Generating Optimal Running Routes

John G. Crawford

Academy for Science and Design

Senior Project

Mrs. Amy Bewley

May 12, 2022

Code

[Github Link]

Goal

Having run every street in my hometown of Hudson over the course of half a year, I didn't want my journey to end there. I wanted to run all the streets in as many towns as I could in the shortest time possible. In order to do this, I needed to be smart with my route planning, maximizing the number of streets I could run in the least distance possible. In my journey to run Hudson, I was not very efficient at all, running some of the same streets 5+ times. It would take better planning and efficiency to get to even one town before I headed off to college and that is what lead to the idea for this project. I wanted to use Python to create a program that would do all the hard work for me. I would simply input an area of streets I wanted to run, and the program would spit out a route that covered all the streets in the least distance possible.

Process

Chinese Postman Distance Solver

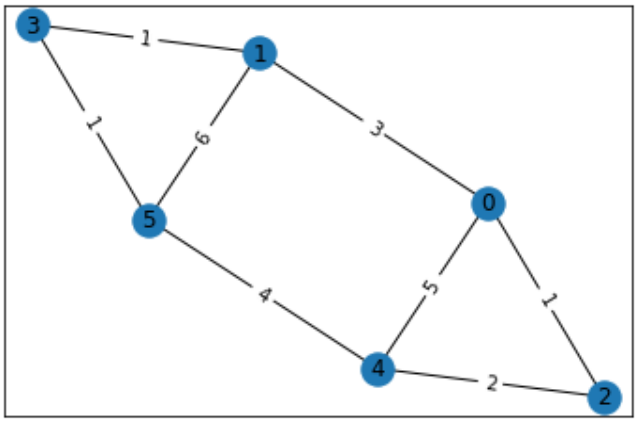
When starting this project, I figured the hardest part would be finding the Chinese Postman distance (the shortest distance to go across every edge in a graph), so I wanted to start there. I used python graphs composed of a list of lists, with each list representing a node, and each value in that list representing the distance between that node and the respective node (based on the index in the list). For example, if the first list (index = 0) contains a 3 in the 1st index position, that means there is a connection between node 0 and node 1 with a distance of 3 (as shown in Figure 1). If there is a 0, that means there is no connection between the nodes. These graphs would be the basis for the Chinese Postman Solver, as I figured it would be simple to later convert the map/street data into this list of lists graphs to use in the solver. This backward way of solving this problem is what leads to problems later on and almost a month of 'wasted' time.

Figure 1

Example of list of lists graph and output:

```
graph2 =                [[0, 3, 1, 0, 5, 0],
                        [3, 0, 0, 1, 0, 6],
                        [1, 0, 0, 0, 2, 0],
                        [0, 1, 0, 0, 0, 1],
                        [5, 0, 2, 0, 0, 4],
                        [0, 6, 0, 1, 4, 0],

                    ];
```



My initial program started off by adding the weights of all of the edges of the graph. If the graph was an Eulerian Circuit (traverse every edge of the graph only once and return to the start point), this value would give us our minimum distance. Then Fleury's algorithm could be used to return the Eulerian Circuit path (GeeksforGeeks 2021). However, Eulerian circuits are only possible when every node has an even degree (even number of connecting edges). In the graph in Figure 1, only nodes 3 and 2 have an even degree, each having two connecting edges. If odd nodes are present in the graph, which is extremely common, then it is not an Eulerian circuit because you have to cross at least one edge twice. In order to convert a non-Eulerian graph into a graph that is Eulerian, we need to add artificial “second paths” between the odd nodes to make them even. This is more complicated than simply connecting any two odd nodes, as we need to connect all the odd nodes so that the added distances are as low as possible.

First, a python function must be written to retrieve all the odd nodes from the path, and then one to take those odd nodes and create a list of all possible combinations between them. This is an example of brute force coding: trying every possibility instead of taking shortcuts to find the solution easily. Brute force coding is way easier to code for an amateur like me, hence the reason it is used here. This is further expanded on in Sharma (2020). Now that all the possible pairings are made, Dijkstra's algorithm can be used to find the shortest distance between the node pairings, and each of these distances can be summed up for each combination (GeeksforGeeks, 2022). After the added distances are calculated for every possible combination of odd nodes, we simply take the minimum distance and return the pairing which gave this minimum distance.

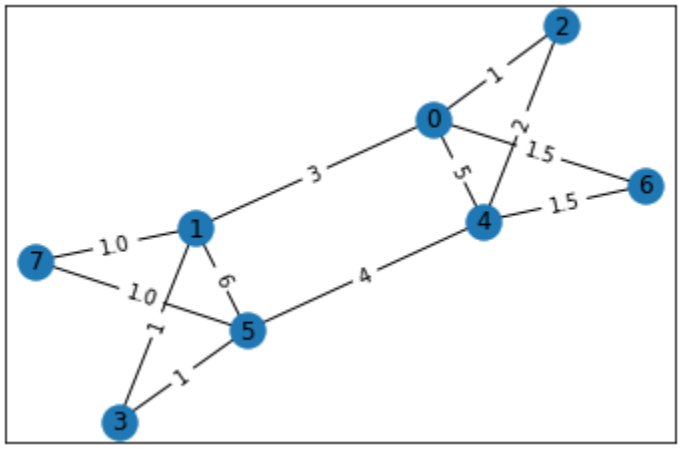
Once I have a list of the doubled nodes, we simply need to add those to the graph to complete the Eulerian circuit. The problem was that the list of lists graph format did not allow for indicating multiple links between nodes, so I needed to come up with a new solution. I was currently plotting the list of lists graphs using a package called NetworkX (details in NetworkX Developers (n.d.)). These graph structures are what the Fleury algorithm operates on, so if I can double the paths in NetworkX, the list of lists doesn't matter. As I concluded at the time, NetworkX doesn’t work well with doubling paths either (multigraphs are able to accomplish this although I did not know this). My solution was to create a new ‘artificial’ node that sat in between the two nodes that needed to be doubled. Each of the nodes would connect to this ‘artificial’ node, and the distance between them would be half the distance of the original Dijkstra's path. This would effectively add the link with the same distance, there would just be another node in between that didn’t actually exist within the graph. Fleury’s algorithm could then be used to output the Euler Circuit Tour!

This brute force method works well for small graphs as shown in Figure 1, but when expanded to entire road town-road networks, it can take a long time to compute, especially as the amount of odd vertices increases. This program would most likely not work well for complicated street networks, not to

mention the fact that I had no idea how I was going to convert the street data into the graphs the program uses to calculate distance. As indicated in Figure 2, the solution to add artificial nodes made the graph confusing to look at as well.

Figure 2

Example of graph including doubled paths and Euler Tour:




```
Chinese Postman Distance is: 28
Doubled paths are:  ((0, 4), (1, 5))


0-1
1-3
3-5
5-1
1-7
7-5
5-4
4-0
0-2
2-4
4-6
6-0
```

Note. This is the same graph used in Figure 1. That graph had 4 odd nodes: 0, 1, 4 and 5. The program indicates the Chinese Postman Distance, the doubled paths that add the least distance, and the path of the Eulerian Tour. Nodes 6 and 7 are the artificial nodes added in order to double the paths between 0, 4, and 1, 5.

Map/Streets Data

Now that I had completed the Chinese Postman Solver, I needed to get the actual data and convert it to the list of lists graph type used in the program. This was way more complicated than I anticipated. I initially thought the Chinese Postman Solver itself would be the bulk of the project, and that the data would be easy to implement into the program, but this was certainly not the case.

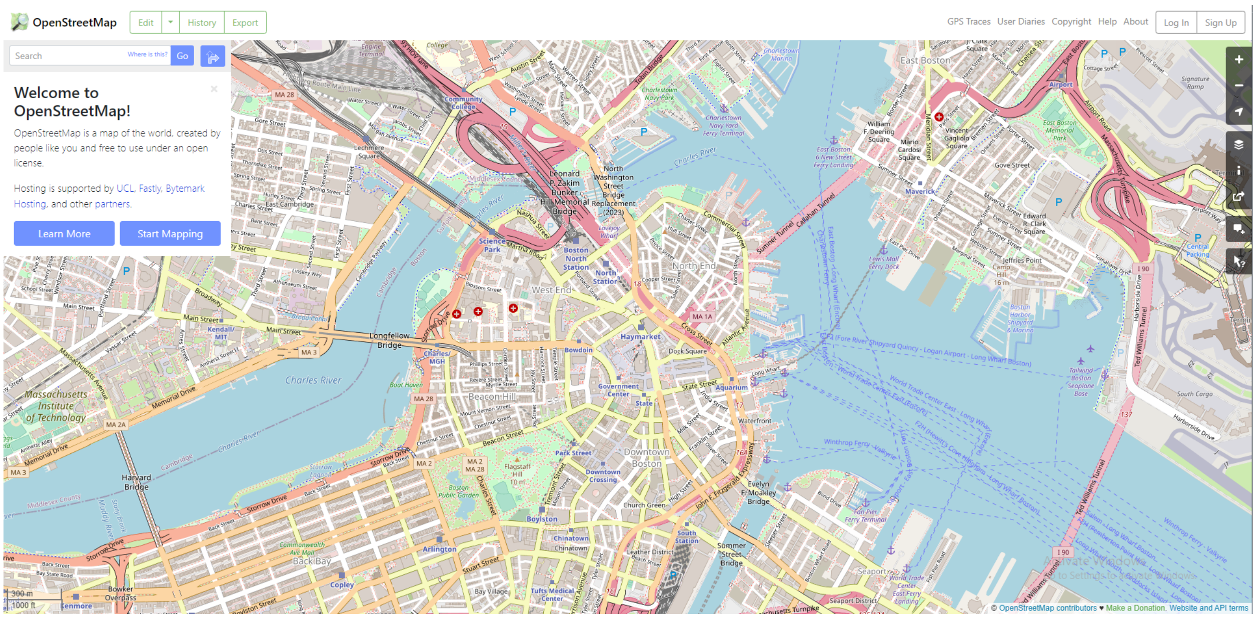
First off, I needed to find a map data source. In my search, I found three options: Google Maps, OpenStreetMaps, and OpenRouteService. It was tough to navigate through all the different packages and features for each source to get the data I needed. In accordance with the Chinese Postman Solver I created, I needed nodes at every street intersection as well as the distances of the edges that connected those nodes along the streets. Google Maps had this feature in the form of a distance matrix, but the Google Maps API was not free, and it was not open-source. I could've activated a trial just to complete this project, but I wanted it to last forever so that I could actually use it in the future. While scouring through blogs for the website I use to record my runs, CityStrides, I discovered that the app that I use to track my runs uses OpenStreetMaps. OpenStreetMaps is comparable to Wikipedia for maps. Using local knowledge, a community of mappers contributes to OpenStreetMaps in order to map out extreme detail, including rock walls in the woods and trails (OSM Foundation, n.d.). While this means that some rural areas aren't mapped as well, OpenStreetMaps is extremely detailed, open-source, and accurate. OpenRouteService was open-source and had a lot of the tools I needed, but it was just more convenient to use the same data CityStrides was using since that is how I tracked which streets I ran.

After weeks of toying around with these different data sources and different geospatial data plugins such as Folium, GeoJSON, GeoPY, and GeoPandas, I still couldn't figure out how to get what I needed. I had no way to automatically create nodes at each street intersection and calculate the distances between that node network. While looking into OpenStreetMaps more, I found it worked well with software called QGIS. QGIS is a "free and open-source geographic information system" that allows the user to visualize, edit and create geospatial information (QGIS, n.d.). There are plenty of user-made

plugins within QGIS, and it is through these plugins that I was able to accomplish what I needed using OpenStreetMap data.

Figure 3

OpenStreetMap:



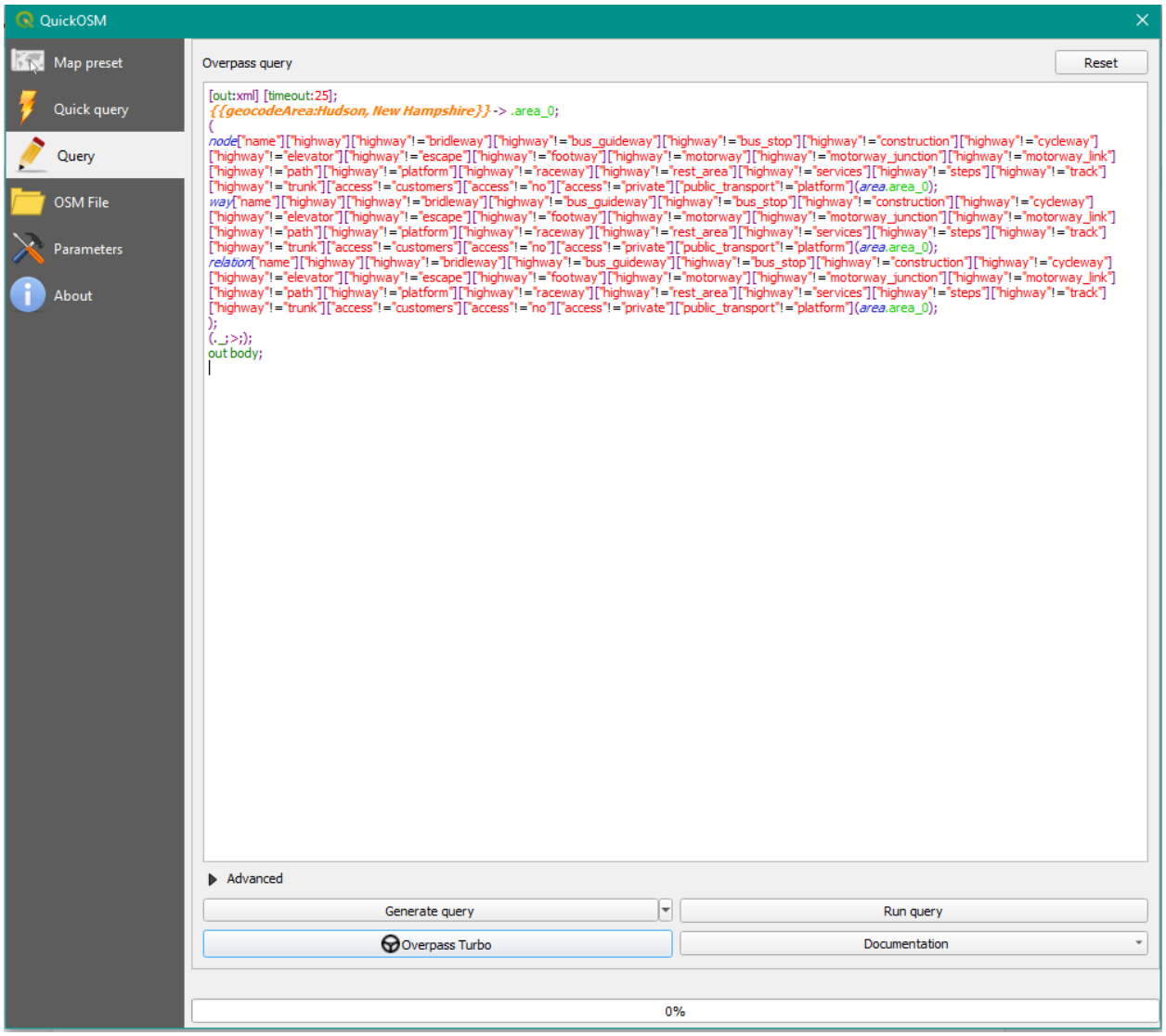
QGIS and OSM Querying:

OpenStreetMaps has feature indicators for everything included in its maps. Each feature is classified using a broad key (ex. Highway, boundary), with each key having multiple values for specific types of that key. A list of keys and values for map features in OSM can be found at OSM Foundation (2021). For example, 'highway' is a really broad term, and the values allow us to specify whether the highway is a residential road, a primary road, or something else. Using a plugin in QGIS called QuickOSM, we can use this key and value database in order to select the features we are interested in using. While being able to select what features we want to use, we can also select what features we *do not* want to use. This works better in our case, as we can simply state to ignore certain types of roads (private roads, interstates) and keep all the remaining ones. Luckily for me, CityStrides publishes the query they use (link) to select the streets that they count, so I simply used that same query, as shown in Figure 4, to

select all the valid streets to run in a specific town. The inputted town name is geocoded into coordinates through the plugin.

Figure 4

Example of a QuickOSM query:



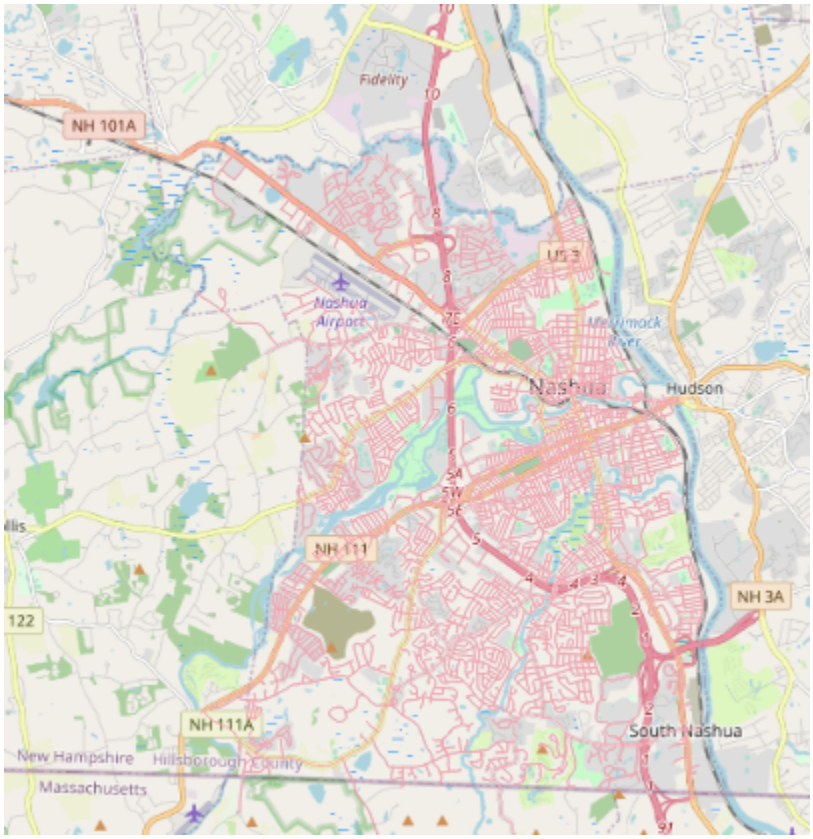
Note. Link to Copy/Paste

On a personal computer such as my own, it would take up way too much storage to download all the features of every street in the world, so I can simply allow the user to specify what town they want to run their streets in. If the user wants to select streets across multiple towns, there are other ways to

query the data, such as only selecting the streets that are viewed on the map (canvas extent function) or using the 'around' function to select streets around the specified town. Once the streets are downloaded onto QGIS, there are a few problems. If there is a street selected that crosses multiple towns, it will select the entire road and not just the part that is within the town borders. In order to fix this, the boundary of the town needs to be queried using QuickOSM. The vector function 'clip' in QGIS can then be used with the line layer of the streets and the polygon of the town border in order to only keep the streets that are within the border polygon.

Figure 5

Streets in the town of Nashua, New Hampshire selected using CityStrides Query:



When streets are queried in this way, it is not possible to only select a small portion of an individual road; if you select a portion of one road, it will select the entire thing. In order to prevent this, the processing vector geometry tool 'explode lines' can be used to split the lines into hundreds of little

line segments that can be individually selected. This helps if you are planning a route where you only want to cover a portion of a really long street so that it just selects the section of it that you want. In order to do the selection process of the streets that you are making a Chinese Postman Route for, the 'select features by polygon' can draw a polygon around the area of interest.

Figure 6

Example of street network selected to find Chinese Postman Route

Note. As seen in the image, all the streets outside the border of Nashua are *not* highlighted due to the border clip. When creating a polygon around the desired street network, long roads such as 'broad street' in the figure are not selected in their entirety due to the explode lines tool.

Chinese Postman Solver using QGIS

Now that I could select the features that I wanted to use for the Chinese Postman Problem, I needed to figure out how to convert this into the list of lists graph format used in that program. I tried using several tools, such as the distance matrix and the sum line length tools in QGIS, but this returned straight line distance between the nodes and did not return the distance while following the path of the street. There was also no way to return *only* the distance between nodes that the current node was actually attached to via a street. You could only return an *x* amount of nearest distances to nodes, so this didn't work. Doing some more digging, I found something amazing. Someone had already created a Chinese Postman Solver using QGIS, and it was open-source and free to use. The only problem: it was

slightly old and wasn't currently functioning. Upon finding the plugin GitHub and searching through the python files, I found that the plugin was using some plugin functions that didn't exist anymore. I was able to find the current, updated versions of those plugins to return the code to a functional state. The plugin allowed me to select streets in the same way I described in the *QGIS and OSM Querying* section, and it outputted a new layer in QGIS which showed arrows directing the path.

I spent a decent amount of time reading through the python files for the plugin and understanding the method for the solver and comparing it to my previous attempt. I discovered QGIS about a month before the project was due, and it would've taken longer than that to learn and integrate it into a python program, so it was extremely valuable that I had this plugin that already had the basic functionality implemented. In overview, the plugin has an "__init__.py" file that initializes the QGIS interface. While the current package no longer requires this plugin to work, I did not have the time to learn about the nature of these files and how to create a plugin that doesn't need them. There are 3 more python files included in the package, "chinesepostman.py", "postman.py", and "postman_test.py". Postman.py contains all the functions for inputting the data selected from QGIS, converting it to NetworkX graphs, and finding the optimal route. Chinesepostman.py is the main file, where all these functions are combined in order to create the output layer that contained the route, along with the outputted distance. Postman_test.py is a self-checking test that calculates values based on a default CSV file and makes sure the values match up with what they are supposed to be. This is what I used to make sure I fixed the program correctly when changing out the packages.

First, the plugin does some data validation on the selected features in QGIS, making sure that an area is selected that contains vectors made of lines. The program won't work with just a layer of unconnected points. Using the QGIS function QGSDistanceArea(), and a recursive function that yields all the possible combinations of nodes, we can build a NetworkX graph using the 'add_edge' function. After this graph is built, a function in postman.py can be called in order to return solely the largest connected

graph. The user may accidentally select a road that is not connected to their network, so in this case, the program will ignore that unconnected road and only compute on the larger network.

Once we have a complete NetworkX graph with one connected component, we can find the Chinese Postman Path. The plugin here took a slightly different approach than I did when I first attempted to build the program without QGIS, and instead of simply taking all the odd nodes, it just made a new graph with *only* the odd nodes and the shortest paths between each of them. Instead of finding all possible combinations of the nodes and finding the one that adds the least distance, this plugin simply used a function that automatically did this. The NetworkX function "max_weight_matching" gives us the optimal node pairing if we assign the weights as negatives so that it actually calculates the minimum and not the maximum. The function "build_eulerian_graph" is then used to return the original graph that now contains the optimal additional node pairings to make the odd nodes even. Then another NetworkX function that I wish I had originally known about, "eulerian_circuit" can be used on the Eulerian graph in order to find the optimal path. Some additional programming is done to just return the nodes.

After we get the Eulerian graph and node path of the Chinese Postman route, we can use that in order to build our output. In order to find the distance, we simply add up the weights of all of the edges in the Eulerian graph since each edge is only crossed once. Comparing this to the sum of all the weights of the edges in the original graph, we can get a feeling of how efficient the route is. This shows us how much distance we have to do due to doubled paths. As of now, while I still search for a way to better represent the path, the program also outputted a list of node coordinates that can be copy-pasted to use elsewhere. This is a feature that is solely used for development purposes. The plugin also uses QGIS functions in order to output a single PolyLine that traces the entire route with arrows to indicate directionality.

Overall, the plugin is simpler than I thought when first attempting to create it from 100% scratch at the start. The hard part for me was transforming the QGIS data into NetworkX graphs and creating the output layer, but once I had the data as a NetworkX graph, it was easy to follow and modify the code to get the output I desired. This plugin essentially combines several functions from plugins that do all the heavy lifting, and most of the code is related to the QGIS aspect of it.

Problems with the Chinese Postman Solver Plugin

A few problems with this plugin was that the arrows would seemingly switch directions on the output layer every once in a while (supposedly because the same optimal path could be run more than one way and still be optimal), so it was hard to track what the optimal route was. It was clear which roads were doubled, however, and QGIS allows you to export the layer as a GPX file to upload to any Garmin running watch. The problem was that my Garmin watch didn't support this route feature, so I could only view it through the Garmin app by following the path of the route with my mouse. I still need to find a way to reconstruct the route in a more presentable manner. The plugin also didn't allow me to define a starting point for the route, it chose one seemingly at random. Displaying the true starting point on the layer would be a waste of time due to the fact that the route is the same no matter what starting point you choose, the order of directions is just different. If we can't specify the starting point, it is useless to include it.

Figure 7

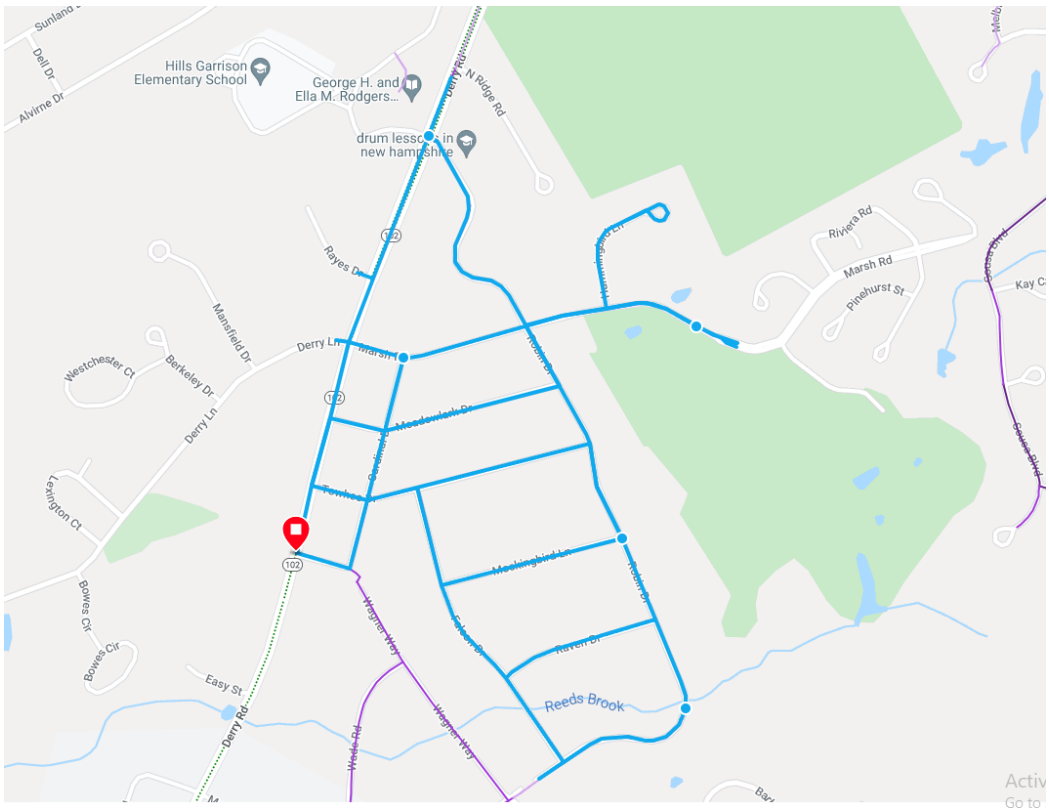
CPP Plugin's QGIS Layer Output

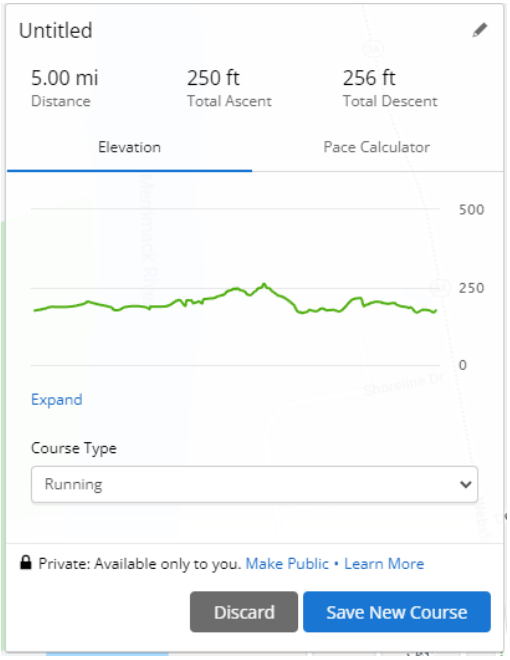
Note. As you can see, it is very confusing to figure out what the actual route is using solely this.

The coordinate system used for OpenOSM querying (EPSG: 4326) uses coordinates for distance as opposed to meters, so the output units for the plugin were also something that was not easily understood. In an attempt to convert this to meters, I tried changing the coordinate system to something such as EPSG: 3857 which uses meters but found no success and had to do too much troubleshooting. Decimal degrees are also essentially impossible to do a straight conversion to meters due to the differences in length of a degree based on what latitude or longitude you are at (it would've been possible with complicated calculations if I was given the components, but I was only given the total). If you export the path to Garmin, it fixes this problem and outputs the distance in miles. This is not practical though, and in the future, I would like it to be able to output a distance right there in QGIS.

Figure 8

Garmin Output





Note. Way more presentable than the plugin's QGIS output and the distance is in miles instead of meters.

If you want to solve a really large street network with this program, the time complexity is very high and it takes a long time to process and display an output. When I calculated the optimal route for the entire road network of Hudson, it took upwards of 30 minutes to compute.

In the future, I also want to work to make the process more simple and easy to get the Chinese Postman path for the area you want. As of now, you have to query in the proper streets, do some modifications to their geometry, and then run the program and upload it to Garmin (which not everyone has). There should be a way to type in a town name and instantly display a graph of all the streets in that town with each road “exploded” (as described earlier), and not multiple steps to accomplish this. Ideally, I would just be able to draw a shape around the streets I want to calculate this for and have it output the path, but unfortunately, I was not able to accomplish this in time.

Conclusion

If I was doing this all over again, I would definitely stress focusing on the macro ideas instead of just diving in and just doing random things in hopes of it working. I wasted a lot of time doing things that didn’t work during this project, and if I had better planning and idea of what I really needed to do, I would have realized that something wasn’t going to work long before I actually did. Once I found QGIS, my time was used productively toward the end goal, but it took me three months to arrive at that point. I did this project with a backward process, attempting to build the CPP solver before I knew how to integrate the street data, and that was my biggest mistake.

I came into this project hoping to at the very least provide something that I could actually use as I tackle neighboring towns this summer and try to run them as efficiently as possible, and I did that. It was not as general and easy to use as I wished it was, but in the end, it did what it was supposed to. My next step is to make a better output of the route and distance in meters or miles instead of degrees. After that, I want to make it take fewer steps to accomplish and easier for other people in the CityStrides

community to use. It is not currently 100% open-source due to the usage of Garmin, and I want to change that as soon as possible, hence why I want to figure out the routing problem first.

References

GeeksforGeeks. (2021, September 15). Fleury's algorithm for printing Eulerian Path or circuit. GeeksforGeeks. Retrieved May 1, 2022, from https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/

GeeksforGeeks. (2022, February 22). *Dijkstra's shortest path algorithm*. GeeksforGeeks. Retrieved May 1, 2022, from https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

NetworkX developers. (n.d.). *NetworkX documentation*. NetworkX. Retrieved May 1, 2022, from https://networkx.org/

OpenStreetMap Foundation. (2021, February 15). *Map features*. OpenStreetMap Wiki. Retrieved May 2, 2022, from https://wiki.openstreetmap.org/wiki/Map_features

OpenStreetMap Foundation. (n.d.). *About OpenStreetMap*. OpenStreetMap. Retrieved May 1, 2022, from https://www.openstreetmap.org/about

QGIS. (n.d.). Retrieved May 1, 2022, from https://qgis.org/en/site/

Sharma, A. (2020, November 10). *Chinese postman in python*. Towards Data Science. Retrieved May 1, 2022, from https://towardsdatascience.com/chinese-postman-in-python-8b1187a3e5a#cf3b