Jaywoo Jo
Holly Lee
Hemank Kohli

CS225 Final Project Results Report

The following final report describes the outcome of our project, our development , and our discoveries made.

I. Goals

We used the OpenFlights (https://openflights.org/data.html) data set for our project, where we read the data and used it to create a graph to run algorithms of interest. We decided to use the airports.dat and routes.dat files listed on the OpenFlights website. The airports.dat file contains a list of over 10,000 airports with data items such as the airport name and country, the 3-digit IATA code, and longitude and latitude values. The routes.dat file contains flight paths from one airport to another (for example, ICN to SIN).

Our first goal was to find a way to parse the data we need from the files, and our second goal was then to construct the actual graph. We read the data files to create a graph where vertices are the airports (the string of 3-digit IATA codes), directed edges represent flight paths, and edge weights represent the distance in kilometers. In order to calculate this weight, we used the Haversine formula that calculates distance on Earth using longitude/latitude values. Additionally, we used the graph implementation provided to us during CS225 labs to save us time.

Our last goal was to then run our algorithms of interest. It needed to be one of the traversals covered in class and two other options from the complex/uncovered options we were provided with. Firstly, we ran a BFS traversal and outputted the visited vertices; secondly, we found the shortest path using Dijkstra's Algorithm; and thirdly, we found strongly connected components.

Along the way we heavily tested our code, and later created a large number of test cases to test our functions and algorithms.

II. Development

a. Parsing Data and Building Graph

All of our main functions and algorithms that we wrote went in airports.cpp and airports.h, and within those files was the `Flight` class. The constructor for `Flight` takes two string values to be parsed, one for the name of the airport file, and the other for the name of the

flight route file. We have the airports.txt and routes.txt files saved in our project_files, which represent the long data files from OpenFlights. In addition, we also created a routes_test.txt and airports_test.txt which contains just a few lines pulled from the original dataset to simply our testing. When testing our code, we would often create a `Flight` object using these test.txt files for simplicity purposes.

In order for our graph weights to represent distances, we first needed to parse through the airports.dat file. An example of a single line in airports.dat can be seen below:

```
3930,"Incheon International Airport","Seoul","South
Korea","ICN","RKSI",37.46910095214844,126.45099639892578,23,9,"U","Asia/
Seoul","airport","OurAirports"
```

Throughout the project we had to make multiple design decisions. Instead of using the full names of airports, we decided to use the 3-digit codes as vertices (such as "ICN" shown above). As a result of this design choice, all the data besides the 3-digit codes and the longitude and latitude values were irrelevant to us. We parsed these airports.dat lines in our `readAirports()` function, where we return a `map<Vertex, pair<double, double>>`. The string represents the airport name, such as "ICN", and the pair represents the longitude and latitude, such as "37.4691, 125.451". Since we don't need to be extremely precise for this project, we used doubles as it stored enough significant figures for us to perform our calculations. As we will elaborate more on later, 37.45910095214844 vs. 37.4691 wasn't going to make for a significant difference in our calculations.

In parsing line by line using ifstream, we kept track of the comma indices and then used substring in order to extract the data we needed. For example, the longitude point is always between the 6th and 7th comma, so we run a loop and extract the substring between those indices, convert the string of numbers into a double using a helper function, and then add it to the map. Essentially this function iterates through the entire list of airports and successfully maps the airports to its corresponding coordinates. It's very important to note that while all major airports have a 3-digit IATA code, some small airports don't have this 3-digit code, so we decided to not store these values since the routes.dat files don't even show routes that go through these small airports. Essentially airports that don't have 3-digit IATA codes aren't well known airports for the general population.

We then created a `readFlights()` function, which did a lot of the heavy lifting. It first parsed through the routes.dat file. An example of a line from routes.dat can be seen below:

```
AI,218,SIN,3316,ICN,3930,Y,0,77W
```

The only important information here is the "SIN" and "ICN", which indicates that a flight path from SIN to ICN exists. To reiterate, this is a direct path from SIN to ICN. For each flight path in the data file, our function calls `readAirports()` to get information on the airport's coordinates, uses the coordinates to then call a helper function called `distanceHaversine`

which returns the distance between the two coordinates. The `distanceHaversine` function infers that the radius on Earth is fixed. In reality, the radius at different points on Earth is different, so the Haversine formula is merely an approximation. But through testing, we could see that it was very accurate and mostly off by less than 10KM as when we looked up the distance through an online miles calculator. The distance is thus rounded to an int because the decimals aren't necessary. `readFlights()` then uses this data to construct the graph and assign corresponding edge weights. Additionally, mainly for testing purposes, `readFlights()` also returns a `map<pair<Vertex, Vertex>, int>`, which represents the two airports in the path as a pair mapped to the distance between them. For example, looking up the key of `{"SIN","ICN"}` would be mapped to `"4627"`, which in practical terms means that the path distance from SIN to ICN is calculated to be 4627 kilometers.

b. Algorithms

The BFS function is simple; it performs a BFS traversal on a graph. It is important to note that as seen in our BFS test cases, we made the `Graph g` instance variable of our `Flight` class a public variable so that it can be modified. Our code is written so that the constructor takes in the names of the data files and saves it as instance variables so that the graph can be built without parameters in `readFlights()` and `readAirports()`. In making the Graph public, we could test our algorithms on more simple graphs, like with string integer values instead of airport names, to ensure that it functions properly.

The shortestPath function uses Dijkstra's algorithm to find the shortest distance path between the airports given in the graph. The function takes two arguments, Vertex src and Vertex dest which correspond to an origin and a destination airport, respectively. A queue of Vertex is used to traverse the graph. Map<Vertex, double> distance is used to keep track of the minimum distance from the source to the current node. Map<Vertex, string> pred records the predecessors of each node, and later can be used to get the entire routes of the travel. Updating the "dist" map of each vertex is the key idea of Dijkstra's algorithm. First, we set the "dist" of all vertices to infinity except the source one—it should be 0.0. When traversing we check the adjacent vertices of the current node and update their "dist" values to the minimum. It is important to store the distance from the current node to the adjacent node as well so that we can keep track of the shortest distance from the source to each node in the graph. The variable "distFromCurr" is used for this reason. The "pred" map is also updated during the traversal. When there are no vertices left in the queue, we stop the traversal. This way, when two airport codes are passed into the function it returns the shortest distance between those two and also prints the entire route with the shortest distance.

The strongly connected algorithm finds strongly connected components within a given graph. A component is strongly connected if every vertex is reachable from any other vertex within that component. To find strongly connected components, we used Kosaraju's algorithm, which is a two pass algorithm. The first pass uses DFS and a stack to order vertices in the graph

by their finish time. The second pass uses the reversed graph, (where each edge points the opposite way to the original graph), and the stack to compute sets of strongly connected components within the original graph. We were able to test this algorithm on simplified data sets, and later tested it on a subset of our OpenFlights data set to confirm that it works as intended.

c. Testing

As seen in our test cases, we created thorough test cases that test important functions that can be run all at once (`./test`), individually (such as `./test test_integerEasy_BFS`), or in groups (such as `./test [BFS]`). When running our test cases, we also designed it to output the expected vs. actual results. The `readAirports()` test cases test that our functions are able to parse the correct coordinates for airports. The `readFlights()` test cases tested that the flight path between two airports outputs the correct distance. We didn't have to specifically check that we correctly parse the datafiles in our test cases into maps because if the datafile of flight paths wasn't correctly parsed, the path wouldn't have been correctly added to the function, and there's no way the distance function could have accessed the map data in order to calculate the correct distance. Additionally, for the test cases of the `readAirports()` and `readFlights()`, we check that the actual output is within a certain margin of our expected answer. We use margins for the distance because the Haversine distance formula is only an approximation that assumes a fixed radius on Earth, and we also use margins for the coordinates because we rounded the longitude/latitude in the data files to a double when storing it in a map in `readAirports()`.

For our algorithms, there are a few test cases for BFS that test the traversal on a graph of integers (integer values as strings because our graph only holds a Vertex in the form of a string), and a few that test it on a graph of flight paths.

For Dijkstra's algorithm, we have tested with the simple integers as strings and the actual 3-letter airport codes. The first two cases are to check if the algorithm works on the basic graph that it chooses the shortest path among many routes. The last one tests if the shortestPath function gives the correct output with a much larger dataset (i.e., the route_text.txt and airports_text.txt). We get the expected output by calculating the distance with the longitude and latitude values of the airports and compare it to the actual result of the shortestPath function.

For the strongly connected algorithm, we used two simple data sets and a subset of the OpenFlights data set to test against the written code. The first test case is a simple data set designed to test the basic functionality of the strongly connected algorithm. The second test case is a slightly more complicated version of the simple data set to confirm that the algorithm works for more complex cases. Finally, the subset of the OpenFlight data set is used, to test for real world usage.

III. Final Deliverables and Concluding Thoughts

Overall this project was very interesting, but also extremely time consuming. And it didn't help that we were a 3 person group instead of 4. It took nearly an entire week of full attention to finish the project. Despite it being finals week, we had to find time to work together on this project. Because we were not working on this project together in person, we had to find a way to effectively communicate and hold each other accountable. Through a discord server and FB messenger chat that we created, we had actual calls on a near daily basis, and when we were not calling, we were making sure to always communicate through chat. We would chat about what we had accomplished so far and what we intended to do on a specific day.

Most of us were not familiar with collaborative Git workflow, so learning how to branch, stash, and merge were very important lessons. It definitely was a more challenging experience than an MP where we just fill in functions with classes already premade. For this project, we had to make strong design decisions and continue to revise aspects of our code as we saw fit. Being able to apply our flight datasets to run algorithms such as finding the shortest path that have practical applications was extremely interesting. Additionally, being able to apply the data structures that we learned in class was captivating. Overall, we thought we were able to meet all the evaluative components whilst creating a well functioning product that has real life applications.