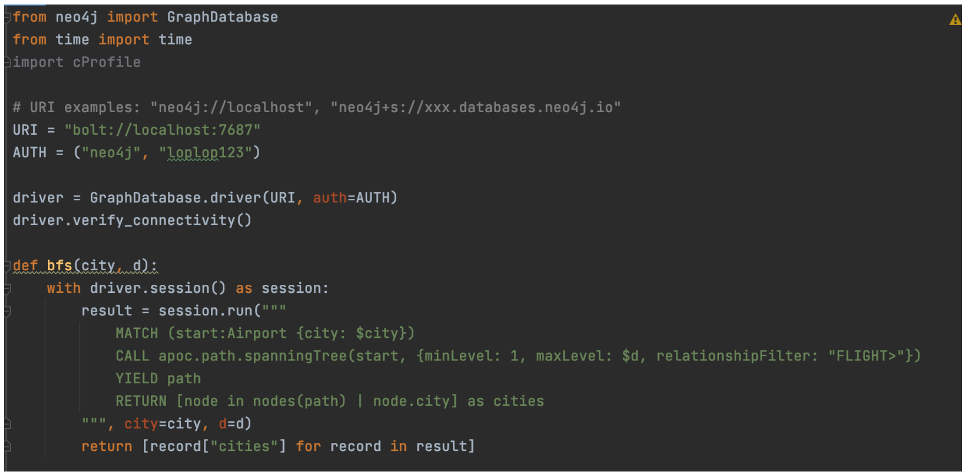
City:	London
NumberOfRoutes:	1232
City:	Atlanta
NumberOfRoutes:	915
City:	Paris
NumberOfRoutes:	724
City:	Chicago
NumberOfRoutes:	697
City:	New York

NumberOfRoutes: 614

- **Execution time**: 177ms

1st Custom Algorithm:

D-Bound reachability algorithm:



```
from neo4j import GraphDatabase
from time import time
import cProfile

# URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
URI = "bolt://localhost:7687"
AUTH = ("neo4j", "loplop123")

driver = GraphDatabase.driver(URI, auth=AUTH)
driver.verify_connectivity()

def bfs(city, d):
    with driver.session() as session:
        result = session.run("""
            MATCH (start:Airport {city: $city})
            CALL apoc.path.spanningTree(start, {minLevel: 1, maxLevel: $d, relationshipFilter: "FLIGHT>"})
            YIELD path
            RETURN [node in nodes(path) | node.city] as cities
        """, city=city, d=d)
        return [record["cities"] for record in result]
```

Input: bfs("Spokane", 2)

Output:

[['Spokane', 'Salt Lake City'], ['Spokane', 'Boise'], ['Spokane', 'Minneapolis'], ['Spokane', 'Denver'], ['Spokane', 'Los Angeles'], ['Spokane', 'Phoenix'], ['Spokane', 'Oakland'], ['Spokane', 'Seattle'], ['Spokane', 'Portland'], ['Spokane', 'Las Vegas'], ['Spokane', 'Salt Lake City', 'San Diego'], ['Spokane', 'Salt Lake City', 'Dallas-Fort Worth'], ['Spokane', 'Salt Lake City', 'Billings'], ['Spokane', 'Salt Lake City', 'Sacramento'], ['Spokane', 'Salt Lake City', 'Jacksn Hole'], ['Spokane', 'Salt Lake City', 'Moab'], ['Spokane', 'Salt Lake City', 'Philadelphia'] …. rest omitted.

This data is in fact accurate.

Execution time: 0.028134 seconds

Input: bfs("New York", 2)

Output:

[['New York', 'Grand Rapids'], ['New York', 'Dallas-Fort Worth'], ['New York', 'Key West'], ['New York', 'Miami'], ['New York', 'Little Rock'], ['New York', 'West Palm Beach'], ['New York', 'Charlottesville Va'], ['New York', 'Toronto'], ['New York', 'Burlington'], ['New York', 'St. Louis'], ['New York', 'Knoxville'], ['New York', 'Chicago'], ['New York', 'Richmond'], ['New York', 'Columbus'], ['New York', 'Louisville'], ['New York', 'Washington'], ['New York', 'Pittsburgh'], ['New York', 'Dayton'], ['New York', 'Tampa'], ['New York', 'New Orleans'], ['New York', 'Sarasota'], ['New York', 'Montreal'], ['New York', 'Minneapolis'], ['New York', 'Omaha'], ['New York', 'Nashville'], ['New York', 'Boston'], ['New York', 'Chicago'], ['New York', 'Oranjestad'], ['New York', 'Wilmington'], ['New York', 'Fort Lauderdale'], ['New York', 'Washington'], ['New York', 'Bentonville'] …. rest omitted.

This data is in fact accurate.

Execution time: 0.169224 seconds

Input: bfs("Seattle", 1)

Output:

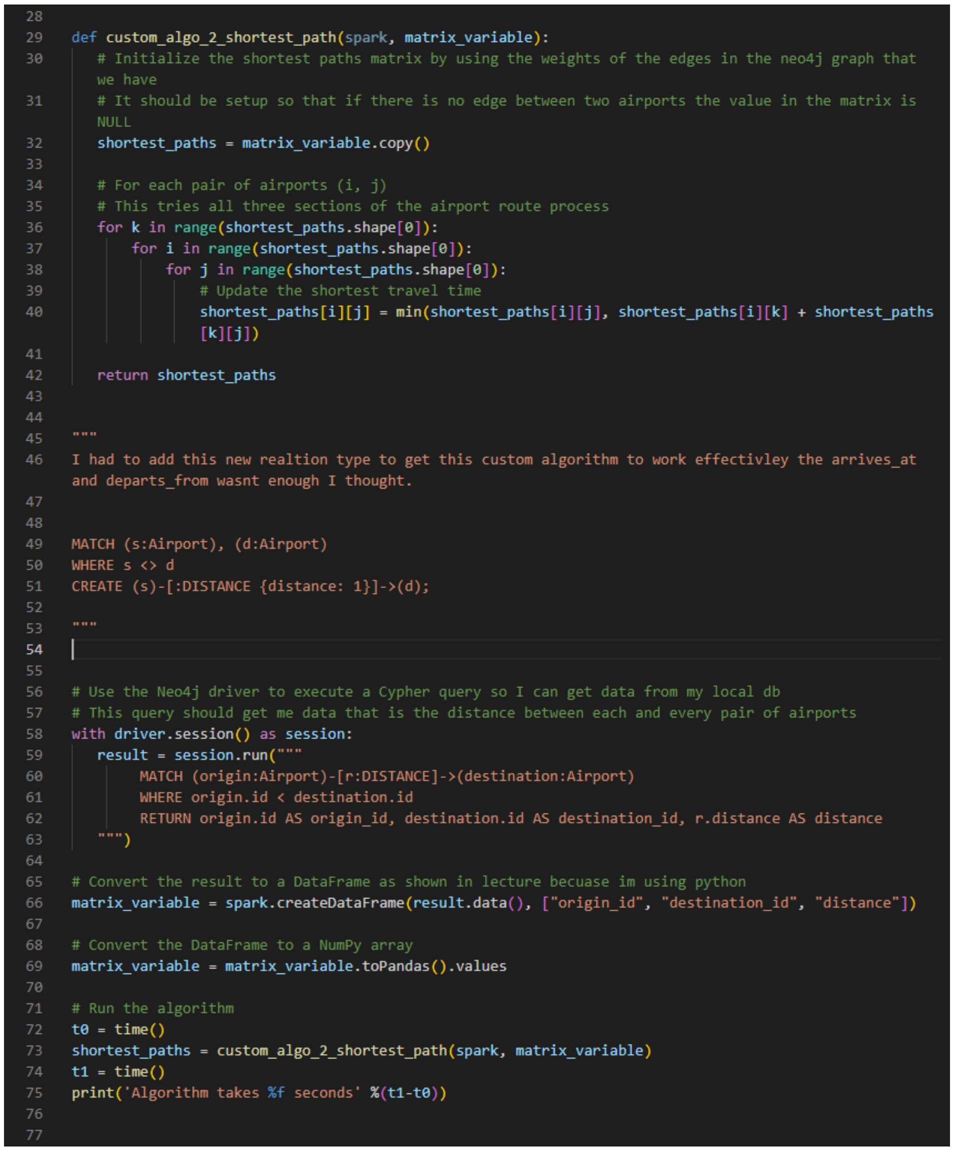
[['Seattle', 'Atlanta'], ['Seattle', 'Philadelphia'], ['Seattle', 'Boston'], ['Seattle', 'Las Vegas'], ['Seattle', 'Lihue'], ['Seattle', 'Beijing'], ['Seattle', 'Denver'], ['Seattle', 'Charlotte'], ['Seattle', 'New York'], ['Seattle', 'Bellingham'], ['Seattle', 'Pasco'], ['Seattle', 'London'], ['Seattle', 'Paris'], ['Seattle', 'Dubai'], ['Seattle', 'San Diego'], ['Seattle', 'Frankfurt'], ['Seattle', 'Edmonton'], ['Seattle', 'Baltimore'], ['Seattle', 'Ontario'], ['Seattle', 'Burbank'], ['Seattle', 'Seoul'], ['Seattle', 'Tokyo'], ['Seattle', 'Bozeman'], ['Seattle', 'Phoenix'] …. rest omitted.

This data is in fact accurate.

Execution time: 0.015566 seconds

Overall, this algorithm is quick and efficient, as its execution time averages roughly 0.0708 seconds per query.

2nd Custom Algorithm:



```python
def custom_algo_2_shortest_path(spark, matrix_variable):
    # Initialize the shortest paths matrix by using the weights of the edges in the neo4j graph that we have
    # It should be setup so that if there is no edge between two airports the value in the matrix is NULL
    shortest_paths = matrix_variable.copy()

    # For each pair of airports (i, j)
    # This tries all three sections of the airport route process
    for k in range(shortest_paths.shape[0]):
        for i in range(shortest_paths.shape[0]):
            for j in range(shortest_paths.shape[0]):
                # Update the shortest travel time
                shortest_paths[i][j] = min(shortest_paths[i][j], shortest_paths[i][k] + shortest_paths[k][j])

    return shortest_paths


"""
I had to add this new realtion type to get this custom algorithm to work effectivley the arrives_at and departs_from wasnt enough I thought.


MATCH (s:Airport), (d:Airport)
WHERE s <> d
CREATE (s)-[:DISTANCE {distance: 1}]->(d);

"""


# Use the Neo4j driver to execute a Cypher query so I can get data from my local db
# This query should get me data that is the distance between each and every pair of airports
with driver.session() as session:
    result = session.run("""
        MATCH (origin:Airport)-[r:DISTANCE]->(destination:Airport)
        WHERE origin.id < destination.id
        RETURN origin.id AS origin_id, destination.id AS destination_id, r.distance AS distance
    """)

# Convert the result to a DataFrame as shown in lecture becuase im using python
matrix_variable = spark.createDataFrame(result.data(), ["origin_id", "destination_id", "distance"])

# Convert the DataFrame to a NumPy array
matrix_variable = matrix_variable.toPandas().values

# Run the algorithm
t0 = time()
shortest_paths = custom_algo_2_shortest_path(spark, matrix_variable)
t1 = time()
print('Algorithm takes %f seconds' %(t1-t0))
```

Input: The weighted graph

Output:

```
>>>Above is a test for neo connection. Below is the results
>>>Algorithm takes 14.874377 seconds
>>>Shortest distance from 34541 to 67890 is 8
```

Execution time: 14.874377 seconds

b. Describe whether you are storing the results of the algorithms in a database and/or data files. How do you plan to present the results to the user? Perhaps you plan to run the algorithm on-demand and present it to the user, or you plan to execute the algorithms off-line, store their results, and present the result on-demand.

We have thought that the best way to go about storing the results of our algorithms is to store them based on the amount of resources needed to execute them if they were needed again and complexity of the algo itself. We will not store the results of the built-in algorithms because they are not complex and are very easy to run multiple times. Align with that we will not store BFS algo as well because it takes in a variable parameter so to get accurate results and save space in our storage system it would be easier to have the user wait. This algo will be run-on-demand. The 2nd algo which is the shortest_paths is the only one of the four algo's that we will consider storing the results in a database. This is because it takes a lot of time to run and the results are more generalized to a large section of the graph. This makes the results dependable and repeatable so if a user were to call it twice there is no need to recalculate it all over again.

3. User Interface and Data Visualization:
    a. Describe your plan for the user interface and data visualization. Note: the final user interface should be interactive - take inputs from a user and provide outputs using algorithm results. The user interface should include data visualization, if applicable.

        We plan on using a command-line interface (CLI) as our search engine's user-interface. The CLI will offer a straightforward menu-driven system where users can interact with the application by selecting from a list of options. The menu will display clear and concise instructions and syntax for conducting searches. Functionalities will include airport/airline search, airline aggregation, and trip recommendations as outlined in the course projects. We will likely use some form of data processing, such as MapReduce, SQL queries, or (maybe) graph algorithms using MatlibPlot or GraphX in Spark.

4. Algorithm Source Code:
    a. Provide source code of your algorithms in a Zip file.

        Zip file was attached to this document when submitted.