Leading Question

In our project, we investigate the real-world applications of graph-data structures using the Open Flights dataset in conjunction with a path finding algorithm and a traversal. We intend to use the dataset provided to us in class (https://openflights.org/data.html). Specifically, we hope to find the shortest path between the United States and Europe, as well as the shortest path between different landmarks. We will use the nearest airport to each corresponding location and construct a route based on the shortest path using DFS traversal, Dijkstra's algorithm, and a Page Rank algorithm.

Dataset Acquisition

Data Format

We will be using the airport and routes databases from OpenFlights to perform the analysis (airports.dat and routes.dat respectively). Although these files contain massive amounts of data - over 10,000 airports and 67663 routes - the CSV formatting will make them easier to parse. Each entry in the route database contains the source airport ID and destination airport ID, which we can look up in the airport database to determine their latitude and longitude degrees to calculate the distance between the airports. The distances will be used in the Djikstra's algorithm, while the IDs will be used when constructing the adjacency matrix for PageRank. We plan to use as the entire dataset.

Data Correction

Because each individual entry of input is comma-delimited and on a single line, we can parse the file by extracting each line and using the overloaded 3-arg getline() method from the <string> C++ library to retrieve each data field between the delimiters (commas, in this case) for that particular line. The OpenFlights website notes that the value \N is used for "NULL" to indicate missing values, which we can account for when parsing the data. If any of our target fields (source ID, destination ID for now) are missing, the entry will most likely not be used in our analysis. A potential naive solution to recover the IDs, however, could be to use other information provided in the entry, such as the airport's IATA or ICAO code.

Data Storage

We will create a simple Location class to store the latitude and longitude degrees of each airport, as well as a method to calculate distance given another airport's Location. With this, we can utilize C++ maps to store every airport's unique identifier and their Location as key-value pairs. We are using maps (as opposed to unordered maps) because the efficiencies of lookup and insertion are guaranteed O(log N). While we don't need the key-value pairs to be sorted, unordered maps consume extra memory for internal hashing, which we want to avoid because of how much data we are planning to store. We will also use a map to store UID-size_t pairs to aid

in populating the adjacency matrix, which will be composed of vectors. They are dynamically resizable while still having an array-like structure. Retrieval is O(1) and insertion is O(n).

Algorithm

We will use the DFS traversal algorithm to traverse through our graph structure. The nodes in the graph will represent the airports and the edges will have a weight (distance) attached to it. Djikstra's algorithm will be used to find the distance between the starting point and final destination. The final output for our project will be a vector of vertices which represent the shortest path. The plan for now is to use AVL Trees for our data structure. Djikstras: This algorithm finds the shortest distance between two airports which are represented as nodes. The input for this algorithm are two nodes which represent the starting point and the destination, and it will return a vector of all the airports one would see on the way. The time complexity of the algorithm is O(nlogn), where n represents the number of nodes. Page Rank: This algorithm will be helpful when finding the most important airports around the world and ranking them. The input of this algorithm is a node from the direct graph where we are storing the information of all airports and the range of all the airports which we would want in our adjacency matrix. The output will be an adjacency matrix of all the popular airports. The runtime of the algorithm is O(n*m), where n is the number of nodes and m is the number of edges. DFS: This is the algorithm we will use to traverse through our graph. The input of DFS will be a graph and a starting vertex. The output will be a vector containing all the nodes/airports in the graph. The runtime of this algorithm is O(n+m) where n is the number of nodes and m is the number of vertices.

Timeline

Since we have 5 weeks to complete this project, we will split up the work that we do accordingly. For Week 1, we will complete the proposal, contract, set up the git repository, and organize a template for our algorithms. In Week 2, we will complete Dijkstra's algorithms and set up tests for it. In week 3, we will complete the Page Rank algorithm and complete the tests for it as well. During week 4, we will implement the DFS in our project. For week 5, we will finish any loose ends and complete all the tests for our algorithms making sure everything is working. We will also create our final presentation and write-up of our project.