CS225 Final Project Written Report

<u>The output and correctness of each algorithm:</u>

1) Traversal

For our traversal, we decided to use a BFS on our adjacency matrix to help achieve our project goals. This algorithm can be found in the graph.cpp file in the 'src' folder. To quickly refresh, BFS stands for breadth-first search and it is a traversal algorithm used to traverse through the nodes of a graph. In this traversal method, we begin our search at the root node and then traverse into the neighboring nodes on the same level before progressing onto the next. In this way, BFS is the opposite of DFS which traverses nodes in a depth-first manner. The reason why we chose to go with BFS instead of DFS is because BFS enables us to find the shortest path between two nodes which fits our project goals better. Our BFS algorithm primarily serves as a function we can use to verify if there is a path that exists between two airports. The function takes in two parameters, a source airport and a destination airport, and returns true if a route exists between the two and false otherwise. Since we are using an adjacency matrix, running this BFS traversal would result in a runtime of O(N^2) where N is the number of nodes in our graph.

To test this function we have a few test cases designed to assert whether or not there is a path that exists between two airports we hand-selected from the datasets. We hand-selected two pairs of airports in which a path exists between and one pair in which there is no path between. This way we could ensure that our function passes both cases.

2) Covered Method

For our covered method, our group decided to use dijkstra's algorithm in order to find the shortest path of routes between two given airports. This algorithm can also be found in the graph.cpp file in the 'src' folder. This algorithm starts at the starting node and then visits the neighboring nodes. While visiting the neighbors, dijsktra's algorithm calculates the distance to the neighbor by adding the weight of the edge between the starting node and the neighbor to the starting node's current distance. If this new distance is less than the current distance of the neighbor, it updates the neighbor's distance and adds the neighbor to a queue of nodes to be visited. The algorithm keeps visiting nodes and updating distances in this fashion until it reaches the destination node, at which time it has discovered the shortest path to get there. Overall, this algorithm has a worst case runtime of O(N^2) on our adjacency matrix where N is the number of nodes in our graph.

To test this function we have a couple of test cases designed to assert whether or not our algorithm correctly finds the shortest path between two airports. We check to see if dijkstra's algorithm runs properly by observing the shortest distance value for the shortest path it calculates for the given inputs. Additionally, we test to see if dijkstra's algorithm correctly identifies when there is no path between the given inputs.

3) Uncovered Method

For our uncovered method, our group implemented an eulerian path identification algorithm to check for eulerian paths within our graph. A eulerian path can be defined as a path in a graph in which each node is visited exactly once. If our algorithm detects an eulerian path in our graph it outputs the path in the form of a vector populated with the eulerian path. If a path is not detected, the function will simply output an empty vector. The overall runtime for this algorithm is O(E) where E is the number of edges in the graph.

To test this function, we have two test cases designed to assert whether or not a eulerian path can be found in our graph. For one of our test cases, we had to hand-make a new dataset in which an eulerian path exists in order to make sure that our algorithm can detect an eulerian path. For the other, we used a dataset in which we were sure no eulerian path exists to make sure that our algorithm can detect when such a path does not exist as well.

The answer to the leading question:

To refresh, our leading question for this project was to find the shortest path between two given airports. To achieve this we utilized two datasets: one that contained a list of source and destination airports as well as their corresponding latitude and longitude coordinates and the other contained route information between airports. Using this data, we initialized a graph in the form of an adjacency matrix in which nodes were airports, edges were connecting routes, and the weight of the edges was the distance between the two airports. From here, we developed algorithms to help answer our leading question as well as go beyond it.

With our BFS traversal algorithm we were able to verify if a path exists between two given airports. Using this information, we were able to go further and implement dijkstra's algorithm to find the shortest path between the two airports. This algorithm was the most important for our project as it directly answers our leading question. With this algorithm, we were able to find the shortest path between any two given airports. This information would be useful for anyone that is curious to know the path they can take to get from one airport to another while achieving minimum distance traveled. After testing and examining numerous different outputs, we found that the shortest path between two airports is not the quickest path nor the one that seems the most logical. We often struggled to understand the output of our algorithm because we would end up confusing shortest path with quickest path. For example, when we examined SFO to ORD, dijkstra's algorithm produced a path of (SFO -> OKC -> ORD). Logically, this may not seem right for the shortest path since that would just be a direct trip from SFO -> ORD. However, when we examined closer, we found that the distance between SFO to

ORD is 1 km shorter when traveling through OKC. This raised the question of how useful this algorithm would be, since a traveling individual would value time more than distance covered. Therefore, we believe this algorithm would serve more useful to those of the commercial airline industry as they can examine the shortest path to take from point A to B so that they can minimize the amount of energy and money being spent per flight.

After we answered our leading question, we wanted to go a little bit further and create an algorithm that could detect eulerian paths within our graph. This algorithm produces an output of the eulerian path if found. If there is no eulerian path found this function will simply output an empty vector. This algorithm's use is to see if the graph can be traversed by visiting each edge only once. In terms of our project, this means it can be used to find different airports to visit when planning a trip. Tourists may find this algorithm helpful as it provides them with the opportunity to avoid visiting the same airport for stopovers/layovers on a given trip.