dzlee2-jt20-jguerr43-kirilk2

RESULTS

The goal of this project was to parse the data from Openflights and use Dijkstra's algorithm with the data to find the shortest paths between 2 airports. We also planned to show which airports along that path would be the most important airports by using DFS with Tarjan's algorithm to see which airports have the most routes as edges.

We do our data parsing in readFromFile.cpp where we use fstream to turn the strings in files into various vectors of strings. By taking in the name of a file as input, we are able to make each line of the file into a string to then be placed into a vector of strings. Following that, we then use a method using stringstream that takes in an element from the vector of strings that contains the lines of the file, comma separate the words in the line, and puts the words derived from that individually into another vector, effectively making a vector of vectors for each line in the file. This is how we were able to derive ID, IATA, and Latitude and Longitude for every single airport.

Stored in Graph.cpp, our implementation of our graph structure used a 2-D vector adjacency list where each APNode vertex represented an airport that stored the ID, IATA, Longitude, and Latitude as attributes. For our edges, we parsed through the routes datafile to find all the valid paths between airports and used that data to search between the IDs of the vertices and create edges. In each edge we store the next vertex's ID and calculate the distance between the starting vertex to the destination vertex if it is found in the route's data. The edges use distance as their weight. There were a lot of invalid data entries in the datafile that were missing IDs, IATAs, or had extra commas in the strings. We were able to circumvent this by using ASCII to distinguish digits from other characters and parse only the characters that were of interest. In order to handle invalid nodes, we made exceptions and print error messages to let the user know that those IDs did not exist.

In DFS.cpp, we were able to implement a full traversal of the graph. By running the project with “Start Airport ID” as the starting vertex and “End Airport ID” as the ending vertex, you will then be able to see the maximum number of visits that can be made. Otherwise, the user can also input a chosen number and you can see the traversal of the graph until that number of “jumps” has been made. We wrote two implementations, traverse() and path_finder(). traverse() traverses the whole graph ending at the “End Airport ID” inputted by the user at the start whereas path_finder() just traverses until it reaches the inputted destination, which doesn’t necessarily visit every node or use every edge in that case.

Following the pseudo code provided in lecture, the covered algorithm we chose is Dijkstra’s which is located in dijkstra.cpp. This algorithm uses the same input by the user given at the start and will return the shortest path between the two airports as denoted by their ID’s (Figure 2). This was very hard to test as there are many direct paths between airports and therefore we needed to create a special test case to show that our algorithm is working as shown in Figure 3. The shortest path is highlighted in green which can be proven by the sum of the weights and comparing it all with the sum of the rest of the paths.

For our uncovered algorithm, we selected Tarjan’s algorithm for strongly connected components. We were not able to implement it as it required a custom DFS to traverse the graph, and we did not have enough time to adapt DFS any further. If it had worked on this large dataset, the results would have been slightly harder to show through the output, so we planned on using a special test case which we had already written using the same dataset shown in Figure 3. In one set, it would have vertices: [4, 5] and in the second set vertices: [2, 3, 6] and so on.

Overall, we were able to make our algorithms work, and we found that in the modern world, airports are so well connected that it is hard to find cases where multiple edges are needed to travel from one airport to another. Hopefully in future we can continue our research using Tarjan’s and learn more about the world’s most vital connections.

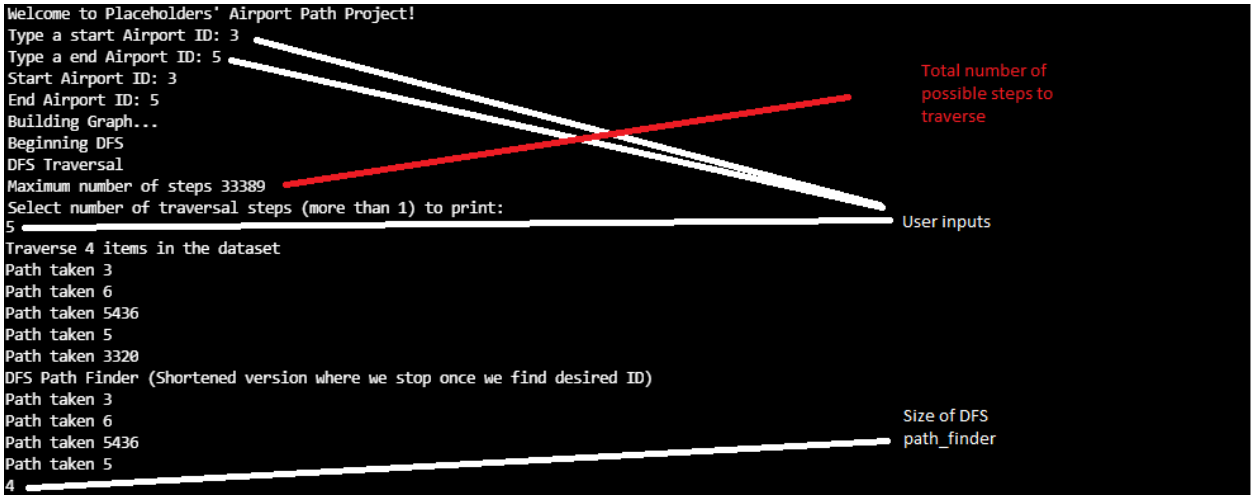

Figure 1: DFS traverse and Path Finder (User input Start Airport ID:3 and End Airport ID:5)

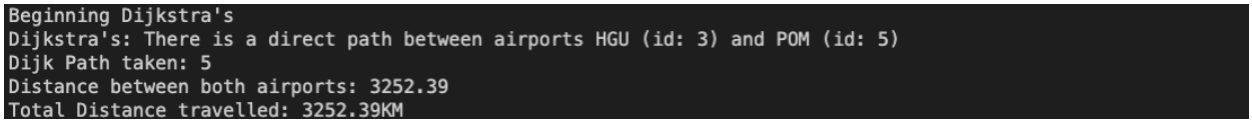

Figure 2: Output of console given the User input Start Airport ID:3 and End Airport ID:5

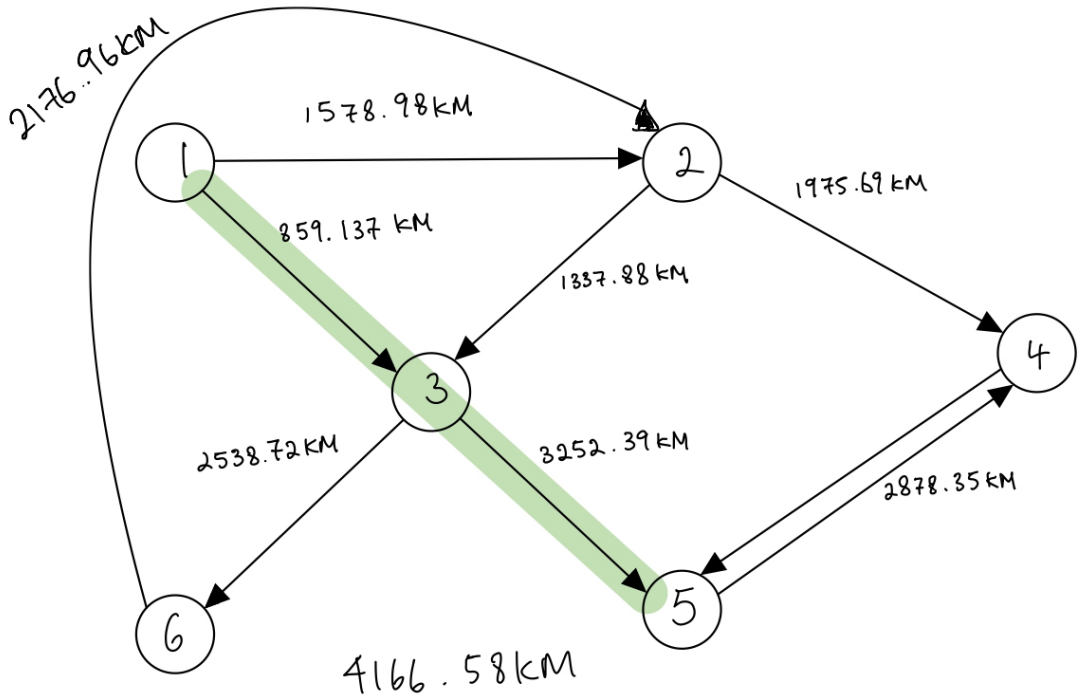

Figure 3: Dijkstra's Path Highlighted in Green