Project Proposal
Rohith, Michael, Calvin, Justin

1: Leading Question

How can we determine the most efficient way to travel from one airport to another within the U.S.?
"Efficient" can refer to the shortest trip in terms of distance or flying time, least number of stop-overs, or lowest price for a flight. Thus, using a dataset of almost every flight route and airport in the U.S., we can construct a directed graph (with airports being the nodes, and flight paths being the edges), and determine the shortest length from shortest-path algorithms (and shortest time with some variable manipulations/calculations), the lowest price by adding weights to the edges, and least stopovers by computing walk lengths. Ultimately, a user would choose a custom option(s) (least length/time/stops/price), with the application returning the most efficient flight paths.

2: Dataset Acquisition and Processing

- Data Format
  The dataset we plan to utilize is the OpenFlights database, which stores information identifying thousands of airports worldwide (including their 3-letter codes, city location, and latitude/longitude location)--representing the graph's nodes, along with a separate file containing routes (airline names, source/destination airports, and number of stopovers)--representing the edges. These two files are both in a ".dat" format, with each line containing comma-separated values between the airline, source, destination, and stop count. The Airports file is 1MB (contains 14,000 rows), and the Routes file is 2MB (67,663 rows). While the storage size is minimal, the large number of rows in each file may make our app too complicated to debug, so we will use a subset of this data (containing only airports and routes that stay within the U.S.).
- Data Correction
  We will write a function to read the Routes CSV file and will store each line in a vector (with commas indicating the next index). We will check the latitude/longitude values to ensure each airport/route stays within the U.S. before creating a vector for that line, and the end result is a 2D vector (with each row corresponding to a line from the Routes data file). We will create a similar 2D vector for airport information.
  To ensure these data structures are error-free, each line vector should have a usable/unusable indicator element (the last index), so that the program will only include "usable" vectors in the result. If the CSV reader detects nonexistent values (or just a space) between commas, we will instead insert null values for such indexes in the vector, and mark the vector as unusable. If the other file contains conflicting information (e.g. Routes vector contains a null element, but Airports vector contains a usable string for the same information), the non-null value will override both vectors, and if no null values exist in a line vector after correction, the vector will be marked as usable and will be included in future output results.
- Data Storage
  We will ultimately store information in the vector into a weighted+directed graph. Each graph node will point to a certain row in the Airports 2D vector, with information accessed directly through that vector. The graph's edges will be stored in an adjacency list (with each element's first index pointing to a graph node, and the element's second index containing a vector of elements pointing to rows in the Routes 2D vector). Particularly, iterating through rows in the Routes vector, if the first two indexes match available graph nodes, an element will be added to the adjacency list, pointing to that row in the Routes vector.

Transferring the CSV contents into 2D vectors will take O(M*N) time (M is the number of rows, or airports/routes, and N is the number of columns, or details per row, in the largest CSV file). Searching for missing information involves iterating through each row, which takes O(M1+M2) time (M1 and M2 are the number of rows in the airports and routes files, respectively). Creating the adjacency list takes O(N) time (N represents a Route, or edge, which is assigned to an Airport, or node).

3) Graph Algorithms

- Data Parser (CSV->Map of vectors of data entries)
  We will need to create a data parser for the CSV file of flight data that will convert it into a map of vectors of data entries. We believe that we will have to use 2 of the datasets on OpenFlights in order to get the names of certain airports associated with airport IATAs, as well as the routes with sources and destinations. This algorithm will take in a CSV file of flight data and should output vectors of data, or specific data entries associated with each route. If using a map, this function can be O(n) time since it only needs to iterate each line once, and add any connections to the adjacency list once. The amount of memory taken up will be O(n).

- Dijkstra's algorithm will take in two nodes and a graph that the nodes belong to and will output the shortest pathween between the two nodes. The shortest path can be determined by weights put on each edge or number of edges crossed through. Our goal for the time complexity of this algorithm is O(ElogV + VlogV) and a space complexity of O(V).

- A* search algorithm is another algorithm that we could use for finding the shortest path between two nodes. The inputs will be a starting node and a desired node along with a graph that contains the nodes. It will output the shortest path between the two nodes. Our goal for the time complexity of this algorithm is O(E) and a space complexity of O(V).

- Breadth first traversal is a general algorithm for traversing a graph. It's generally used to find a vertex. The input will be a desired vertex and it will output the location of the desired vertex within the graph. The goal for the time complexity of this algorithm is O(E) and a space complexity of O(V).

4) Timeline

Date (Assumed due date of weekly development log)	**Goals**
November 4th	Finish Team Contract, Project Proposal, and Github link
November 11th	Configure CMake (make files), Create Codebase (folders and necessary files), Create Class/Header files (include some function and variable declarations, if possible), begin creating graph from CSV file
November 18th (Mid-Project Check in)	Finish creating graph from CSV file, implement Dijkstra's algorithm to find shortest path in graph, and add test cases for this algorithm, and start preparing second algorithm (A* search)

	if possible
December 2nd	Finish A* search algorithm, add test cases for A*, and start implementing BFS Traversal
December 8th (Project due)	Finalize project (input/output), finish written report (results.md) and README.md, prepare final presentation video