Team Name - Thomas and Friends
Team Members - Camila Djurinsky Zapolski, Pietro Landell, Ethan Wilson
GitHub URL - https://github.com/camidjur/PROJECT3-GRAPH
Link to Video - https://youtu.be/Et_Bx5VYe1Y

Problem: What problem are we trying to solve?

In this project, we are attempting to figure out what the shortest path is, from one point to another within a city. Specifically, we are using the shortest path to make comparisons regarding time and gas emissions, for different modes of transportation, to provide users full understanding of the result of their choices of transportation.

Motivation: Why is this a problem?

We were motivated to choose this topic because we are all from Miami, a part of Florida that is notorious for its lack of public transportation, and its traffic. Due to this, transportation via car results in an increase in gas emissions, in part increased by the extended time travel takes when hitting peak hours. This, in comparison to transportation in Gainesville, where there is an excess of bus routes that run every hour, where walkability is encouraged within the surroundings of UF, and where most students have bicycles, raised the question of whether knowledge about the results of transportation could potentially influence choice of transportation. If, when given the opportunity, more people chose to use a mode of transportation that did not emit CO2, this could have a great impact on reducing emissions and therefore, the effects of climate change.

Features implemented

In order to raise awareness of this, we decided to compare the distance that is traveled within the shortest path between two vertices, and compare the time it would take, and the gas emissions that would result, depending on which mode of transportation was picked. To do this, we found some average speeds for each transportation, making the assumption that the same speed limit would be implemented within all roads. Thus, in urban areas, the average speed for a car is 30mph, for a bus it is 20mph (due to the number of stops it has to do within its route), for a bike it is 11mph, and walking it is 3mph. The second feature we kept track of was average gas emission per mile traveled. Walking and biking obviously do not emit CO2, but we found that the average passenger car emits 400g of CO2 per mile traveled, and the average bus emits 300g. This was used in later calculations in order to determine the total amount of gas emission within the respective paths.

Description of Data

The data we used came from Overpass Turbo, which is a web-based data filtering tool for OpenStreetMap. OSM is a website where users can fill in information about the streets they drive through. Using several query searches, we downloaded three files with a GeoJSON formatting, corresponding to walking paths, biking paths, and driving or public transport paths. GeoJSON files are used to store geographical data and information about specific paths and points. The data detailed several "LineStrings" which corresponded to roads that had starting coordinates, coordinates in between, and ending coordinates, which were added to our graph as vertices with two way edges. The file was parsed through, the coordinates were made into an instance of a class Coordinates that had a latitude and longitude, and these were saved onto the graph as an ordered map with coordinates as keys and vectors of weights and adjacent coordinates as values. The data and its implementation makes several assumptions which should be noted. For the weights, it uses the euclidean formula for distance meaning it neglects the curves on roads. The data, however, includes so many close together nodes that this should not matter too much. It also assumes every road is two ways and that buses go wherever cars go, for the sake of simplicity. This could be easily changed with better data.

Tools/Languages/APIs/Libraries used

The shortest path algorithms were written in C++, using the map, vector, queue, and limits libraries for their respective implementations. The map and vector libraries were necessary since this was the data structure that we chose to hold the graph, with coordinates as keys and vectors of weights and adjacent coordinates as values. Additionally, the limits library was used within the shortest path algorithms in order to set some entries as a value meant to be overwritten/relaxed within implementation.

Algorithms Implemented

The algorithms that were implemented for this project were Dijkstra's algorithm and Bellman Ford's algorithm for shortest path. These implementations were guided by pseudocode either provided in class or through research on the algorithm. Dijkstra's algorithm is theoretically limited, in the sense that it can only ensure the optimal solution if all the weights are positive. On the other hand, Bellman Ford's algorithm does allow for negative weights. Luckily, for this project, our data set did not include negative weighted values, ensuring that the solutions found were indeed the optimal ones. Furthermore, their respective implementations are different, with Dijkstra's utilizing a priority queue to keep track of the visited vertices, and Bellman Ford's simply iterating through the graph $|V| - 1$ times – V being the number of vertices in the graph – to account for the source vertex's weight being 0.

Additional Data Structures/Algorithms used

There was a single function that was added to the shortest path implementation, and this was a function that was written to print out the statements regarding speed and gas emissions previously mentioned. In this function, we calculated how long it would take to travel this distance, in mph, for the respective mode of transportation chosen, as well as how much faster or slower this mode was in comparison to the rest. The same was done for the gas emissions, multiplying the average emission per mile for that mode of transportation, and comparing it to the respective emissions for that same distance, if a different transportation vehicle was used.

Another group of algorithms used can be found within the main.cpp and gather information in the form of a menu. There are three functions, getTransportationMode, getSearchAlgorithm, and getStartAndEndPoints all with identical structures, the first gathers information regarding the transportation method, the second gathers information regarding which algorithm to use and the final function records the coordinates being used. Each of these algorithms output the menu options to the user and will recursively continue to ask for an input until the user inputs a valid response. These are called within the main function to gather information regarding the user's choices which are then sent to the data.cpp file.

In the data.cpp file, we create a class called Coordinates which is a longitude and a latitude and is used throughout the program to save the vertices. It also has two overloading operators to allow for comparisons. The function calculateDistance uses an estimate for the length of one latitude or longitude change to convert it into miles and uses the Euclidean formula for distance to do so. The last function is the getData() function which downloads the file according to the user input and parses through it to create the graph structure with the vertices and edges necessary.

Distribution of Responsibility and Roles: Who did what?

In order to efficiently work on this project, we divided the roles and algorithms between us. Pietro cleaned and adapted the data in a manner that would fit the map data structure that we decided to utilize for the graph representation. Camila wrote the shortest path algorithms for Dijkstra's and Bellman (which later had to be slightly altered by Pietro to utilize the class for the data points), and Ethan implemented the abstraction of the functions and all of the user - interface aspects of the problem. This included input verification and validation, the choice of which algorithms need to be called at which points, as well as the incorporation of all of our respective parts into a cohesive program.

Any changes the group made after the proposal? The rationale behind the changes.

One of the biggest changes that we made after the proposal was that we did not take into account the prioritization of speed or gas emissions, we simply calculated the shortest path based on distance. The reason we did this is because we came to the realization that prioritizing speed would simply mean dividing all of the weights by the mph speed of the respective choice of transportation that the user chose. This would ultimately result in the same path, just with a different interpretation of the weights. In the same way, prioritizing for gas emissions would simply be multiplying the weights by the average gas emission per mile, depending on each mode of transportation. Additionally, if the user were to choose walking as their mode of transportation, all of the weights would become 0, not allowing us to use the algorithms listed above to find the shortest distance between the two vertices.

However, we did not want to compromise the idea of the project, so we still made the gas emissions and speed comparisons, we simply made them for all modes of transportations, disregarding the prioritization of the user. We thought it was important to provide this information, the manner in which we did so did not matter as much as the ultimate message.

Big O Complexity

The time complexity, in the worst case, for Djikstra's algorithm is O(2V*log(V) + E*log(V)) ~ O(E*log(V)), if we assume that E > V, where V is the number of vertices in the graph and E is the number of edges in the graph. The reason for this is that the outer while loop that iterates as long as the priority queue is not empty will iterate for every single vertex, and then inside of the while loop, we are iterating to every adjacent vertex, a.k.a. checking every outgoing edge for each vertex, and relaxing its weight if necessary. Working with a directed and connected graph, this would therefore be O(E*log(V)).

The time complexity, in the worst case, for Bellman and Ford's algorithm is O(V * E), where V is the number of vertices in the graph and E is the number of edges in the graph. The reason for this is that in order to relax all of the edges in a Bellman and Ford algorithm, we are iterating |V| - 1 times, and then also iterating through every single edge. We also iterate through every edge once again in order to check after all the relaxations were made, if there are any negative-weighted cycles. However, the time complexity for this is O(E), assuming that V*E > E, since V is always a positive number, we conclude that the time complexity for this algorithm is O(V*E).

The time complexity, in the worst case, for the three functions getTransportationMode, getSearchAlgorithm, and getStartAndEndPoints are O(n) where n is the number of times it takes the user to input a valid response. These functions are simple and consist primarily of displaying some form of text as a menu. Once the menu is displayed the user can then input in their choice which is recorded and returned if it's valid, and if not, the function asks for a valid input and recursively calls itself. The time complexity is primarily decided by the user input and because of this nature, the worst case time complexity is O(n). However, in the best case scenario of the user inputting a valid response on the first attempt, the time complexity will be a constant O(1).

The worst case time complexity of taking in the data in the getGraph() function is O(N * log(V)), where N is the number of lines in the data file that is imported and V are the vertices of the graph. The code just goes line by line, reading the file and taking in the necessary data. Creating the coordinate structure happens in constant time but the lookup and adding edges to the graph happens in log V time, since the map structure uses a red-black tree to organize the data. Therefore, looking up for every coordinate causes the two to be multiplied and gives us the big-O time above.

<u>As a group, how was the overall experience for the project?</u>

The overall experience for this project was good. Although we were a little stressed about the proximity of the due date of this project to that of the final exam, we were able to work on our parts independently, while holding weekly check-in meetings that allowed us to keep each other accountable and present in the project. These check-ins were also extremely helpful in keeping us on track and being able to connect our code into one piece.

<u>Did you have any challenges? If so, describe.</u>

The biggest challenge we faced was actually parsing through the data and putting it in a format that could have been used and interpreted by the algorithms and the data itself. This was incredibly frustrating given that the data set came in measures of longitude and latitude for the whole world, and having to figure out how to narrow down the information so that it was more digestible became difficult. Even after parsing through, the algorithm is severely limited by the data we collected, and a lot more data parsing and work would need to be done in order to make the software useful.

<u>If you were to start once again as a group, any changes you would make to the project and/or workflow?</u>

With more time, and perhaps less finals in the way, we would have wished to create a more advanced and user friendly user interface. However, given the time constraints that we faced, we decided that it was better to focus on the most relevant core parts of the project, rather than compromise the quality of our work in order to make an additional facet of the project.

<u>Comment on what each of the members learned through this process.</u>

Camila Djurinsky Zapolski:

I think that I have a much deeper understanding of Dijkstra and Bellman and Ford's algorithms. Although I knew how to find the shortest paths from a source vertex by hand using Dijkstra's method, it was very interesting to see how this would be done with programming. Additionally, despite Bellman and Ford's algorithm conceptually, we never really covered in class how the algorithm worked, making this a new learning experience for me.

Pietro Landell:

Most of my work was dealing with data and learning how to read GeoJSON files. I learned about different places to look for data and several different file formats. The data processing aspect also taught me a lot, especially how to parse through data and extract the exact information you want. I also created a Coordinate class and had to adapt the algorithms we learned in class to work with this specific class, constructing overloading operators for comparison in the class as well. It was a very rewarding experience and I feel that my programming abilities have improved.

Ethan Wilson:

A majority of my time spent on this project was spent organizing the different parts of the program and putting them together into one cohesive tool. I mostly accomplished this by creating the front end menu that the user uses. Throughout this process I gained some insight into how larger projects can be run and organized. Not only this, but I also gained some experience working with APIs and user interfaces when implementing a system to convert user inputted addresses into coordinates to be used on the backend.

References

https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
https://overpass-turbo.eu/index.html
https://bridgesuncc.github.io/tutorials/Data_OSM.html
https://docs.locationiq.com/docs/introduction