ECE 487 Final Project

Ryan Berning

December 2021

1 Introduction

The purpose of my project was to make Miami University into a graph object and find the shortest path between points. The walking paths were represented with edges and the intersections were represented with nodes. The implementation of the project was successful, utilizing an a* style algorithm to find the shortest path between two given points. After the base function was shown to work, additional functions were added. The functions were a "traffic variable" that would increase pathing cost through congested routes and the ability to have multiple destinations in a route, as well as type of travel.

2 Background

When considering a project for this class to display a CAD flow and read through a data structure, a program that created directions to get from place to place quickly came to mind. The "conversion" from buildings and paths to nodes and edges was immediately apparent. A use for reading through the converted graph data object was also apparent, determining routes between buildings.

The directions-finding program was also attractive as complexity could be added later with different features that modified the map data and the way it was interpreted. For example, by adding a variable for congestion, certain routes could become less desirable even though their raw distance to the given destination is the shortest. Giving an option for mode of transportation was also a possible addition that would give the user different time(s) to the destination, depending on their selection.

Figure 1: Map of the university with nodes labeled and paths highlighted

3 Idea Description

The idea behind the pathing algorithm is to go to the given start node and check connected nodes to see which one is the closest to the destination node. Then the closest node and its connected nodes are checked in the same way and the process repeats[Wikipedia, 2021]. Adding multiple destinations would then require shifting the input start node and end node to the previous end node and next given destination node respectively.

The idea for the congestion function is to add a variable to the node data, that increases the cost of pathing through that node. This occurs at certain times of day, so an input for time of travel is necessary. This feature also influences the travel time displayed by the program.

4 Experimental Setup

The setup of the initial algorithm is an a* inspired shortest-path pathing algorithm. The algorithm is first given a start node and end node. Then, it checks all nodes connected to the start node to find the one closest to the destination node. Once the closest node is found, it resets the initial process, this time checking the distance of all nodes connected to what was the closest node before. This process is repeated until the algorithm reaches the destination node, or it has traversed the entire graphs worth of nodes, signalling a failure and termination condition to prevent infinite looping.

The levelization of the graph object is actually built in before conversion to a graph object, by virtue of being a map. Since maps use latitude and longitude to

describe where objects on them are, when converted to a graph object, continuing to use the latitude and longitude makes significantly more sense than trying to re-levelize the graph. Distance is found using trigonometric equations made for finding distances on a sphere called Haversine equations[Jason Winn, 2013]. Thus, finding the closest node simply requires plugging the current closest node's latitude and longitude into a class that provides a convenient Haversine method along with the destination node's latitude and longitude, and repeating this for the node to be compared against. The distances the method returns are then checked against each other to determine closeness to the destination.

To convert the map of the university, node data was input to a Comma Separated Value(CSV) file, which was then easy to read into the program. The nodes representing intersections were given numbers and named in the following format: intersection_25. While all building nodes were named after the respective buildings, Laws Hall for example being titled Laws_Hall. A point on the map corresponding with where the node name was written on the map image[figure 1], was taken, giving latitude and longitude values. After the name, latitude, and longitude, the remainder of each CSV row was the connected nodes for each given node.

After the initial algorithm was shown to work, more features were added. First was the multiple destination option. The user can input how many destinations he would like to go to, and the program will correspondingly output the same number of "Enter the destination" prompts. Congestion was added first in the CSV file by placing a column of multipliers between the longitude and the connected nodes. This value is multiplied by the program when finding distance to the destination to compare closest nodes. The use of the congestion multipliers is based on a time input by the user, with the multipliers only being considered between 10am and 4pm. The value input for time is an integer in military time(1100 being 11am for example). On the map in figure 1, the congested routes are shown with yellow and orange highlights. The yellow corresponds with a 1.1 multiplier, or 10% slowdown and the orange represents a 1.2 multiplier, or 20% slowdown.

The final feature added to the program was the different forms of travel. The program takes in a string and uses that to determine the traveling speed of the user. If the string doesn't match with the options programmed in(bicycle and scooter), it defaults to the average walking speed.

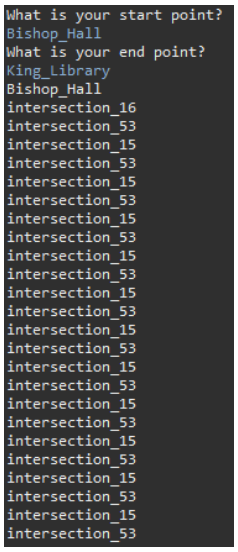

Figure 2: Example of back and forth error

5 Results

The results of the program are overall very promising. The shortest path algorithm almost always works to find the destination node with no looping. The program takes in to account the congestion of nodes depending on the given time and will change paths because of this. The variations in pathing are usually slight however, but attributing a higher congestion value to the nodes to force more paths would have lead to unreasonable times and routing for realistic cases. The program outputs an estimated travel time and this time is influenced by the congestion values on the path.

The multiple destination feature works exactly as it should. When multiple destinations are chosen, the output path crops the intermediate destinations to prevent duplicates. The program allows for "method of locomotion" input, so the user can choose scooter(an e-scooter), bicycle, or walking(the default). Depending on the choice, the output time to arrival will change.

The one error I have found with the program was initially a common occurrence but has become more of an edge case issue. The error in question being back and forth dithering at the node before the destination [figure 2]. The program will go back and forth between the closest and second closest node to the destination node, even though in the code simply arriving at a node connected to the destination should force the algorithm to complete. While this issue was larger initially, what I found was that the more nodes a destination was connected to, the more likely algorithm was to complete and not go back and

forth. I also found an error in my node class setup that switched longitude and latitude which made the routing do non-intuitive moves, which ceased on fixing the error.

6 Conclusion

In conclusion I think this project was a success and a good practice in parsing through a data object like a graph, to perform routing. All the features I had suggested in the proposals have come to fruition with the exception of having a car as a method of travel. For the size of the map and the limited number of actual roads on it, implementing a car would have shown had few options for routing besides the outside edges. Having done the a* algorithm now I would be interested to see what implementing a breadth first search and depth first search would be like, to compare coding complexity and efficiency against the a* algorithm.

7 Future Ideas

In the future, I think further testing is needed to determine why the algorithm goes back and forth right at the last node(s) before the destination. The function to count the number of node checks and cancel the search should it equal the total number of nodes does work, preventing infinite looping, but the program checks the last node's connected array so it should be seeing the destination always.

As far as adding features and complexity, I think it would be interesting to add more of the university, ideally the whole campus map eventually to see how the algorithm performs over even larger sets. The bottleneck to such a venture and the reason I didn't do it currently is that even the main campus area has a large number of intersections, or nodes. Even grouping closely packed crossings and representing them with a single node number left the CSV with 128 non-building nodes and 17 building nodes. All of these were input by hand from marking the points on google earth to find the longitude and latitude, to inputting the connected nodes as seen from the image(figure 1). Finding a program or API that I could work with to input these points would be very helpful.

A change to how the algorithm works, or how it deals with buildings could be interesting to implement as well. Currently all building nodes are just that, single points. For ease of use they are a single point instead of multiple entrance nodes, but making the algorithm look at buildings as a geographic area defined by the coordinates given instead of points, I think the pathing could be made more reliable(no more back and forth) as well as more accurate to real life scenarios.

References

[Jason Winn, 2013] Jason Winn (2013). Haversine.java. https://github.com/jasonwinn/haversine/blob/master/Haversine.java. Accessed: 2021-11-23.

[Wikipedia, 2021] Wikipedia (2021). A* search algorithm. https://en.wikipedia.org/wiki/A*$_search_algorithm. Accessed : 2021-11-23.