CS 225 Final Project Results

Saket Vissapragada, Siddhartha Adatrao, Sruthi Kilari, Shreya Sharma

Fall 2020

1 Introduction

The goal of our project was to detect the shortest flight path from a start point to an end point on a randomized graph of airports. In order to accomplish this, we implemented Dijkstra's Algorithm to find the shortest path between any two points. Furthermore, we also wanted to find the shortest path between any two points when given a point to visit in between. We were able to accomplish this through the Landmark Path Algorithm. We also implemented a BFS traversal of our randomized graph.

2 Discoveries and Outcome

2.1 Preproccessing/Graph Construction

As we were preprocessing the data, we noticed that the initial CSV file from OpenFlights had commas in the airport name which caused the parsing to be difficult. We also noticed that the only data that was relevant data to the goal that we wanted to wanted to accomplish was an airport identifier (name) and the location (latitude and longitude coordinates). When constructing our graph, we noticed that the Haversine formula would be useful for calculating distances on a sphere. However, after some research we also noticed that there is some error associated with calculating the distance since the Earth is not a perfect sphere. Since our project goal did not specifically require displaying the distances, we figured that the error associated would be okay due to the distances being relative to each other.

2.2 BFS

For BFS, we took inspiration from mp_traversals. We knew that it traverses breadth first through an undirected graph using a queue, so we focused on putting the airport data into a graph successfully. From there, we just used our knowledge from class to write a BFS traversal on the graph of airports.

2.3 Dijkstra's Algorithm

When learning about Dijkstra's Algorithm, we found that it finds the shortest distance from the start point to the end point using directed edge weights. As mentioned above, our graph was created to have an only one edge between every node.

Therefore, when this graph is directed, not all nodes will will point to all other nodes. This was done intentionally to actually be able to use Dijkstra's algorithm. If all nodes led to all other nodes, then the shortest path would be directly to those nodes. We implemented Dijkstra's using a priority queue as a minimum heap making a custom compare function to compare tentative distances of paths. While visiting each node, we faced the issue of having a priority queue that could not be edited if a new tentative distance to a given airport was found than a previous path. To solve this issue, we stored the tentative distances in an unordered map. Every time we added a node to the queue we added/updated the tentative distance to the map if it was smaller or if it was the first instance of the node as a neighbor to a visited node. Furthermore, every time we needed to visit a node by popping off an airport off of the queue, we checked if the the value stored in the queue was the same as the up to date distance in the map. If they were different, we knew that another instance of the same node would be in the correct place somewhere in the queue and we would pop the top node off without adding it to visited or checking its neighbors.

2.4 Landmark Path Algorithm

When researching the Landmark Path Algorithm, we noticed that it essentially looks for the shortest path from point A to the landmark to be visited and again from the landmark to point C. Therefore, we decided to use our prior implementation of Dijkstra's Algorithm in our Landmark Path Algorithm. We called the function from our Dijkstra's Algorithm class that found the shortest path between two points in our Landmark Path class twice in order to implement this. We then saved both the paths taken from point A to the landmark and from the landmark to point C into one vector. This allowed us to successfully calculate the shortest path between two points when given a landmark to visit in between.

3 Conclusion

Overall, we learned how to deal with large datasets and modularly (testing with a small portion of our large dataset) and incrementally testing (testing frequently on small segments of code that we right) our code. We also learned how to collaborate on developing software and how to solve issues such as merge conflicts and pair programming.