CSC111 Project: Route optimisation for emergency vehicles based on real traffic congestion data

Kaartik Issar, Aryaman Modi, Garv Mohan Sood, Craig Katsube

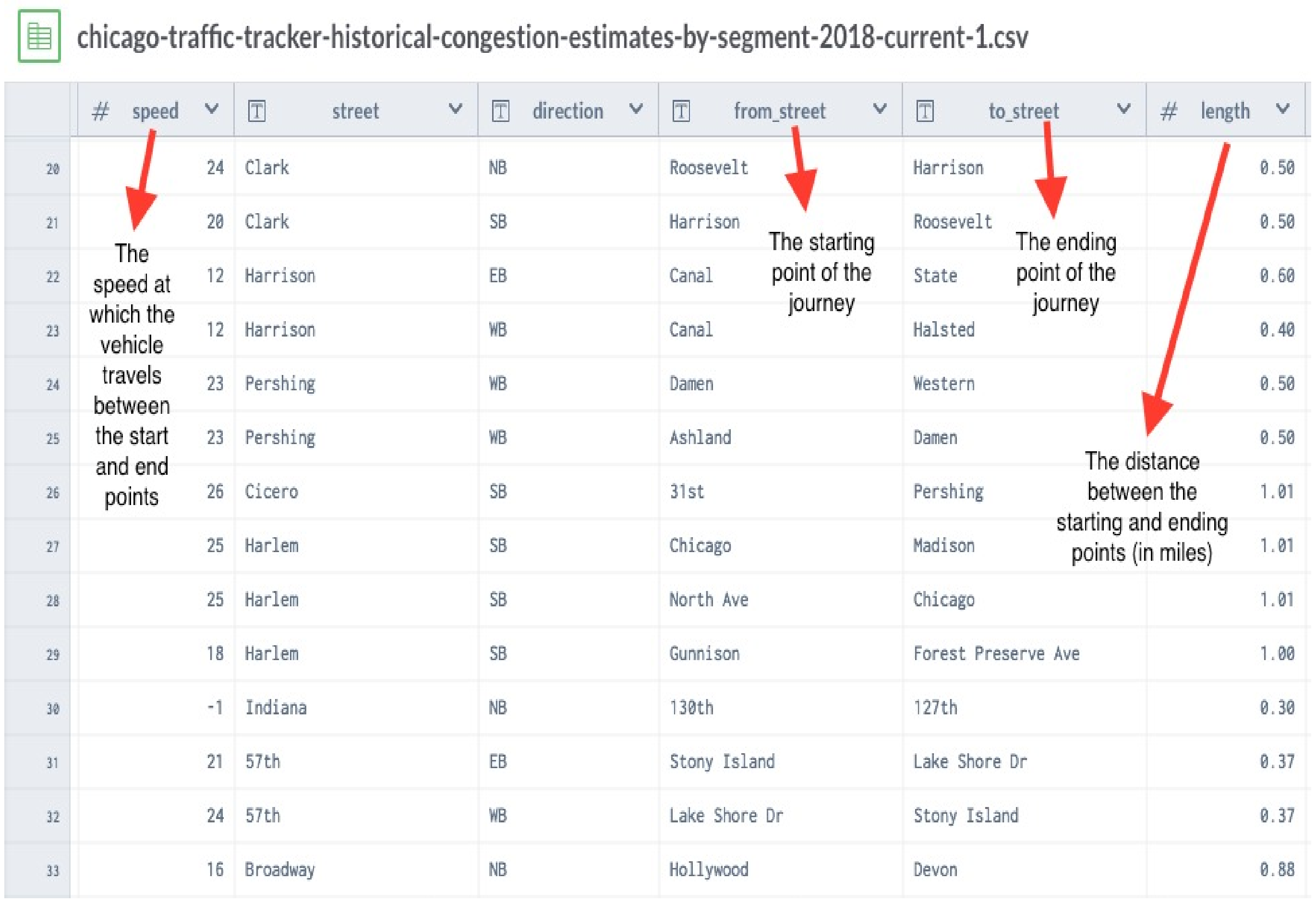
Tuesday, March 16, 2021

Problem Description and Research Question

Despite living in an ever growing and technologically advanced world, one of the alarming setbacks, human population still suffers from is the pathetic state of transportation for patients and accident victims in critical need. According to statistics, in 2017, 20 percent of the emergency patients' deaths were caused by traffic congestion (The Nation, 2017). As the world is turning into a concrete jungle, and more and more families are buying automobiles, it is becoming increasingly difficult for ambulances to respond to emergency needs in ideal times set by WHO (10 minutes). It is important to realize that in a lot of cases, we have the medical infrastructure and facilities to save the life of an emergency patient, but due to the inadequate transport infrastructure and congestion in the cities, we are failing to even get these patients to the help they need. Thus, it is the need of the hour that, we come with ideas to overcome this issue – which is costing us humans lives. In India, every second cardiac arrest patient takes $>$ 400 minutes to reach hospital ("More than 50 percent Heart attack patients reach hospital late",Hindustan Times, 2017). This is far above the ideal interval of 30 minutes. There are numerous articles published on digital media that have raised this issue – "20 per cent of emergency patient deaths blamed on traffic jam delays", published in The Nation, Thailand and "Choking life: 12 on way to hospitals die in traffic jams", published in The Times of India to name a few. Thus, this problem is not restricted or unique to a particular region, and is universal, thus it is our responsibility as students, who are fortunate enough to have the tools and knowledge, to help society overcome such issues.
Therefore, we propose to **develop a program that takes the locations – one starting and one ending point (there may or may not be an intermediate location between the two paths) – and finds an optimized path between the two locations, based on the data set we have adopted for our project.** The particular case study we are going to build our project on is the city of Chicago – the 3rd most congested city in USA. Helping those in emergency needs, to reach hospital on time can save a lot of lives and improve life expectancy as a whole.

Dataset description

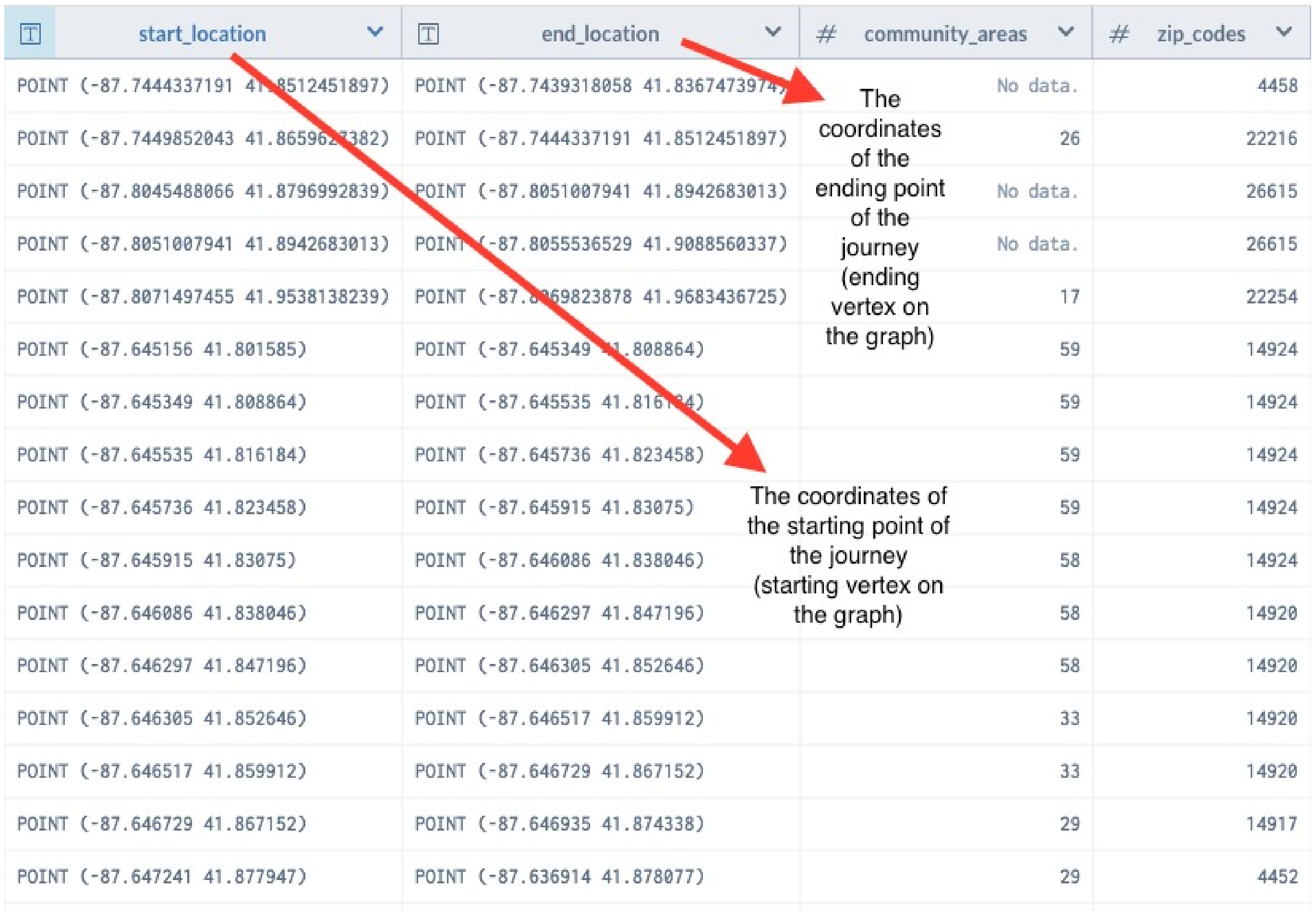
We used a subset of the dataset "chicago traffic tracker historical congestion estimates by segment 2018 current1". The data was found and obtained from data.world. The dataset we are using to construct our graph contains the historical estimated congestion for over 1,000 traffic segments in Chicago's non-freeway streets. This data has been collected in real-time by continuously monitoring and analyzing GPS traces received from Chicago Transit Authority (CTA) buses.
The dataset contains many columns - start and end locations (streets) and the distance between these locations to name a few. Furthermore, it also has specific latitude-longitude coordinates for each of the respective start and end locations. The dataset also provides us with the speed of the automobile while travelling from the start to the end point and the number of buses encountered on the route. Using the pandas and csv libraries, we filtered the original dataset and obtain a new dataset, which doesn't contain data of rows where the value of speed attribute was 0 or -1. These were basically the rows for which the data about speed attribute wasn't available. In addition to this, we also only included the following columns in our new dataset - time,segment id,speed,street,direction,from street,to street,length,bus count,hour,day of week,month,start latitude,start longitude,end latitude,end longitude.


chicago-traffic-tracker-historical-congestion-estimates-by-segment-2018-current-1.csv

	# speed	street	direction	from_street	to_street	# length
20	24	Clark	NB	Roosevelt	Harrison	0.50
21	20	Clark	SB	Harrison	Roosevelt	0.50
22	12	Harrison	EB	Canal	State	0.60
23	12	Harrison	WB	Canal	Halsted	0.40
24	23	Pershing	WB	Damen	Western	0.50
25	23	Pershing	WB	Ashland	Damen	0.50
26	26	Cicero	SB	31st	Pershing	1.01
27	25	Harlem	SB	Chicago	Madison	1.01
28	25	Harlem	SB	North Ave	Chicago	1.01
29	18	Harlem	SB	Gunnison	Forest Preserve Ave	1.00
30	-1	Indiana	NB	130th	127th	0.30
31	21	57th	EB	Stony Island	Lake Shore Dr	0.37
32	24	57th	WB	Lake Shore Dr	Stony Island	0.37
33	16	Broadway	NB	Hollywood	Devon	0.88

The speed at which the vehicle travels between the start and end points
The starting point of the journey
The ending point of the journey
The distance between the starting and ending points (in miles)

Figure 1: Dataset Screenshot 1


The coordinates of the ending point of the journey (ending vertex on the graph)
The coordinates of the starting point of the journey (starting vertex on the graph)

start_location	end_location	community_areas	zip_codes
POINT (-87.7444337191 41.8512451897)	POINT (-87.7439318058 41.8367473974)	No data.	4458
POINT (-87.7449852043 41.8659627382)	POINT (-87.7444337191 41.8512451897)	26	22216
POINT (-87.8045488066 41.8796992839)	POINT (-87.8051007941 41.8942683013)	No data.	26615
POINT (-87.8051007941 41.8942683013)	POINT (-87.8055536529 41.9088560337)	No data.	26615
POINT (-87.8071497455 41.9538138239)	POINT (-87.8069823878 41.9683436725)	17	22254
POINT (-87.645156 41.801585)	POINT (-87.645349 41.808864)	59	14924
POINT (-87.645349 41.808864)	POINT (-87.645535 41.816184)	59	14924
POINT (-87.645535 41.816184)	POINT (-87.645736 41.823458)	59	14924
POINT (-87.645736 41.823458)	POINT (-87.645915 41.83075)	59	14924
POINT (-87.645915 41.83075)	POINT (-87.646086 41.838046)	58	14924
POINT (-87.646086 41.838046)	POINT (-87.646297 41.847196)	58	14920
POINT (-87.646297 41.847196)	POINT (-87.646305 41.852646)	58	14920
POINT (-87.646305 41.852646)	POINT (-87.646517 41.859912)	33	14920
POINT (-87.646517 41.859912)	POINT (-87.646729 41.867152)	33	14920
POINT (-87.646729 41.867152)	POINT (-87.646935 41.874338)	29	14917
POINT (-87.647241 41.877947)	POINT (-87.636914 41.878077)	29	4452

Figure 2: Dataset Screenshot 2

Computational Overview

Dataset Transformation

As described in the data set description, the original data set we obtained was humongous and rogue, so we filtered the original data set using the pandas library. We first converted the original csv file into data frame object using the read_csv function and then subsequently selected specific columns to our need. We also removed row entries where the speed values were 0 and -1. Lastly, we converted the final data frame into a csv file using the to_csv function and saved the new csv file with the name "transformed_final.csv"

Tkinter GUI Interface

On a high level, the main file is responsible for starting the input GUI device, filtering and processing the response taken from the GUI and passing it into the output maps. More specifically, the main file uses our dataset to populate the GUI—which asks the user for an input—and connects the GUI's output to a locally defined function. This local function takes in the input options and filters the dataset based on the user's input and then creates a graph from this filtered dataset. Using the graph and the start- and end-points input into the GUI, the program uses our shortest path script and returns the list of locations (latitude longitude) which is then passed into the output visualizers. The GUI is built to let the user choose options without any knowledge of the underlying datasets. When choosing an option, the other options are automatically restricted to choices that are compatible with the current selection of options. This dynamic is executed through the mediator pattern, where each input Widget is a colleague which notifies the mediator to filter the data and tell the other input Widgets to restrict their choices to the filtered data. A drawback to this approach is that our dataset is incredibly large, so iterating through the data to filter it takes time on the order of single-digit seconds. The wait is not incredibly large, however, and the ease-of-use saves much more time (as otherwise the user would be fiddling around with combinations that are not supported by guess-and-check); furthermore, although not present in this project, from a design-perspective, the filtering logic can potentially be extracted and replaced with a more efficient strategy.

Computation

The core concept at use for our Computation is based around Graphs. We represent the Chicago City as a graph, and every location given in the data set as a vertex of the graph. We define two classes - a Graph class and a vertex class. The Vertex class initializes vertices each with an item (The name of the street) and its neighbors along with their corresponding edge weights. It also initialises each vertex to store latitude and longitude points corresponding to that location as we utilise them in our visualisation. We use these to construct our Graph, which has weighted edges representing the amount of traffic between any 2 connected vertices (streets). The weight in each of the weighted edges is the amount of time taken by the automobile to travel between the 2 vertex points. These weights are subsequently used by the path calculator functions that computes the least congested path(s) i.e., the path(s) that takes the least amount of time to travel between the two chosen points. Our Graph class represents vertices in a dictionary with the items (street name) as keys and corresponding values as instances of the Graph class.
We included the following helper functions in the Vertex and Graph classes, which we use in our computations to find optimised paths. We have included brief descriptions for some of the important helper functions:
1) **check_connected** : Uses a recursion algorithm to check whether the given location (vertex) in Chicago city is connected to another given location.
2) **print_all_connected** : Prints all the street locations that are connected to a given street. Prints the immediate neighbours of a vertex, and then recursively prints its neighbours' neighbours and so on.
3) **add_vertex** : Adds each street from our dataset to the graph.
4) **add_edge** : Adds a path between 2 street locations. This represents the road connectivity in the graph.
5) The **get_all_paths** function in the Graph class uses the **paths** function from the Vertex class to return all possible paths between any 2 connected points inputted by the user.
6) **get_weight** : Returns the time taken (weights) to travel between the 2 given street locations. It is calculated using the 'length'(of street) and 'speed' values from the dataset.
7) **get_all_lat_long** : Returns a tuple with both the starting and ending latitude and longitude coordinates of the inputted street locations (vertices).
8) **connected** : Uses the print_all_connected function from the vertex class to print all the street locations that are connected to the given street.

The computation for a singular shortest path follows the classic Dijkstra's algorithm of creating paths in sequential order by the shortest cumulative path. This approach is appropriate because we do not have negative weights and we have only one type of data stored as weights in edges (time).
• The Dijkstra algorithm was implemented recursively, using a heap of Path objects (that have the __lt__ special method implemented to compare cumulative path weights).
• The algorithm begins at the target position because the path direction is built in reverse order—essentially forming a flipped tree where all leaves (some arbitrary vertex in the graph in the same connected component as the end point) path upward towards the root (the end point).
• At each recursive call, the algorithm follows the Dijkstra concept at each iteration:
– returning the path if the last vertex in it is the target point (the start vertex since we began at the end vertex)
– getting the path with the shortest cumulative weight (popping a Path from the heap) and marking the vertex at the end of this path as visited (by the Dijkstra algorithm assumption, the path formed by the first time a vertex gets visited is the shortest path to that vertex).
– iterating through all non-visited neighboring vertices and adding the path for this vertex to the heap.
On a high level, the computation of multiple shortest paths passing through a specified set of interim vertices can be done by generating all paths from points A to B, filtering these paths by ones that pass through the required interim vertices, sorting these paths by weight, and returning some number of the shortest paths. From this description, it is painfully apparent that this computation is strenuous and infeasible for any real dataset, so a more complex process was developed to reduce the computational running time.
Once again on a high level, we first reduce the graph to a near-complete subgraph with only the vertices required for the path and the edge between each pair of vertices is the weight of the shortest path between these two vertices (calculated with the Dijkstra's algorithm). Computing all paths from A to B passing through every vertex in this reduced graph is a much more computationally feasible task, especially when considering that users will most likely only be asking for a number of interim locations countable on both hands. Using the path created from the reduced graph, we can then combine all the shortest paths between each pair of vertices in the path to get the multiple total paths from point A to B passing through all specific interim points.
On a detailed level, we first create a mapping of each relevant vertex to another mapping containing another vertex associated with the shortest Path to that vertex using the Dijkstra algorithm. From this mapping, we create the reduced graph using the total weight of each path as the edge weights.
Our vertex function, paths is responsible for returning all the paths between two points in our reduced graph. The function works recursively so as to find the path from our starting point to our ending point, if it reaches the ending point it stores that path in a list variable which is mutated by our paths function.The graph function get_all_paths returns this list variable containing all possible paths from our starting point to our ending point.
In our shortest_path_calculator file we further operate on this list of possible paths, to obtain the shortest path between them according to edge weight. The function connected_with provides us with this shortest path.
With the paths using the reduced graph, we apply the mapping again to get the shortest paths between each pair of consecutive vertices in the calculated paths and concatenate all these shortest paths together in our function gets_original_gives_full_path to form the total shortest paths passing through all the interim vertices.

Visualization

For the visualisation aspect of our project, we use two different libraries - matplotlib and gmplot to produce two different visualisations for the optimised path obtained by our above mentioned algorithms and functions.
In one half of our visualization, we use the matplotlib library. Firstly, we obtain a snippet of the map of Chicago using OpenStreetMap. To compute the longitudinal and latitudinal dimensions of the map we did a brief computation in our datatransformation.py file. In this computation we iterated over the whole data set and found the largest and smallest latitudes and longitudes out of all the possible locations in our data set. These acted as the end points when obtaining the map from OpenStreetMap. In this visualization, we plot the given list of locations (optimised route) on the map of Chicago City. Each location is labeled by the name of the location, and the number at which the location appears in the path. While moving the cursor over the map, one can see the longitudinal and latitudinal values of the point where the cursor is.
In the next half of our visualization we use the gmplot module. We have two functions one which outputs the path for just two points and another which outputs the path for two points with the specified intermediate points. (This was done so as to have two files storing the output of the plotting.) The function first generates the map using the

latitude and longitude points for the center of the city of Chicago. It then places the different points in our path on the map on the basis of latitude and longitude and it plots the path between them. The above operations are carried out by the gmplot.scatter, gmplot.marker and gmplot.plot functions. The user can bring the cursor on top of the markers so as to see the name of the point. The start and end points are displayed by black markers and the intermediate points are displayed by blue markers.

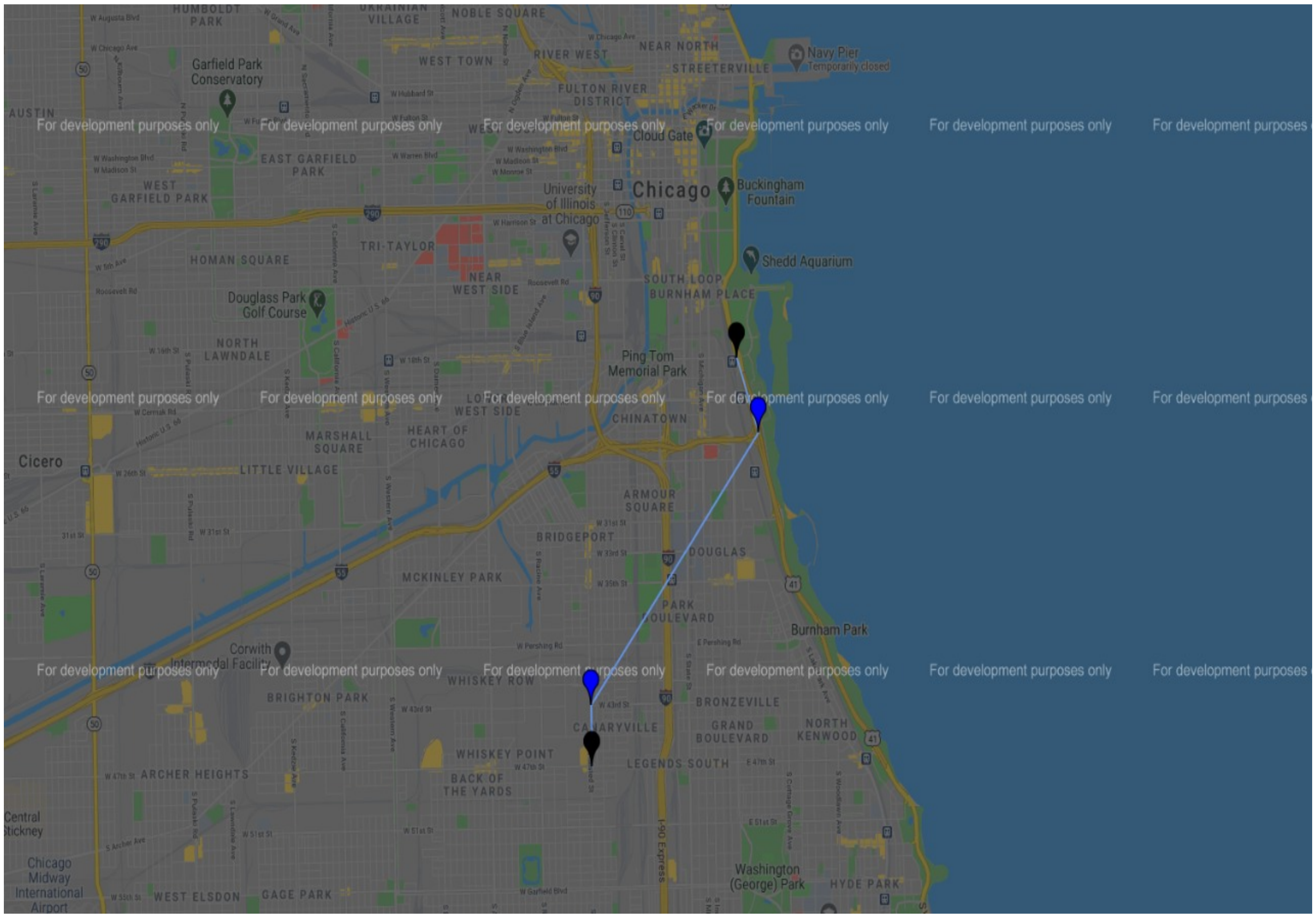


Figure 3: Map 1 - Route from 47th to 18th via the intermediate streets '43rd' and 'I-55 Expy' using Gmplot

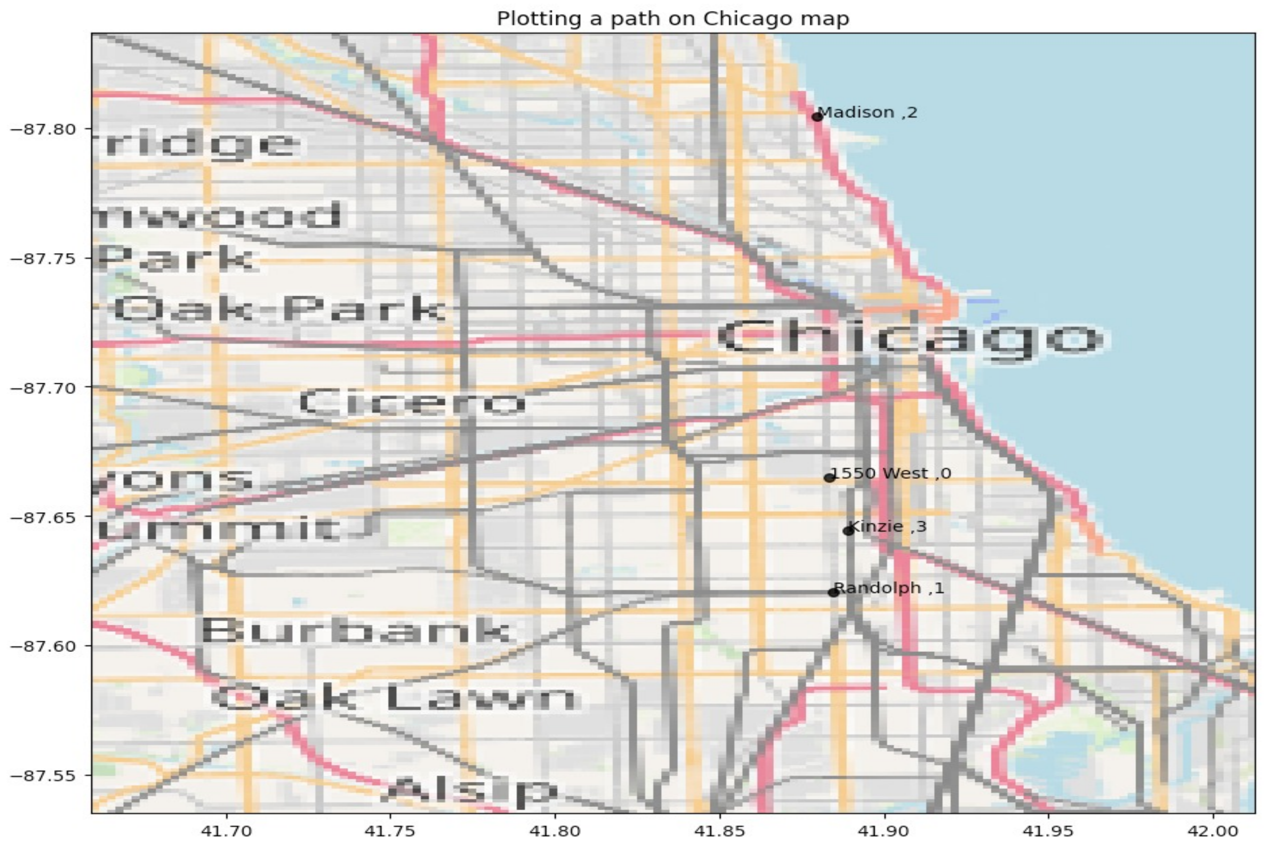


Figure 4: Map 2 - Route from 1550 West to Kinzie using Matplotlib

Obtaining data set and running the program

Our program functionality is entirely based on the Chicago city traffic data. For reference - we first downloaded the original data set in csv format from data.world (link : https://data.world/cityofchicago/sxs8-h27x) and then transformed it by removing certain row entries for which the speed attribute was not available. We also included only a subset of columns in our data set from the original data set. We would be sharing a zip file containing both the original csv file and the transformed csv file using "UTsend". It has been sent to the course email address from the following email address - **kaartik.issar@mail.utoronto.ca** For the purposes of our computations we will be using the file 'transformed_final.csv'. The user should ensure that this file is in the same folder as the rest of the python files.
We suggest that you set Google Chrome as your default browser as it is the most compatible with our visualisation.

Instructions

1. Download all the files from MarkUs and save them in the same folder.
2. Open requirements.txt in PyCharm to install all of the Python libraries needed to run our project.
3. Pickup the datasets sent via UTSEND from the following email - **kaartik.issar@mail.utoronto.ca** (Please ensure the dataset files('transformed_final.csv') and all other files are in the same folder)
4. The user needs to run the main.py file. Once this is done, a popup (image shown below) for inputting values opens.
5. The user gets to select the day of the week(1 to 7, where 1 is Monday), the time (24hr clock) and the months (February or March) in the GUI popup.

6. Once the user has selected the above information, they are provided with all possible start and end points. They can traverse through the drop down menus using the up and down arrows of the keyboard and press the enter key to select the points.

7. For the program to run, it is mandatory for the user to select the start and end points.

8. There is also an option for the user to add intermediate points. They can choose the first intermediate point in a similar manner to the start and end points, however to choose more intermediate points they can press the 'Add' button and choose the location they want to add. They can repeat this procedure to keep adding more intermediate points. The user can press the 'Remove' button if they would want to remove any intermediate points.

9. Once the selections are done, the 'Export Selection' button should be pressed to visualize the route.

10. The matplotlib visualization opens as a pop-up on the screen and the gmplot interactive visualization opens in the web browser.

11. In order to run the file again the user has to run the main.py file on a new python console (this is due to the Tkinter GUI package used in the main file).

    Note: It is observed that in MacBooks, when the main.py is run and the inputs are given, the gmplot visualisation is first saved as "map2.html" or "map1.html" in the same folder as other files for our project. We then open "map2.html" or "map1.html" and select Google Chrome as the browser to obtain the visualisation. (It is saved as map2.html when there are no intermediate points between the start and end points, and saved as map1.html where there are intermediate points between the start and end points)

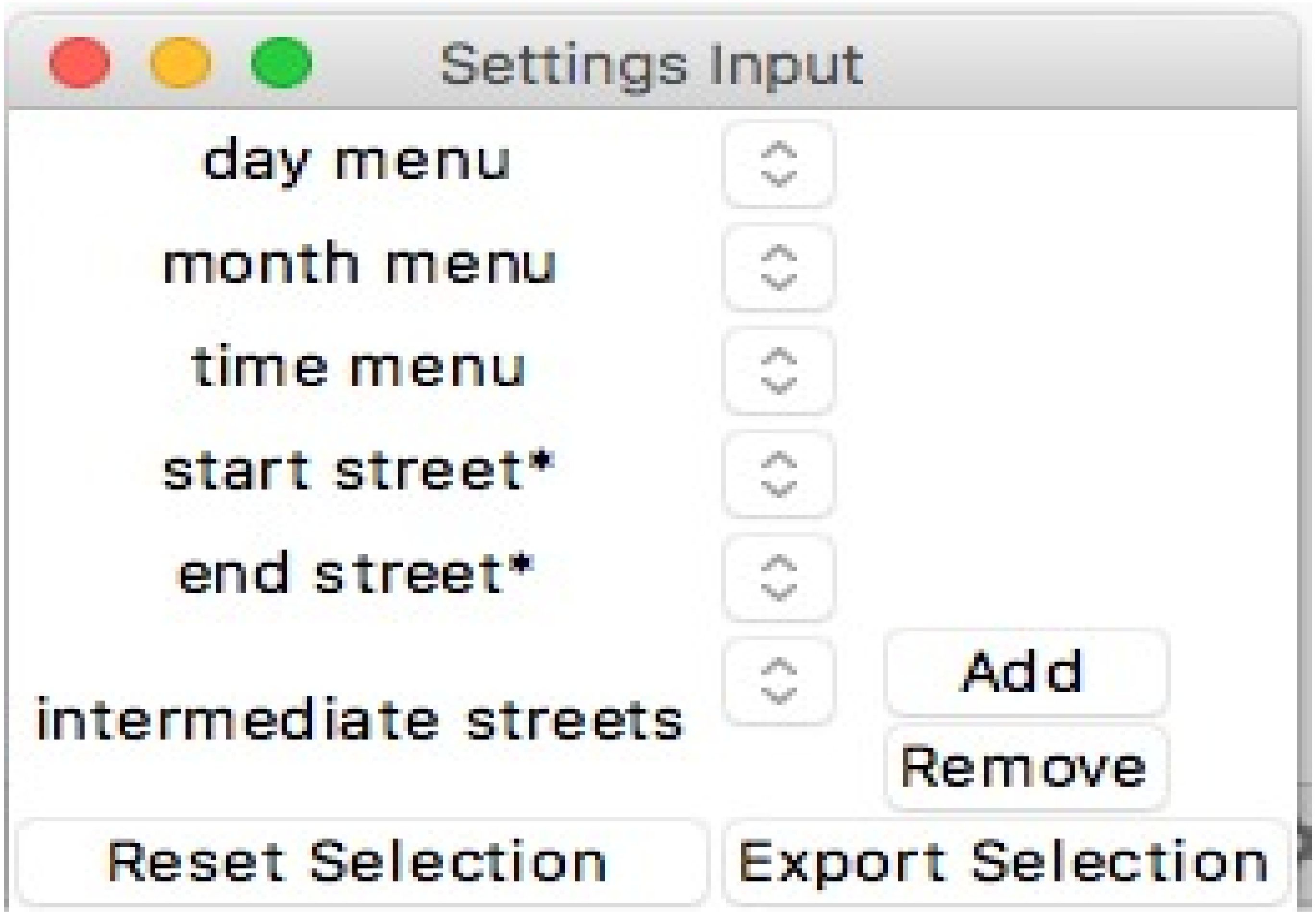


Figure 5: GUI popup for inputting details

Changes from Project Proposal

For the most part, we have have not made any major changes in our project from the project proposal. One of the significant changes we did make at the grass root level of our project was changing what we consider as the weighted edge between two given locations. Weights between two vertices play an imperative role in our program algorithm. Earlier we considered traffic congestion score - the ratio of bus count to length- as the weight between two given locations. On contemplating over this further, we realise that a drawback of our dataset's bus count data attribute is that it only tells us about the number of buses on a path. It doesn't tell us about the vehicle load caused by other vehicles present on that path - and thus fails to give a comprehensive description of the traffic congestion. In this case, the traffic congestion score would tell us the opposite of what we expect it to tell us. This is why we changed the weight between two locations to the ratio of distance between two locations and the average speed of vehicles on that path. This gives us a fair and quantitative estimate of how congested a path is between any two given locations.
Taking the advice of our TA from the project proposal we have changed our original GUI interface plans, so as to enable the user to provide day, time and month inputs. Our module then creates the graph based on these user inputs and then subsequent computations follow as described in the above few sections.
Not a change, but another addition we have made is that now we have two visualisation for our project. Using the gmplot we get a visual representation of the actual path (lines) the user needs to follow in order to reach his or her destination in most optimised time. Using matplotlib we sequentially plot the geographical points of the locations in a given optimised path, and label each of the plotted points corresponding to their location.

Discussion section

Upon analysing certain specific outputs and inputs in our program we found that when we have a starting point and ending point which are neighbours in a graph, there may still exist a shorter path between based on the weight of the edges. This went on to reassure us that our project has achieved its initial goal of finding the shortest path between any two points based on traffic data. The traffic data is emulated as the ratio of length over speed so as to find the average time it takes to travel between the two specified locations. Our results told us that there may exist a shorter(less time consuming) path between two points even if they are directly connected. Thus our computational program achieved its goal.
As mentioned above the goal of our comprehensive project was to develop a program that takes the locations – one starting and one ending point (there may or may not be an intermediate location between the two paths) – and finds an optimized path between the two locations, based on the data set we have adopted for our project. Initially, we were somewhat apprehensive if we would be able to achieve the entirety of this goal. We initially developed a function to find an optimised path between two given locations using Dijkstra's algorithm. Once we were done with this initial task, we decided to increase the complexity and utility of our project. We then worked to develop a program that finds an optimised path between two given locations by travelling through the given points given by the user. This could be understood as follows: If an ambulance has to travel from location A to B, but the driver needs to pick up a patient from Location C in between, now our new function will return an optimised path between two locations, while going through the intermediate set of locations the user input (Location C in this case).
Any project doesn't come without hurdles, and ours was no exception. The initial dataset we obtained from data.world was humongous. It was over 600MB in size and rogue. This was making the running time of our computation extremely long and tedious, therefore we needed to clean this data, make it small and remove any unavailable entries from it.
Python is a dynamic language with numerous libraries, that offer some incredible features and functions. However, before being able to use a library to our need in the project, we needed to get comfortable working with the libraries itself. This was a little time consuming but a really fun learning experience overall. For visualising the path obtained from our computation, we wanted to use the osmnx python library. Downloading the library turned out to be tedious and complex, therefore we had to look into other alternative libraries for visualising our project.
Another drawback of our data set was that for a single location, there were multiple row entries and for every different row entry of the same location, there is a different longitude and latitude value. For instance, there are 3 different rows in the csv file for the location "Harlem". Each of these 3 different rows have different start and end longitudinal and latitudinal values. This discrepancy of difference in geographical points for same locations makes a same location look different and far off when visualising on the graph. This drawback has the potential to make our visualisation look clustered and confusing to the user.
The above mentioned drawback could have been avoided with the use of a Google API Key. We could have exact latitude and longitude locations for points using the geocoding feature of gmplot. Similarly if we would have had a API Key we could have displayed the gmplot data without "Development Purposes Only" and our visualization

would look much clearer.
Looking forward, we feel that our project has tremendous utility and potential. Currently our project was based only based on Chicago traffic data in year 2018. We can expand the project by going beyond Chicago, and making an ambulance optimised path finder for any state or country in the world. In addition to this we believe we could incorporate geocoding into our project. In geocoding we transform a description of a location,an address, or a name of a place to a location on the surface of earth. This can help the ambulance driver locate the address of its destination more easily.
In conclusion, we successfully created a program that uses the tkinter library to interactively take input from the user for the day of the week, the time of the day, the month and the starting and ending locations, and thereafter output an optimised path -both visually on the map of Chicago and as a list of locations- between the start and end points.

References

- Masser, Kylie. “The Solution to Ambulance Delays.” EngineerGirl, www.engineergirl.org/108047/The-Solution-to-Ambulance-Delays. Accessed 10 March, 2021
- Kaul, Rhythma. “More than 50 percent of Heart Attack Cases Reach Hospital Late, Govt Data Shows.” Hindustan Times, 10 Mar. 2017, bit.ly/2PZOc4q. Accessed 10 March, 2021
- “Choking Life: 12 on Way to Hospitals Die in Traffic Jams: Raipur News - Times of India.” The Times of India, TOI, 18 Oct. 2016, bit.ly/2OCH6m3.Accessed 8 March, 2021
- “20 Per Cent of Emergency Patient Deaths Blamed on Traffic Jam Delays.” Nation Thailand.com, 16 Jan. 2017, www.nationthailand.com/national/30304268.Accessed 8 March, 2021
- Gmplot. “Gmplot/Gmplot.” GitHub, github.com/gmplot/gmplot/wiki.
- “Software for Complex Networks.” Software for Complex Networks - NetworkX 2.5 Documentation, 22 Aug. 2020, networkx.org/documentation/stable/index.html.
- Priyam, Apurv. “Find and Plot Your Optimal Path Using Plotly and NetworkX in Python.” Medium, Towards Data Science, 25 Aug. 2020, towardsdatascience.com/find-and-plot-your-optimal-path-using-plotly-and-networkx-in-python-17e75387b873.
- City of Chicago, “Chicago Traffic Tracker - Historical Congestion Estimates by Segment - 2018-Current”, 2018, https://data.world/cityofchicago/sxs8-h27x , accessed 5 March, 2021.
- “Tkinter - Python Interface to Tcl/Tk.” Tkinter - Python Interface to Tcl/Tk - Python 3.9.2 Documentation, docs.python.org/3/library/tkinter.html.
- “Chicago Traffic Tracker - Historical Congestion Esti... 2018 - Dataset by Cityofchicago.” Data.world, 2018, data.world/cityofchicago/sxs8-h27x/workspace/file?filename=chicago-traffic-tracker-historical-congestion-estimates-by-segment-2018-current-1.csv.
- OpenStreetMap. https://www.openstreetmap.org/exportmap=5/51.500/-0.100
- “Dijkstra’s Algorithm.” Wikipedia, Wikimedia Foundation, en.wikipedia.org/wiki/Dijkstra