Project Proposal

Leading Question

We want to create a program that allows people to find the shortest route between two airports all around the world.

After receiving the departure airport and the arrival airport, the overall program will output the shortest, most efficient route between the two locations. By doing this, the user will be able to find the most optimal route for the trip. Firstly, we want to see which airports are closest to a certain airport based on the routes. We will achieve this by performing a BFS traversal that takes in a parameter n, and returns a list of airports which are n flight routes from the source airport. For the shortest possible route, we will achieve this by implementing Dijkstra's Algorithm which finds the shortest path in the directed weighted graph between a source and destination. The nodes of our graph will be the airports, the edges are the flight routes and weights are the distances of each respective route. Our third goal is to determine the overall strongly connected components of our graph and output this into a vector of vectors, where each inner vector represents each set of strongly connected components. We will achieve this by implementing Kosaraju's algorithm. This information will be useful to tell which set of airports can cycle between each other.

If time allows, we will try to visualize the output of Kosaraju's algorithm onto a global map png file, where airports will be color-coded based on which strongly connected component they belong to. For example, if O'hare and Dallas belonged to the same strongly connected component and the same for Atlanta and JFK, then ORD and Dallas would be represented as blue while ATX and JFK would be red.

Data Acquisition and Processing

We will be using: airports.dat and routes.dat

Data Format

Our input dataset consists of two files obtained from the OpenFlights Dataset, which contains information on flight routes, airports, and airlines from 2014. The first file "airports.dat" contains information on airports (nodes), its longitude, and its latitude. The

second file “routes.dat” contains information on each directed flight from one airport to another, which act as edges. Each line of the first file contains Airport ID, name, city, country, longitude, IATA, ICAO, latitude, altitude, timezone. Each line of the second file contains the airline, airline ID, source airport, source airport ID, Destination Airport.

The first file is around 1.08 MB and the second file is around 2 MB. We will be using most of these data sets to find the shortest route distance between airports.

Data Correction

When building our graph, we’ll implement various checks that fix any existing errors. The first check will confirm that the latitudes and longitudes make sense. If any of the numbers are outside of their possible range, we’ll increase or decrease the number to the closest number that’s possible. Any edges with start and/or end node IDs that don’t exist will be ignored. We will implement another check to ensure that in every route between two airports each airport’s ID is in our airport dataset.

Data Storage

We’ll represent our graph using an adjacency list. More specifically, each airport will be in a node in our graph corresponding to an AirportNode object. Our adjacency list will be based on a map implementation where the key is the AirportNode and the value is a list of outgoing flights. Each edge in the list will correspond to a pair, where the first entry in the pair is the AirportNode of the destination and the second entry is the computed distance (based on longitude and latitude difference between the two airports). For convenience AirportNode objects will be identified by their three letter IATA code because the routes dataset has each entry based on those codes. We will also create a map from airport name to airport code to easily retrieve the IATA code of any airport in our dataset.

Graph Algorithms

Function Inputs

Our project will also use a BFS traversal that takes in an input n which represents a number of flights from a specific airport. The purpose of this traversal is to find every airport that is n flights away from the inputted airport.

Our project will utilize Dijkstra’s Algorithm to calculate the shortest flight-route distance between two airports based on longitude and latitude points. Since our dataset already

separates information about the nodes and edges into two files, we can use these files as inputs to build our graph into an adjacency list. The “Airports.dat” dataset contains information on the airports themselves, their codes and locations in terms of longitude and altitude. When comparing distance, we will utilize the distance function to determine the distance between two Airports.

Our third algorithm–Kosaraju’s for determining strongly connected components–will be taken in an adjacency list representation of the graph and a source airport (this airport will be a specific one chosen for commencing the traversal) . The purpose will be to perform the algorithm and determine which strongly connected components exist

If time allows, for creating a color-coded visualization, we hope to achieve this using the cs225 folder including the HSLAPixel and PNG classes for modifying pixels on a source image. Essentially, we would find a way to plot each airport based on its longitude and latitude and color it accordingly.

Function Outputs

Our first set of functions would be designed to construct the adjacency list of the weighted directed graph.

The expected output for the BFS algorithm would be a vector of every airport that is n flights away from the source airport.

The expected output for Dijkstra’s Algorithm would be the shortest flight distance between two airports using the flight routes provided. We create another form of Dijkstra’s algorithm to actually return a list of the flights needed to get the shortest route (the actual nodes in the path itself).

The expected output for Kosaraju’s algorithm is a 2D vector of airportNodes where each row represents a set of strongly connected components. More specifically, the return type would be vector<vector<AirportNode>>, such that each inner vector represents the strongly connected components.

Function Efficiency

The Big O efficiency for the BFS algorithm implementation is ideally

$O(|V|+|E|)$ but the time complexity largely depends on n

Where E is the number of edges and V is the number of vertices in our weighted directed graph.

The target Big O efficiency for our Dijksta's Algorithm implementation is ideally

$$O(|E|+ |V|\log(|V|))$$

Where E is the number of Edges in our graph (the number of lines in the dataset and V is the number of vertices (Airports) in our data set.

The target Big O efficiency for our implementation of Kosaraju's Algorithm is ideally:

$$O(|E|+ |V|)$$

Where E is the number of edges and V is the number of vertices in our weighted directed graph.

Timeline

Week 1 (October 31st)

1. Creating project proposal (Github link, Determining project and components)
2. Determine basic implementation of project
3. Ensuring GitHub collaboration works properly

Week 2 (November 7th)

1. Rewrite proposal and select necessary algorithms
2. Working on storing data from the datasets into our project
3. Implement basic BFS Algorithm

Week 3 (November 14th)

1. Work on reading from files and turning data into an adjacency list representation of the graph
2. Finalize BFS algorithm that returns airports that are n flights away from source airport
3. Write test cases to ensure that our graph representation is correct/ideal
4. Begin designing and implementing Dijkstra's Algorithm that returns the distance

Week 4 (November 21st)

1. Begin designing and implementing Dijkstra's Algorithm that returns the list of airports in the path itself

2. Design a method to determine the distance between two Airports
3. Write test cases for Dijkstra's Algorithm
4. Begin implementation of Kosaraju's algorithm to find strongly connected components

Week 5 (November 28th)

1. Finish any lingering parts on BFS algorithm and Dijkstra's Algorithm
2. Finish implementation of Kosaraju's algorithm
3. Write tests cases for Kosaraju's algorithm
4. If time permits, implement the program for outputting a color-coded image of strongly connected components

Week 6 (December 5th) (Due December 8th)

1. Code and documentation clean-up
2. Presentation and Report
3. Github Repo Distribution