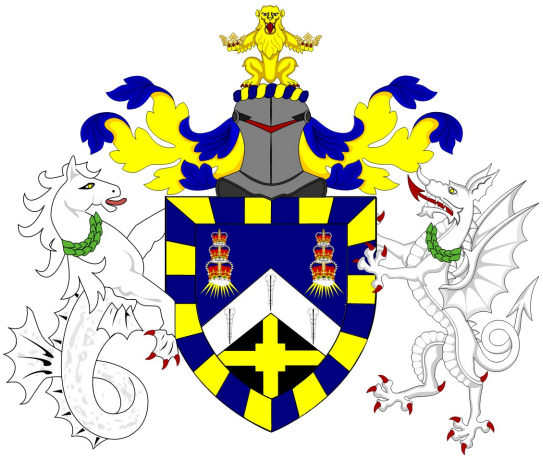
Data Analytics MSc Dissertation MTHM038, 2021/22

Time-dependent route planner for London

Jincy Baby, ID 210203097

Supervisor: Prof. Thomas Prellberg



A thesis presented for the degree of
Master of Science in *Data Analytics*

School of Mathematical Sciences
Queen Mary University of London

Declaration of original work

This declaration is made on September 3, 2022.

Student's Declaration: I, Jincy Baby, hereby declare that the work in this thesis is my original work. I have not copied from any other students' work, work of mine submitted elsewhere, or from any other sources except where due reference or acknowledgement is made explicitly in the text, nor has any part been written for me by another person.

Referenced text has been flagged by:

1. Using italic fonts, **and**
2. using quotation marks "...", **and**
3. explicitly mentioning the source in the text.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Thomas Prellberg, for his invaluable guidance and support. He has been extremely dedicated and patient throughout this period. This work would not exist without his help.

I want to thank my parents and my brother for their love and support during my entire life. I would not have been able to reach this stage without them. I owe thanks to a very special person, my husband, Mr. Jobin Babu George for being the pillar of my life. My heartfelt thanks to my mother-in-law who encouraged me on joining my current master's. My appreciation also goes out to my friends for their encouragement and support all through my studies.

Abstract

The project aimed to produce a list of point of interest that can be reached within a specified time from a specific location based on mode of travel. There are mainly four inputs - Location, point of interest, mode of travel and maximum travel time. The code filter the data according to the given constraints and produces a list of locations which can be selected again to get the shortest path from the initial location to the destination. An additional choice was included in the developed application where one can restrict the list of options by choosing a range depending on how close do you want it to be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation for this work

The world has been a witness to how humans have changed over time. There is no debate on the fact that our concepts on life change with each passing generation. Compared to our ancestors whose sole purpose was to survive in the wild, we are privileged. We have many more options to choose from, and our standards have been raised. The direct impact of these changes is that time has become more valuable. We seldom pause amid chasing our dreams, and in pursuit of a more perfect life. We may even feel proud of being so busy that we do not have time for relationships or sleep. According to the study conducted at Oxford as mentioned in [Pearson], the diaries recorded by people on their daily activities was analysed to compare the data of average use of time for men and women between 1961 and 2015 in UK. It revealed that major part of a standard weekday in a person's life is spend on paid or unpaid work. This also focus on how busy a person is even out of their work hours. When compared to 1961, men got under 50 minutes more free time in 2015. Women reduced unpaid work and increased paid work to end up with no to little change overall. We need to realise that focusing our attention on ourselves for a few hours can feel liberating and even productive. As such, the limited time we can take out of our schedule would be precious and should be utilised efficiently.

Consider that you met with a few friends and planned to hang out in a restaurant. Deciding on a place can be time-consuming as we tend to debate the options, and the more people there are, the more options there will be. Finally, when you decide on a restaurant, there is a chance that the place is far from your location and you have an urgent meeting within two hours. Then, you will again start to discuss other options and this will inevitably waste your precious time. What if there was an application which could find a list of restaurants that

can be reached within 5 minutes from your location? Furthermore, what if this application would show you the shortest path to reach the destination that you choose from the list? The time spent discussing the destination will be reduced tremendously, and as an additional benefit, you will get the shortest route to the destination within the application itself.

1.2 Previous works

Let us consider a few available applications that facilitate locating the meeting points or places to visit. *Google Maps* can find the nearest places from your current locations. *whatshalfway* [WH] and *Meetways* [MW] are platforms that find the halfway point between two different locations which can also be found in [Aeden]. While *whatshalfway* has an option to search for places of interest around the meeting point, it redirects to Google Maps for finding the nearest places and *Meetways* displays a list of user-defined spots of interest. Other available applications perform much the same as the ones mentioned. With the exception of Google Maps, none of these applications display the shortest route to the chosen destination and it is not possible to filter locations based on the input travel time in all the cases including Google Maps.

1.3 Problem Description

In contrast to other applications, we focus on cases where you plan to search for a place near your location with the additional constraint that the place should be reachable within a fixed time depending on the mode of travel. The advantage is that it will filter the type of place that you want to visit based on the input conditions. It will only give you locations based on the constraint and you just need to select one that you prefer to get the shortest route to reach the chosen destination.

This project intends to develop an interface that allows users to enter their current location, the type of place they wish to visit, and the maximum time required to reach the destination. The places that are reachable within the input time will be displayed on the screen. User can click the name to get the shortest route from the original location to the selected location.

1.4 Thesis Overview

The second chapter discusses the project's mathematical and theoretical background. We will examine the basic concepts in graph theory and how they relate to modelling geographical

maps in chapter 3. A detailed overview of the libraries and tools used in the development environment is presented in chapter 3. Chapter 4 examines the project implementation process, focusing on the algorithms used to filter the locations and to find the shortest path to the destination. The results obtained are discussed in Chapter 5, along with the time complexity of the algorithm. Appendix A contains a detailed overview of the project planning and designing.

Chapter 2

Background

2.1 Graph Theory

Graph theory is concerned with the study of graphs, and its origins can be traced back to 1736, when Euler considered the Königsberg bridge problem. Graphs can be used to model many real-world situations, such as cities in a country and the roads that connect them. In fact, graph theory is widely used in spatial data analysis and its network applications. Maps can be modelled using graph theory and the graph algorithms can be applied to solve most of the related problems.

2.1.1 Graph

Definition 2.1.1. A graph G consists of a set V of vertices (or nodes) and a set E of edges such that each edge is associated with an unordered pair of vertices. Graph G is usually written as $G = (V, E)$ and uv denotes an edge between two vertices u and v [Deo].

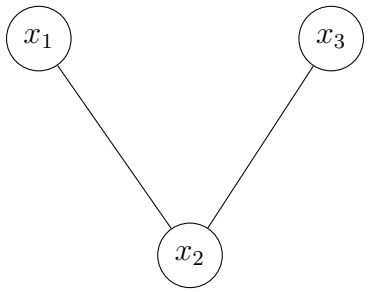


Figure 2.1: Graph

Definition 2.1.2. A weighted graph is a graph which has a number (called weight) associated with each of its edges. If the edge e is labelled with the number k, we say that the weight of the edge is k [Berkeley].

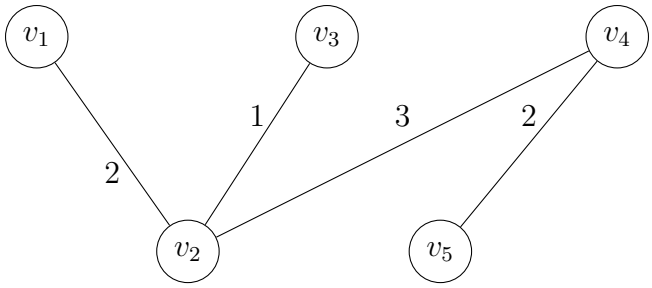


Figure 2.2: Weighted Graph

The graph in Figure 2.2 is a weighted graph where the weight of v_2v_3 is 1, v_2v_4 is 3 and that of v_1v_2 and v_4v_5 is 2.

Definition 2.1.3. A directed graph or digraph is a graph where each edge is associated with an ordered pair of vertices. The edge e from u to v is denoted by (u, v). In a digraph, the in-degree of a vertex is the number of edges leading into that vertex and out-degree of a vertex is the number of edges going out from the vertex [SW].

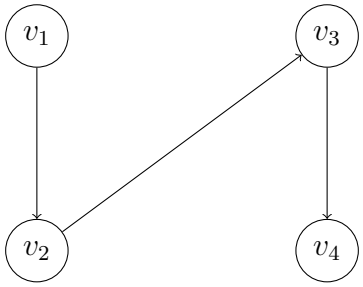


Figure 2.3: Directed Graph or Digraph

The in-degree and out-degree of the vertices for the graph in 2.3 is given in the following table.

node	in-degree	out-degree
v_1	0	1
v_2	1	1
v_3	1	1
v_4	1	0

Table 2.1: In-degree and Out-degree

Definition 2.1.4. A source vertex in a digraph is defined as the vertex with indegree 0 and sink vertex is that with outdegree 0 [MIT].

From the figure 2.3, we found that the in-degree of v_1 is 0. This implies that v_1 is the source vertex. Similarly, since out-degree of v_4 is 0, we can say that v_4 is the sink vertex.

Definition 2.1.5. For a digraph G = (V,E), a path of length n from u_0 to u_n is a sequence of vertices and edges $u_0, e_1, u_1, e_2, ..., u_n$ such that e_i is the edge connecting u_{i-1} and u_i for all $i \in \{1, 2, ..., n\}$ [Berkeley].

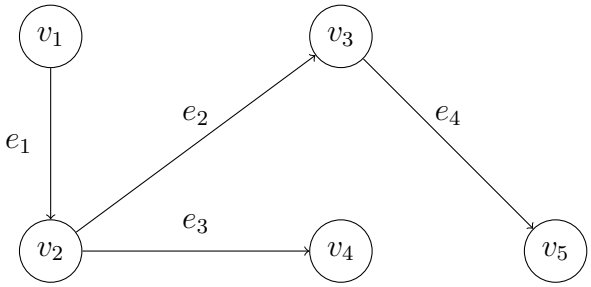


Figure 2.4: Digraph

In Figure 2.4, $v_1, e_1, v_2, e_2, v_3, e_4, v_5$ is a path of length 3. We can also represent this path using the notation $\langle v_1, v_2, v_3, v_5 \rangle$.

Definition 2.1.6. Path length in a weighted graph is the sum of edge weights along the path [Ralphs].

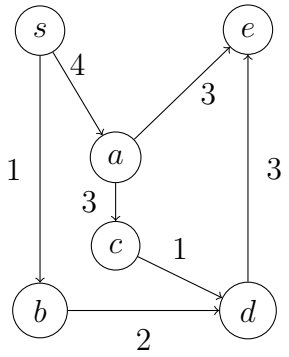


Figure 2.5: Digraph - Example for path length

Let $e_1 = (s, a)$, $e_2 = (a, c)$, $e_3 = (c, d)$, $e_4 = (d, e)$. i.e, $w(e_1) = 4$, $w(e_2) = 3$, $w(e_3) = 1$, $w(e_4) = 3$

Then, the weight of the path $\langle s, a, c, d, e \rangle$ is given by

$$\begin{aligned} w(e_1) + w(e_2) + w(e_3) + w(e_4) &= 4 + 3 + 1 + 3 \\ &= 11 \end{aligned}$$

2.2 Shortest Path Problems

Consider a graph $G = (V, E)$ such that each edge e of G is associated with a weight $w(e)$. There are many variants of shortest path problem such as Single-Source Shortest Path (SSSP) problem, All-Pairs Shortest Path (APSP) problem, and Single-Pair Shortest Path (SPSP) Problem [CA]. In Single-Source Shortest Path problem, the objective is to find the shortest path for a source vertex s to all other nodes in the graph. APSP problem deals with finding the shortest path between all pairs of vertices in the graph.

The aim of SPSP problem is to find a shortest weighted path from a source vertex s to a sink vertex t. In such case, we need to find the path of minimum weight connecting s and t. The minimum weight of a path from s to t is often referred to as the distance between s and t and is denoted by $d(s, t)$. If there is no edge between the vertices u and v, *i.e.*, $uv \notin E$, we take the weight of the edge as $w(uv) = \infty$.

2.3 Common algorithms for finding shortest path

Algorithms that are commonly used to solve SSSP problem are Breadth-First Search Problem (in case of unweighted graphs), Dijkstra algorithm, and Bellman-Ford algorithm. Floyd-Warshall algorithm is usually used to solve APSP. We may use Dijkstra's for solving ASSP by running it for every vertex. However. Dijkstra's was made to compute the shortest path from a single-source whereas Floyd-Warshall originally finds optimal distance between each pair of vertices . The time complexity of both Floyd Warshall and Dijkstra's in solving APSP is $\mathcal{O}(|V|^3)$ [WPF], where $|V|$ is the number of vertices in the graph. The main advantage of Floyd-Warshall is that it is suitable for negative edges unlike Dijkstra's but it does not work for single-source problems.

Dijkstra's algorithm is mostly used to solve SPSP problems since it only has to find the path between a pair of nodes, the time complexity is almost linear. Dijkstra's algorithm is optimal and it is applied in case of weighted graphs. The assumption that we make for Dijkstra's is that the edge weights are non-negative. A* algorithm is known for its highest possible efficiency to solve SPSP problems. A* algorithm is similar to Dijkstra's with the difference that it uses a heuristic function to choose the shortest path. Limitation is that the edge weights should be non-negative as in the case of Dijkstra's algorithm. Since our problem relates to Single-Pair Shortest Path Problem, the common algorithms that can be utilised are Dijkstra's algorithm, and A* algorithm.

2.3.1 Dijkstra's algorithm

Dijkstra's algorithm is said to be a greedy algorithm as it finds a path with minimum weight by making a locally optimal choice at each step of the algorithm. It is a single-source shortest path algorithm. However, its also applicable for SPSP problems. Furthermore, it works for both directed and undirected graphs where all the edges have positive weights. The algorithm has wide range of application in domains that entail network modelling. Notations and structure in the given algorithm was modified to make it easier to understand.

Algorithm 1: Dijkstra's algorithm - Modified

Data: graph $G = (V, E, w)$

Result: find length of the shortest path from source vertex s to sink vertex t

Mark current distance $L(s) = 0$ and $L(u) = \infty, \forall u \in V - \{s\}$;
Initialize Z as the set of all nodes and $S = [\,]$;
while $Z \neq [\,]$ **do**
    Choose node x such that $x \in Z$ and minimum $L(x)$;
    $Z = Z - x$;
    $S = S \cup \{x\}$;
    foreach $u \in V$ *adjacent to* x **do**
        $L(u) = \min\{L(u), L(x) + w(x, u)\}$;
        $w(x, u)$ is the weight of the edge from x to u
return $L(t)$

Implementation of Dijkstra's Algorithm

Assume that a road network is represented by the graph given in 2.6 and the weight of the edges represent the distance in km. We need to find shortest path from node x_1 to node x_6.

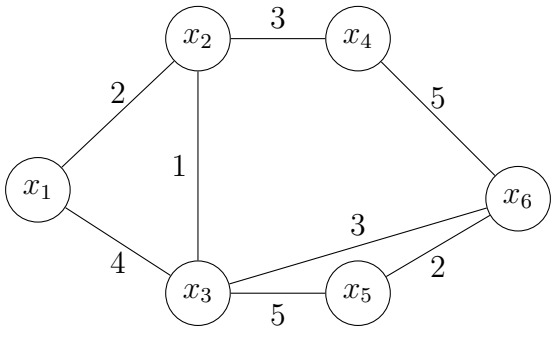


Figure 2.6: Weighted Graph

Step1: Set $L(x_1)$=0 and rest as ∞ ; $Z = x_1, x_2, ..., x_6$

Step2: Identify the vertex with minimum distance from Z. x_1 is the vertex with lowest distance. Remove x_1 from Z.
Step3: Check the vertex which is adjacent to x_1 and change the corresponding distance.

$$L(x_2) = min\{L(x_2), L(x_1) + w(x_1, x_2)\} = min\{\infty, 0 + 2\} = 2$$
$$L(x_3) = min\{L(x_3), L(x_1) + w(x_1, x_3)\} = min\{\infty, 0 + 4\} = 4$$

Step4: Among the unvisited nodes, x_2 has the lowest distance.

In the first subfigure of 2.7, the visited vertex x_1 is denoted by red and the distance value of neighbour vertices is given in blue. The vertex with lowest distance is highlighted with green in the second subfigure.

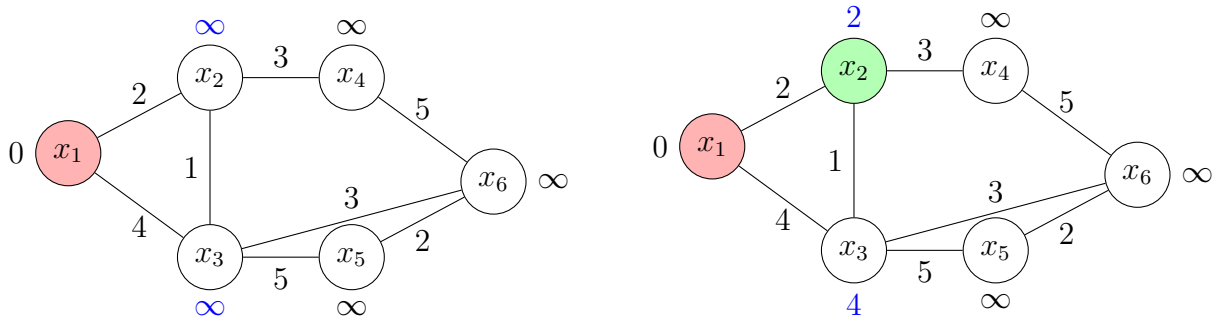


Figure 2.7: Implementation of Dijikstra's algorithm (1)

Step4: Repeat steps 3 and 4 to obtain the result. The adjacent vertices to x_2 are x_3 and x_4.

$$L(x_3) = min\{L(x_3), L(x_2) + w(x_2, x_3)\} = min\{4, 2 + 1\} = 3$$
$$L(x_4) = min\{L(x_4), L(x_2) + w(x_2, x_4)\} = min\{\infty, 2 + 3\} = 5$$

Now, x_3 is the unvisited vertex with minimum distance.

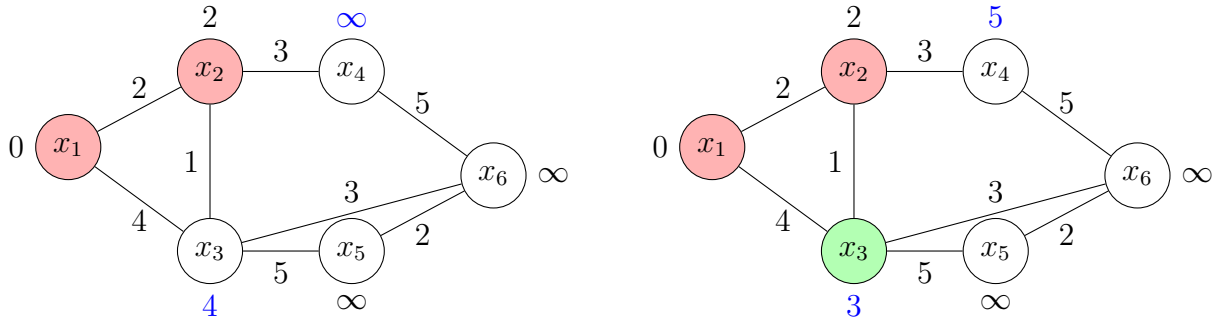


Figure 2.8: Implementation of Dijikstra's algorithm (2)

Similarly, we take the adjacent nodes of x_3, ie., x_5 and x_6.

$$L(x_5) = min\{L(x_5), L(x_3) + w(x_3, x_5)\} = min\{\infty, 3 + 5\} = 8$$
$$L(x_6) = min\{L(x_6), L(x_3) + w(x_3, x_6)\} = min\{\infty, 3 + 3\} = 6$$

Then, the vertex with lowest distance is x_5. However, $L(x_6)$ remains the same since $L(x_5) + w(x_5, x_6) > L(x_6)$. $L(x_6) = min\{L(x_6), L(x_5) + w(x_5, x_6) > L(x_6)\} = min\{6, 5 + 5\} = 10$

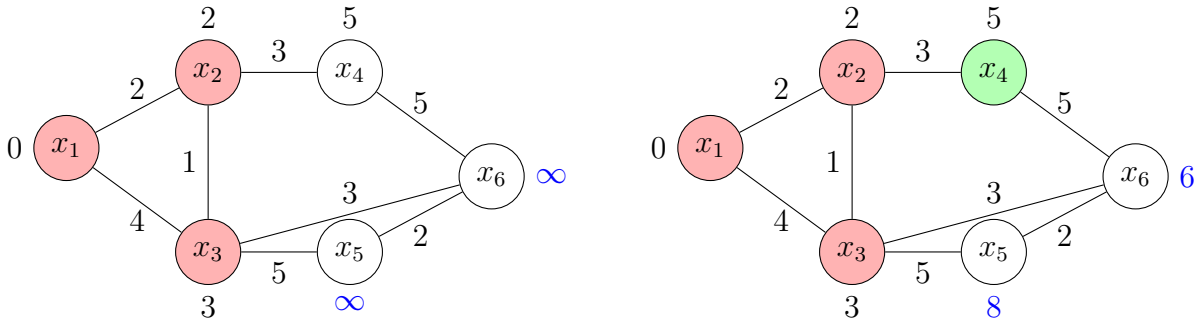


Figure 2.9: Implementation of Dijikstra's algorithm (3)

The minimum distance to reach x_6 is 6 and the path corresponds to the edges whose weights added upto obtain the distance of x_6. Thus, the shortest path from x_1 to x_6 is $\langle x_1, x_2, x_3, x_6 \rangle$.

The figure 2.10 shows the required shortest path where the traversed nodes are represented by red and the corresponding edge is shown using the gray line.

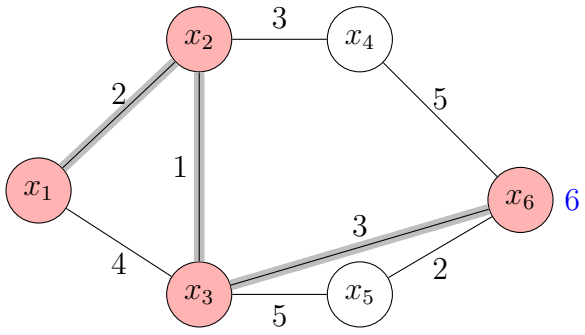


Figure 2.10: Shortest path for the Weighted Graph

2.3.2 A* algorithm

A* algorithm has its foundation derived from the principles of Dijkstra's algorithm and proves to be faster compared to Dijkstra's as it decides on the next node to move by considering a heuristic element. There are many functions that can be chosen as the heuristic function

to be used in A* algorithm. For instance, Euclidean distance or Manhattan distance can be considered when working with graphs with distance as edge weights.

The algorithm tries to find a path with least cost from a start node s to a target node t. To find this, it uses the formula:

$$f(u) = g(u) + h(u)$$

where $f(u)$ denotes the total cost of the node, *i.e.*, it refers to the current cost of shortest path through u to t, $g(u)$ is the distance from s to t and $h(u)$ is a heuristic function which is the estimated distance from u to t [SAL].

Algorithm 2: A* algorithm [Isaac] - Modified

Data: Graph $G = (V, E, w)$ with start node s and end node t
Result: find shortest path from s to t

```
Initialize Z = { } for unvisited nodes and S = { } for visited nodes ;
foreach n ∈ V do
    Add node to Z with g-score & f-score ∞ and previous node as NULL ;
    Z[n] = [g_score, f_score, Previous] = [∞, ∞, NULL]
Set Z[s] = [0, h_score, NULL] where h_score = heuristic(s);
Set finished to False ;
while finished is False do
    Take node in Z with lowest f-score as current node C ;
    if C = t then
        finished=True ;
        S[C] = Z[C] ;
    else
        foreach u, neighbour of C do
            if u ∉ S then
                new g-score = w(C,u) + Z[C][g-score] ;
                if new g-score < Z[u][g-score] then
                    Z[u][g-score] = new g-score ;
                    Z[u][f-score] = new g-score + h-score(u) ;
                    Z[u][Previous] = C
        S[C] = Z[C] ;
        Remove C from Z
Return S
```

The algorithm in [Isaac], explained verbally and in pseudocodes, was modified to make it easier to comprehend. Variables were introduced to denote f-score, g-score and h-score. There are three scores stored in Z and in S. Z[u][g-score] denotes g-score of u from Z, Z[u][f-score] denotes f-score of u from Z and Z[u][Previous] is the previous node from which we find the new g-score and f-score of the mentioned node.

Assume that the values of u stored in Z is [1, 2, s]. This implies,

$$Z[u][g-score] = 1$$
$$Z[u][f-score] = 2$$
$$Z[u][Previous] = s$$

Further, $w(C, u)$ is the weight of the edge connecting the current node, denoted by C and the neighbour node u.

Selecting heuristic function

It is vital to select the heuristic function cautiously as the function should be relevant to the edge weight. Furthermore, the function should be chosen such that it does not exceed the actual cost.

Implementation of A* Algorithm

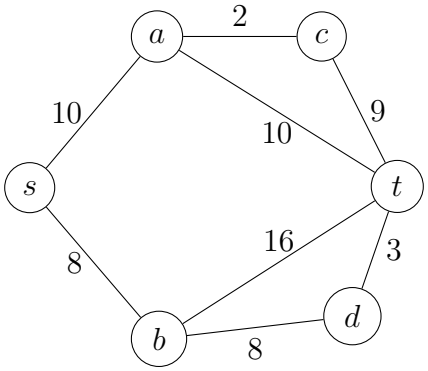


Figure 2.11: Weighted Graph

node	Heuristic distance to t
s	10
a	15
b	5
c	5
d	10
t	0

Table 2.2: Estimated Cost to target node

The objective is to apply A* search algorithm to find shortest path from s to t in the graph 2.11.

Assume that the heuristic function returns the values in Table 2.2 as the estimated cost.

The estimated cost to target node is represented by blue text near the corresponding nodes in 2.12.

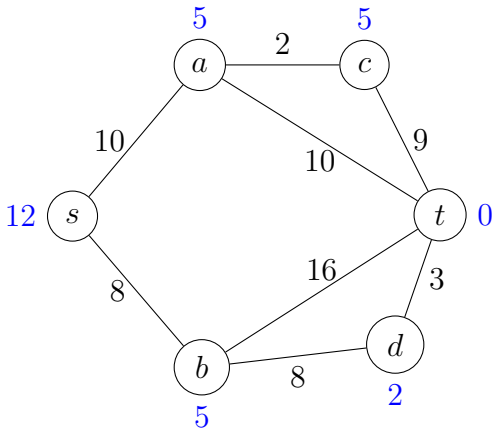


Figure 2.12: Weighted Graph with heuristic distance

Step 1: Initialise Unvisited list Z and Visted list S with node, g-score, f-score and previous.

For the start node s,

$$g - score = 0$$
$$f - score = h(s) = 12$$
$$previous = NULL$$

Z is constructed with f-score = ∞, g-score = ∞ and previous = NULL for all vertices. Then, the g-score, f-score and previous for the starting node s is updated as calculated above. S is the list of visited nodes along with its f-score, g-score and previous. Since there are no visited nodes initially, S is empty.

(a) Unvisited List, Z

node	g-score	f-score	previous
s	0	12	NULL
a	∞	∞	NULL
b	∞	∞	NULL
c	∞	∞	NULL
d	∞	∞	NULL
t	∞	∞	NULL

(b) Visited List, S

node	g-score	f-score	previous

Table 2.3: Unvisited and Visited Lists (1)

Step 2: Select a node from Z such that the node has minimum f-score. s is the node that satisfies this condition.

For the unvisited neighbors of s, we need to update the values of f-score, g-score and previous in Z. The unvisited neighbors are a and b as shown in 2.13.

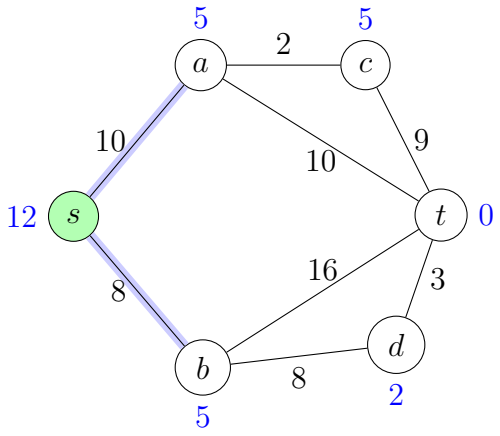


Figure 2.13: Implementation of A* algorithm (1)

Step 3: For the unvisited neighbors, updates its values in Z.

We have,

$$\begin{aligned} w(s,a) &= 10 \\ h(a) &= 5 \\ Z[a][g-score] &= \infty \\ \implies new \quad g-score &= w(s,a) + Z[s][g-score] \\ &= 10+0 \\ &= 10 \end{aligned}$$

Since new g-score < Z[a][g-score], find values of g-score, f-score & previous for a in Z.

$$\begin{aligned} Z[a][g-score] &= new \quad g-score = 10 \\ Z[a][f-score] &= new \quad g-score + h(a) = 10+5 = 15 \\ Z[a][Previous] &= s \end{aligned}$$

Similarly, for b, $new \quad g-score = w(s,b) + Z[s][g-score] = 8+0 = 8$. Since $Z[b][g-score]$ is greater than the obtained value,

$$\begin{aligned} Z[b][g-score] &= 8 \\ Z[b][f-score] &= 8+5 = 13 \\ Z[b][Previous] &= s \end{aligned}$$

Now, update the g-score, f-score and Previous for a, b and c in Z. Remove s from Z and store it in S such that $S[s] = Z[s]$.

(a) Unvisited List, Z

node	g-score	f-score	previous
a	10	15	s
b	8	13	s
c	∞	∞	NULL
d	∞	∞	NULL
t	∞	∞	NULL

(b) Visited List, S

node	g-score	f-score	previous
s	0	10	NULL

Table 2.4: Unvisited and Visited Lists (2)

Step 4: Repeat step 2 and 3 until we enter the target node into S.

b has the lowest f-score in Z. Thus, take b as the current node. The adjacent vertices are d and t.

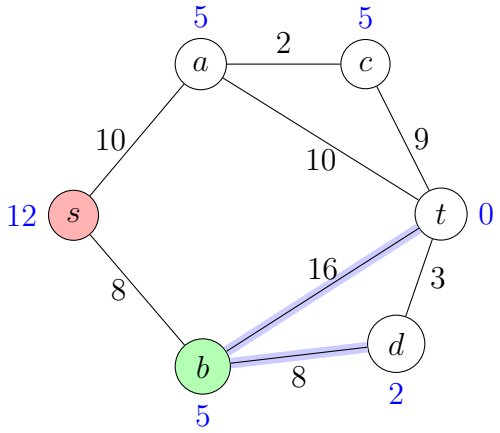


Figure 2.14: Implementation of A* algorithm (2)

For d, new g-score $= w(b, d) + Z[b][g - score] = 8 + 8 = 16$ and

For t, new g-score is $w(b, t) + Z[b][g - score] = 16 + 8 = 24$.

(a) Unvisited List, Z

node	g-score	f-score	previous
a	10	15	s
c	∞	∞	NULL
d	16	18	b
t	24	24	b

(b) Visited List, S

node	g-score	f-score	previous
s	0	10	NULL
b	8	13	s

Table 2.5: Unvisited and Visited Lists (3)

Here, the unvisited vertex with lowest f-score is a and its adjacent nodes are c and t

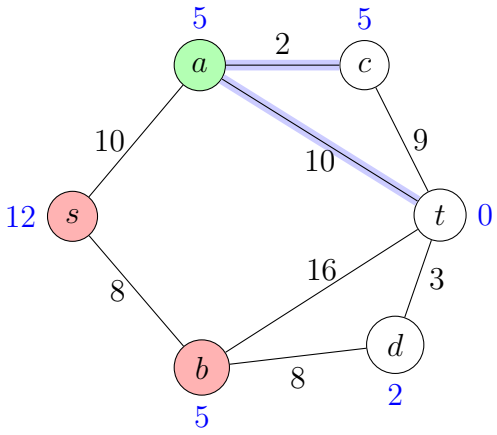


Figure 2.15: Implementation of A* algorithm (3)

Calculating the new g-score for c, we get $= w(a,c) + Z[a][g-score] = 2+10 = 12$ and For t, new g-score is $w(a,t) + Z[a][g-score] = 10+10 = 20$. Update the g-score, f-score and Previous in Z.

(a) Unvisited List, Z

node	g-score	f-score	previous
c	12	17	a
d	16	18	b
t	20	20	a

(b) Visited List, S

node	g-score	f-score	previous
s	0	10	NULL
b	8	13	s
a	10	15	s

Table 2.6: Unvisited and Visited Lists (4)

c has the lowest f-score and its unvisited neighbour is t. new g-score is $w(c,t) + Z[c][g-score] = 9+12 = 21$. Since this value is greater than the g-score of t in Z, there is no change.

(a) Unvisited List, Z

node	g-score	f-score	previous
d	16	18	b
t	20	20	a

(b) Visited List, S

node	g-score	f-score	previous
s	0	10	NULL
b	8	13	s
a	10	15	s
c	12	17	a

Table 2.7: Unvisited and Visited Lists (5)

Now, d has the lowest f-score and its unvisited neighbour is t.

$$w(d,t) + Z[d][g-score] = 19 < Z[t][g-score]$$

Hence,

$$Z[t][g-score] = 19$$
$$Z[t][f-score] = 19$$
$$Z[t][Previous] = d$$

Update Z and S.

(a) Unvisited List, Z

node	g-score	f-score	previous
t	19	19	d

(b) Visited List, S

node	g-score	f-score	previous
s	0	10	NULL
b	8	13	s
a	10	15	s
c	12	17	a
d	16	18	b

Table 2.8: Unvisited and Visited Lists (6)

Now, t is selected and the final visited list is as follows.

node	g-score	f-score	previous
s	0	10	NULL
b	8	13	s
a	10	15	s
c	12	17	a
d	16	18	b
t	19	19	d

Table 2.9: Final Unvisited List

The previous vertex of t is d and that of c is s. Hence, we get the path $\langle s, b, d, t\rangle$.

2.4 Time Complexity of the algorithms

Dijkstra's algorithm can be implemented in various ways where graph $G = (V, E)$ can be represented using adjacent matrix or adjacent list. Assume $|V|$ denotes the number of vertices and $|E|$, the number of edges. In the case that G is represented as an adjacent matrix, the time complexity can be $\mathcal{O}(|V|^2)$ where the priority queue used in the algorithm is an unordered list. However, this can be reduced to $\mathcal{O}(|V| + |E|log|V|)$ [TC] while setting the priority queue as a binary heap or Fibonacci heap when G is expressed as an adjacent list.

The figure 2.16 shows the time complexity of different implementation of Dijkstra's algorithm based on increase in number of nodes.

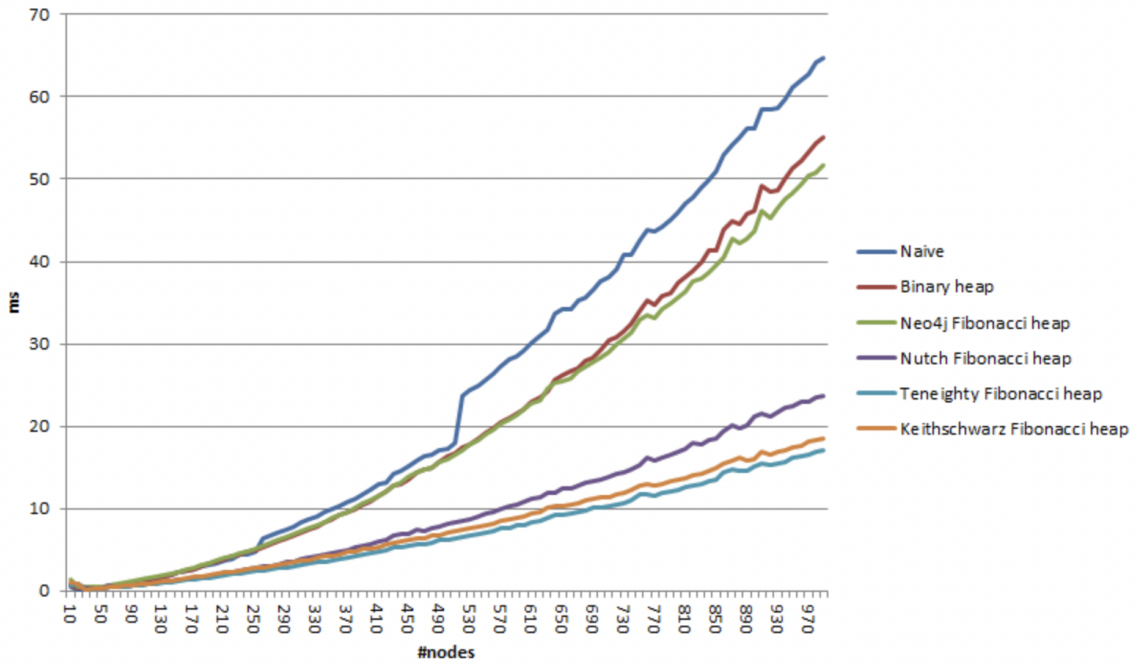


Figure 2.16: Time complexity for the Dijkstra's algorithms [TC]

Considering the A* algorithm, the time complexity depends on the heuristic function that we considers as it is derived from Dijkstra's with the addition that we introduce a heuristic function to decide on the next node to be visited. However, since we introduce the heuristic function, A* is considered to be relatively faster than Dijkstra's algorithm.

2.5 Spatial Analysis

Spatial data or Geospatial data is defined to be the data with positional values and Geoinformation is any information interpreted from the spatial data. We may use a computer-based system to handle the geographical data and this system is descibed as Geographic Information System(GIS). GIS provides four functions. that is Data Preparation, Data Management, Data Manipulation and Analysis and Data Presentation.

Spatial analysis refers to statistical analysis of geographical data and it uses various techniques by integrating both statistics and GIS [HR]. We are concerned with shortest path analysis on spatial data in our project and thus we apply GIS techniques along with shortest

path algorithms to get the required path for the selected location. One of the tool that we use in the project is Geofencing, which is a significant feature that utilizes GPS (Global Positioning System) in order to define geographical boundaries.

2.5.1 APIs for geomapping

API or Application Programming interface is a collection of data structures, classes or functions available for programmers [HD]. Map APIs let a programmer focus on the main tasks and avoid writing the low-level codes required to add a new layer or to display an interactive map [OWM]. This greatly reduce the effort and complexity of creating the codes. Some of the map APIs are Google Maps API, OpenLayers and Overpass API.

The data used in the project is from OpenStreetMap. OpenStreetMap (OSM) is an open source map and the data model has three main components which are nodes, ways and relations. Nodes are points on the map. way is an ordered list of nodes and it represents a street or outline of a house. Relations are used to model geographic relation between objects [JN].

Figure 2.17: Node in OpenStreetMap [JN]

The three components of the data model have tags which describe certain features. In figure 2.17, the value for key "amenity" in tag is bench. This shows that the mentioned node is a bench and based on the coordinates, the bench is in London.

In our project, we use Overpass API to download data from OpenStreetMap. Overpass API uses a query language to access the data and one of the data mining tool that uses Overpass API query is Overpass Turbo which can show the result on an interactive map. There are

different filters that we can use to query. In our case, we need to query nodes based on the tag where the key is amenity or tourism. Some of the filtering methods are:

1. ***Using bounding box filter***
   Example:
   (
   node[“amenity” ∼ “restaurant”](bbox);
   );
   out center;

2. ***Manually specifying the bounding box***
   Example:
   (
   node[“tourism” ∼ “museum”](51.29,-0.35,51.41,-0.11);
   );
   out center;

3. ***Specifying area name***
   Example:
   area[name=“Cardiff”]; (
   node[“tourism” ∼ “gallery”](area);
   );
   out center;

4. ***querying for more than one tags***
   Example:
   area[name=“London”] →.LN;
   (
   node[“tourism” ∼ “museum | gallery”](area.LN);
   );
   out center;

2.5.2 Graph representation of geographic maps

The map of a city that we get from Google maps API or Openstreet API can be viewed as a large graph with numerous nodes and edges representing roads connecting them. We

can convert a road network into a graph or a weighted graph where the edges represents roads and the nodes represent the junction points. The weights associated with each edge can be either the length of the road or travel time. The edges may have more than one weights assigned to it and the weights may be distance between the nodes or time taken to travel from one node to another or even the cost of transport. These type of graphs have application in travel planning search engines.

Consider the route map displayed in Figure 2.18. We can model the map by changing the junction between roads as nodes and the each of the roads between two junctions as edges connecting the corresponding nodes. We may even assign a weight to the graph taking into account the type of problem that we are dealing with.

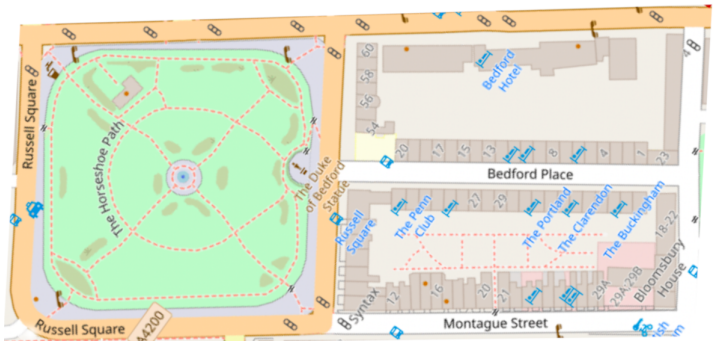


Figure 2.18: Route Map

If we represent roads as edges and the junction points as nodes,then we get a graph representation of the node network. The obtained graph, in Figure 2.19, can be written as $G = (V, E)$ where $V = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ and $E = \{x_1x_2, x_1x_5, x_2x_3, x_3x_4, x_3x_8, x_4x_5, x_5x_6, x_6x_7, x_7x_8\}$.

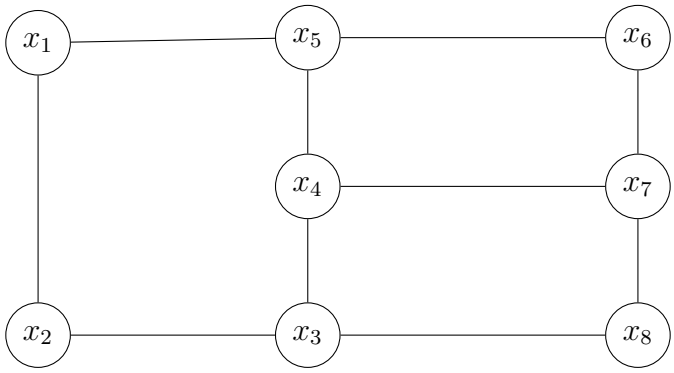


Figure 2.19: Graph Representation of Route Map

Considering the distance (in metres) between each junctions as the weight between the corresponding nodes, we get a weighted graph as shown in Figure 2.20. The edge x_1x_2 has

a weight 177.5, which implies that the road joining the corresponding junction has a length of 177.5m. We can also convert the distance into km then the weight of the edge x_1x_2 will be 0.1775.

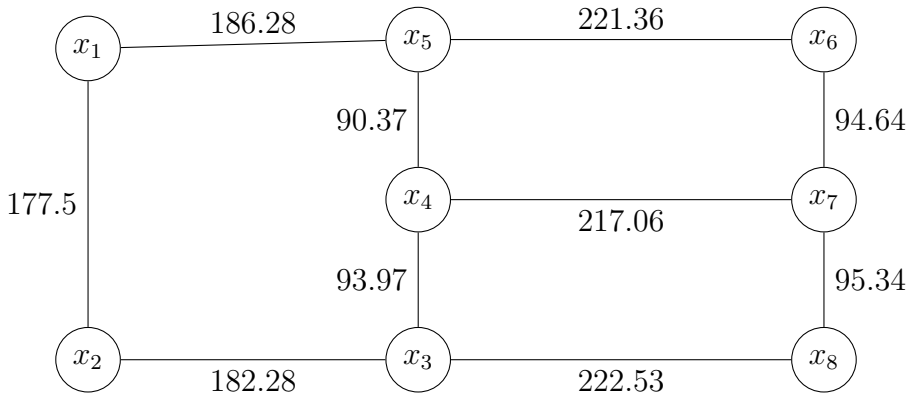


Figure 2.20: Weighted Graph for Route Map

Similarly, travel time can be considered as the edge weights. Let us we assume that a person is walking from the junction denoted by x_1 to that denoted by x_2. If we assume the average walking speed, 5 km per hour, as the speed of the individual, then time is given by

$$t = \frac{0.1775}{5} = 0.0355hrs$$

.

Thus, the weight of x_1x_2 can be taken as 0.0355 in the case where we assume the weight as the travel time and the mode of travel as walking.

Lets say we want to find the shortest path between two towns. Then, the network model can be modelled as mentioned above. Thus, we get a graph with a number of nodes and edges with weights assigned as per our requirement. Further, we can find the shortest path using an appropriate shortest path algorithm.

2.5.3 Universal Tranverse Mercator

UTM or Universal Tranverse Mercator is a plane coordinate grid sytem which divides the world into 60 zones each 6-degrees of longitude in width [USGS]. Both UTM and latitude/longitude are considered to be of equal importance and have preference based on the requirement of the user.

2.5.4 Geofencing

Geofence refers to a virtual boundary surrounding a geographic area. In geofencing technology, the current location of the user is considered to set a boundary around a certain area of interest depending on the requirement [Sale]. For example, a local shop can analyse the data of potential customers within a specific geographic radius from the shop in order to design best marketing plans to attract those customers.

In order to create a simple geofence in Python, we can use the package shapely. The function 'Point' is used to define a point in the given latitude/longitude coordinate and the radius can de defined using buffer [C7].

```
from shapely.geometry import Point
geofence = Point(51.590199, -0.017400).buffer(0.5)
geofence.contains(Point(51.51, -0.017))
```

If the geofence contains the defined point, the output is True. Otherwise, the output is False. In the specified case, output is True as the point Point(51.51, -0.017) is inside the geofence.

Chapter 3

Development Environment

Programming language that is used for our project is Python 3. Python works well with geospatial data since it provides access to many modules with functions for geocoding, plotting maps and so on. The most commonly used python packages for spatial analysis are Geopandas, osmnx and networkx.

3.1 Python packages used for the project

3.1.1 Geopandas

Geopandas is an extended version of Pandas library and the two main data structures that are used in it are Geoseries and Geodataframe. Geopandas create an additional column 'geometry' which is a geoseries that include geometries, like point or polygon, represented as shapely objects [C2]. Since we are dealing with geospatial data, Geopandas is the perfect choice for our project since it makes it simpler for Python programmers to handle spatial data.

3.1.2 geopy

geopy can be utilised to find the coordinates of locations across the globe [C3]. The name of the current location, such as "Stratford, London," serves as the input for our project and we need to obtain the latitude and longitude of the location for the further processing.

3.1.3 osmnx

OSMnx is a Python package to retrieve, model, project, analyze, and visualize street networks and other spatial geometries from OpenStreetMap [C6]. The module that we specially use

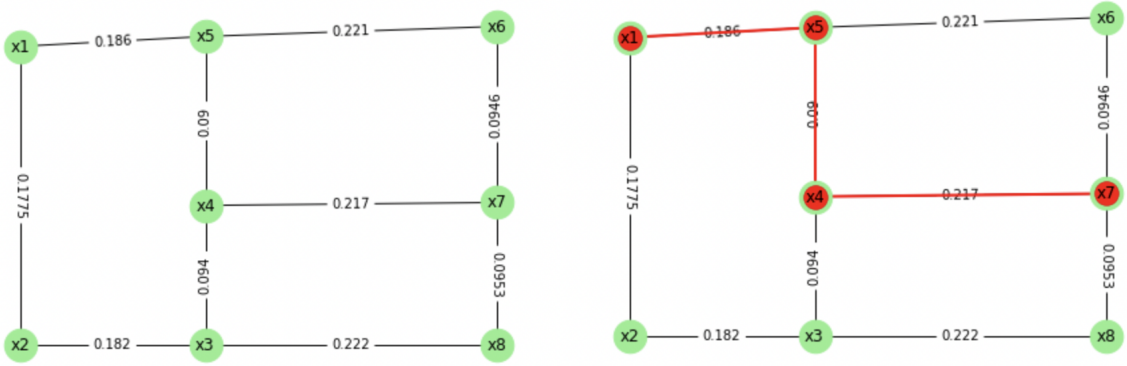
in the project is 'graph' from this package. This module contains the functions to construct the graph models of the street maps depending on the parameter and type of graph that we need for our project. For instance, if we have an address of a location and we need a graph of the road networks within a certain distance of that address, we use graph_from_address function. We also use appropriate functions from this package to add weights to the graph and to project the graph which will be discussed further in the next chapter.

3.1.4 networkx

This package aids in creation and manipulation of complex networks. We use networkx to find the shortest path from our starting location and other data points in the initially filtered data [C1]. Networkx includes shortest path algorithms like Dikstra's , Bellman Ford, A* algorithm and so on. We use shortest_path function where the default method is 'dijkstra'. Apart from this dijkstra_path function and astar_path can also be used as alternative for this function.

Application of the algorithms using networkx

The graph that we created as a graphical model of the road network in section 2.5.2 is given in figure 3.1a. The weight of the edges is given by the distance between the nodes and is calculated in km. The shortest path from x1 to x7 was found using the networkx module and the resultant path is shown in 3.1b.



(a) Weighted Graph Model

(b) Shortest Path from x1 to x7

Figure 3.1: Finding shortest path in Python

In order to find the shortest path from x1 to x7, both Dijkstra's and A* algorithm from networkx was used. The code for the construction of the graph, finding the shortest path and plotting the resultant graph is given in Appendix B.

The execution time that was obtained from running the algorithm in the same conditions for the same graph was noted. The below table gives the execution time taken (in seconds) by both the algorithms at different run times.

index	Dijktra's algorithm	A* algorithm
1	00.002895	00.000910
2	00.001308	00.000196
3	00.000246	00.000248
4	00.000204	00.000198
5	00.000231	00.000202
6	00.001833	00.000412

Table 3.1: Executing time taken of Dijkstra's and A* algorithms

The A* algorithm showed higher performance than Dijkstra's algorithm in the three of the cases and the execution times were similar in the other three.

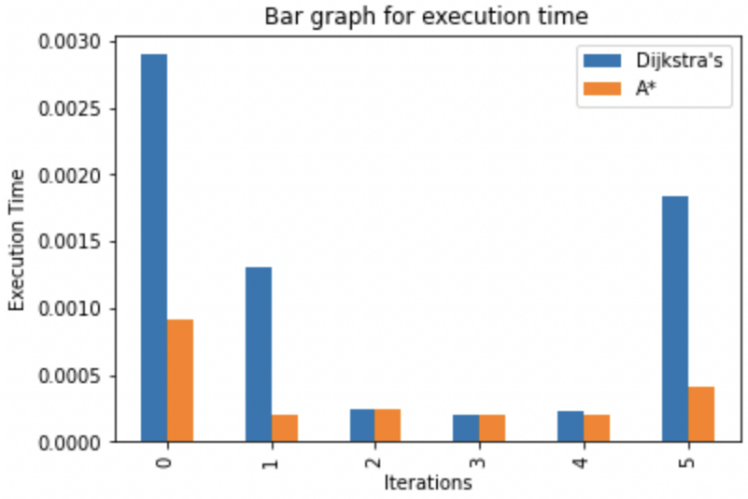


Figure 3.2: Comparison for execution time of the algorithms

3.1.5 Other packages

Apart from this, the other important packages that was used are utm for conversion of geo-coordinates, folium for visualising on the map [C8] and shapely for creating the polygon for geofencing [C7].

Chapter 4

Project Implementation

4.1 Data Collection and preprocessing

The data for pubs, cafe, restaurant and Tourist_spots across London was obtained from Overpass API. Open Overpass Turbo and we can use Overpass API query to query the required points. This is done as shown in figure 4.1.

```
[out:json];
area[name="London"];
(
  node["amenity"~"cafe"](area);
);
out center;
```

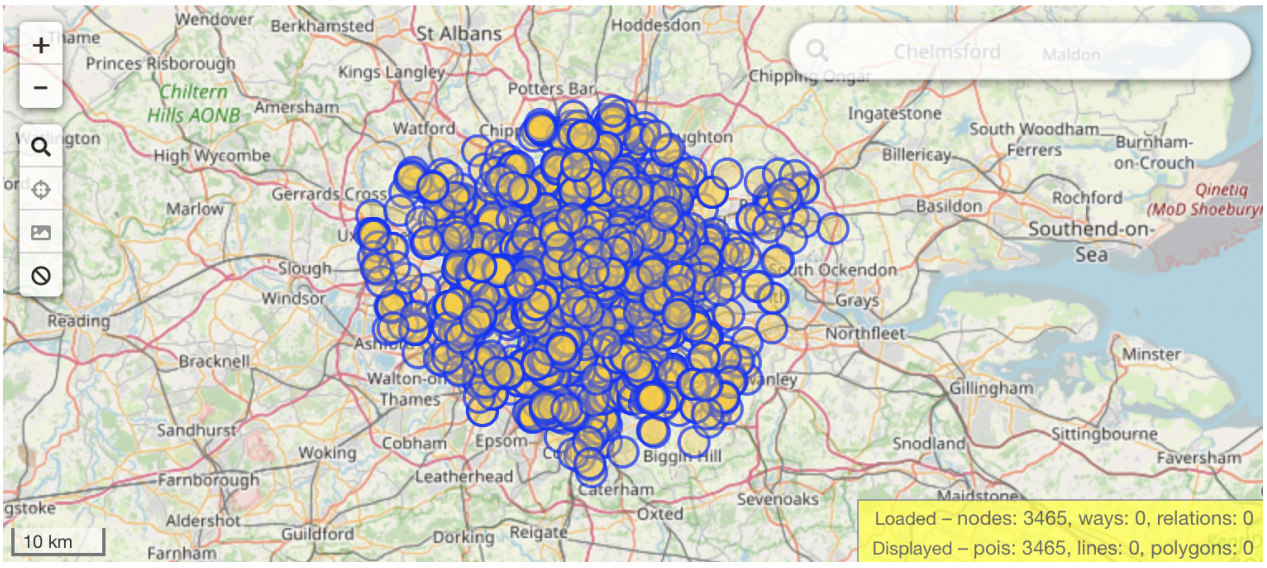


Figure 4.1: Query for cafe in London

In order to download the data, click on Export tab. Then, select download right next to

GeoJSON in the Export window.

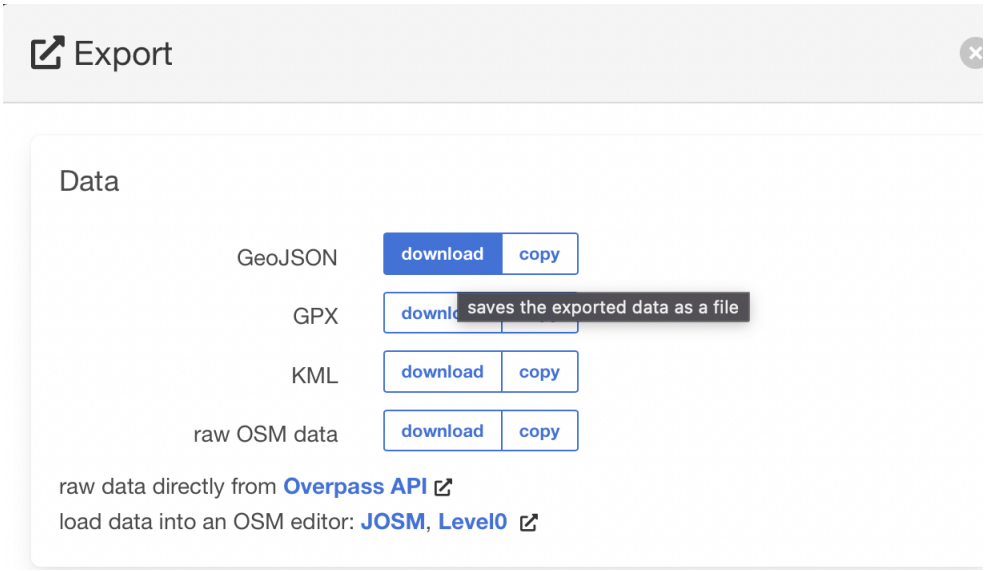


Figure 4.2: Download GeoJSON file

We have saved these data as four different geojson files - 'pub.geojson', 'cafe.geojson', 'restaurant.geojson' and 'Tourist_spot.geojson'. According to which type of place the person want to vist, the appropriate data file will be imported.

There are 3 user inputs that the programme requires and that are the current location, the type of place the user want to visit and the mode of travel. We use 'Nominatim' from geopy package to determine the geolocation for our initial point. For this, we create a 'geolocator' object using 'Nominatim' and when the current location is received, the geocoder is used to find the coordinates of the location. The default domain of this geocoder is OpenStreetMap. Along with this, the latitude and longitude that is obtained from the coordinates are saved as lat and long respectively to be used in the further parts of the code. In the figure 4.3, 'orig' is the tuple (lat,long). Hence, we have $lat = 51.523378$ and $long = -0.033435$ for 'Mile End'.

```
Enter your current location: Mile End

1 print("Current location is", coordinates)
2 print ("Coordinates for the current location is ", orig)

Current location is Mile End, Mile End Road, Mile End, London Borough of Tower Hamlets, London, Greater London, Eng
land, E3 4PH, United Kingdom
Coordinates for the current location is  (51.5253378, -0.033435)
```

Figure 4.3: Current Location and its coordinates

After we get the location name, the type of place and mode of travel is asked to be entered.

Based on the type of place we need to visit, the apropriate geojson file is imported and mode of travel is used in all the cases hereafter where we need to filter the locations.

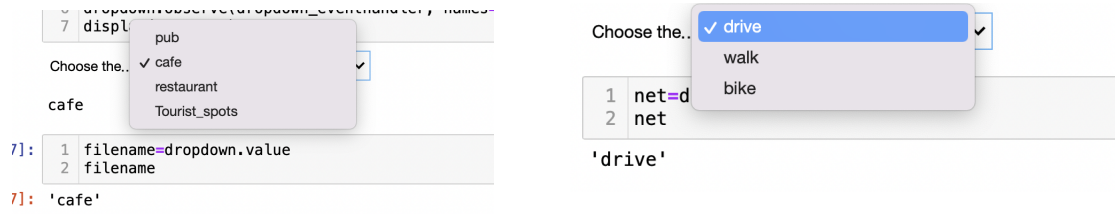


Figure 4.4: Selecting the type of place to visit and mode of travel

The major focus of our concern in the project is to find places that be reached within a user-input time. So the maximum time that should be taken to reach our destination is asked. Since it is evident that most people would prefer to minimize the time taken to travel, the time is calculated as minutes. The higher the time travel, the programme may take more time since there will be more points included in the data set despite filtering with our mentioned techniques. For example, lets say we select drive as the mode of travel and 15 minutes as the maximum travelling time. One can cover a large distance within that time if he is driving and thus the number of locations will be really high compared to if we are giving 1 or 2 minutes as the travel time.

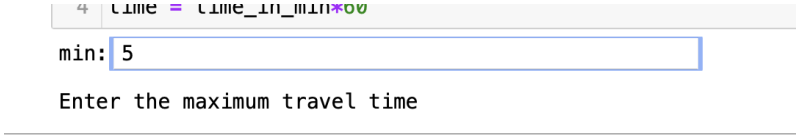


Figure 4.5: Input the maximum time to reach the destination

The data files were checked to find any empty columns or rows to avoid wasting of time when executing the program. The columns are chosen from the data set such that the data can give us a brief description of the places that we are dealing with.

	id	addr:postcode	name	opening_hours	outdoor_seating	geometry
0	node/15758157	RG1 4PS	Global Cafe	None	yes	POINT (-0.96806 51.45285)
1	node/20695954	None	The Beech Street Cafe	None	yes	POINT (-0.31297 51.75212)
4	node/25475389	None	Woburn Cafe	None	None	POINT (-0.12925 51.52658)
5	node/25497832	None	Cafe Angel	None	None	POINT (-0.11611 51.52476)
6	node/25991205	E8 2EZ	Jack's Cafe	None	None	POINT (-0.07248 51.55090)

Figure 4.6: Data for all the preferred locations in London

4.2 Implementing Geofencing

Before we create the boundary with which we are using geofencing, we need to fix on the distance from the point that we are trying to take as the boundary. For this purpose, we initiate a dictionary with mode of travel as key and its value being the average speed in London for each mode. With Speed and the input time, we can calculate the distance 'dist' by $Distance = Speed \times Time$. 'propagate' function from geog package is used to create a circle from a given point and passing multiple angles as the parameter with the distance that we calculated prior. We construct a Polygon form the polygon points using 'Polygon' function from geometry module of shapely and the geometric object is mapped onto GeoJSON-type object and saved as 'Circle.geojson'. This is then used as a mask to examine if the data points are within the constructed circle or outside of it. We create a new column 'geofence' where True is given if the point is inside the circle and False if it is outside the circle. In our example shown, we got 4070 points with False and 475 as True.

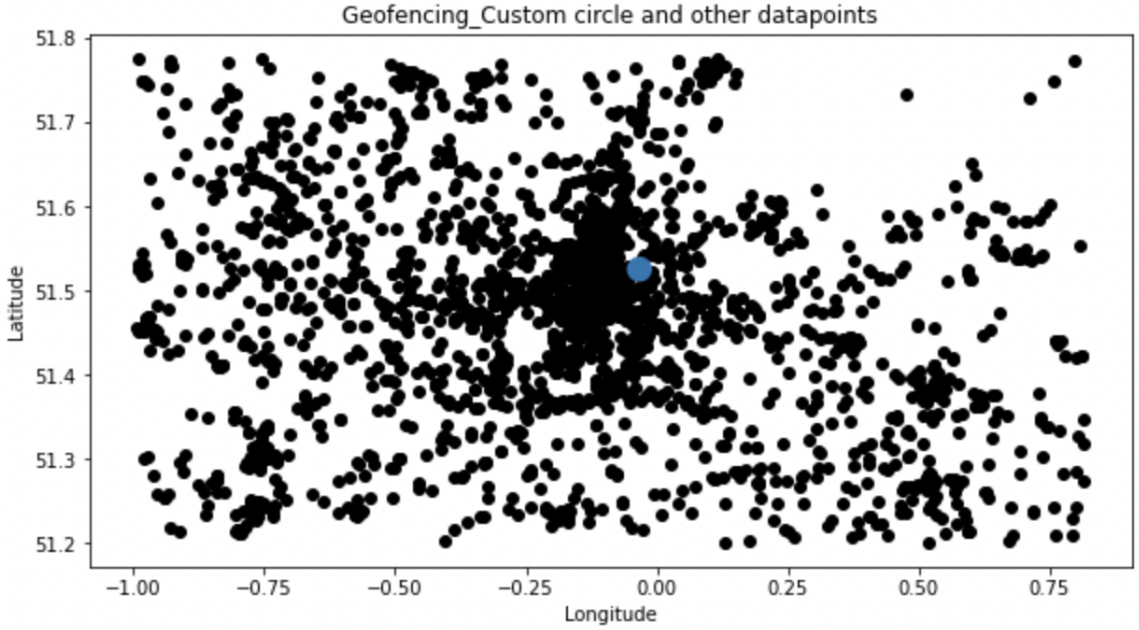


Figure 4.7: Scatter plot of the locations along with the geofencing area

The figure 4.7 shows the plot where the data points and the circle used for geofencing are shown. The distance that we got in this case was $833.33m$ and this was used to construct the circle shown in blue colour. Each of the black points show cafe locations. Clearly, there are cafes which lies inside the constructed circle and those outside of it. The circle is constructed with the maximum distance the person can travel with the given conditions as its radius.

Thus we eliminate a majority of points that are evidently lies beyond the point that can be reached within the mentioned time.

Thus the dataframe in the figure 4.8 is updated with the geofence values. Now, we filter out the points where the value in the column 'geofence' is True. Thus, we can eliminate all the points that lie outside of the circle. This method of filtering makes the programme run faster as we need not bother with irrelevant points.

	id	addr:postcode	name	opening_hours	outdoor_seating	geometry	geofence
3113	node/5931308447	None	Cafe Chi	Mo-Fr 07:00-16:00	yes	POINT (-0.44176 51.48041)	False
3594	node/6763901763	None	Hidden Coffee Roasters	Mo-Fr 07:00-17:00;Sa-Su 09:00-17:00	None	POINT (-0.13865 51.54176)	False
121	node/332117592	None	Sandwich Street Kitchen	None	None	POINT (-0.12628 51.52772)	False
1119	node/1951120429	E18 1AG	Waribashi	None	None	POINT (0.02702 51.59210)	False
3809	node/7142467325	None	Milk Cool	None	None	POINT (-0.19756 51.51064)	False

Figure 4.8: Dataframe with geofence either False or True

The filtered points are shown as blue dots in the figure given in 4.9. plotly_express package is used to plot the given map. The mapbox-style is taken as open-street-map in this case and is constructed with the latitude and longitude values that we get from the column 'geometry' of the above dataframe.

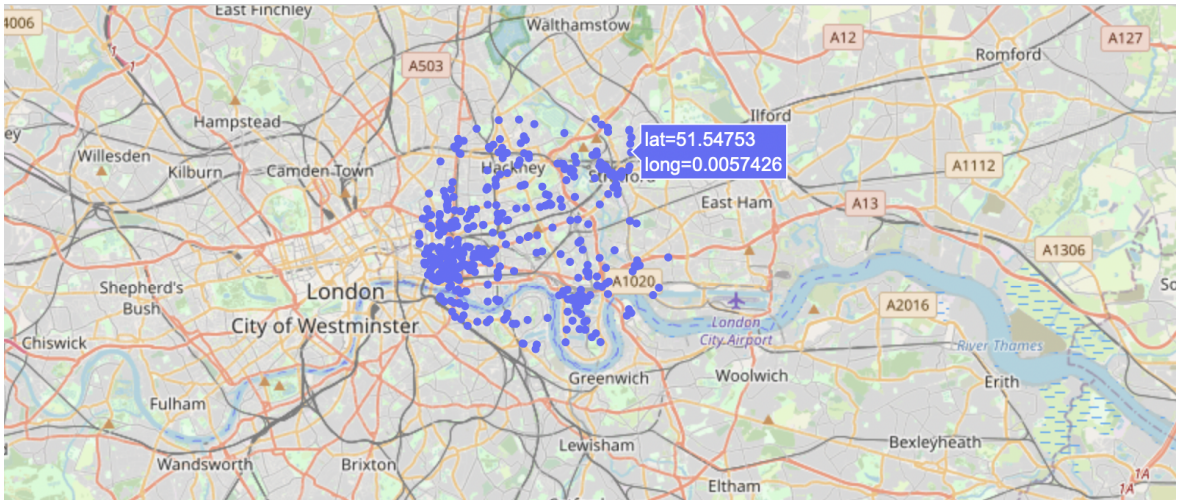


Figure 4.9: Filtered locations in OpenStreetMap

4.3 Finding shortest path and filtering locations

Since we need to construct the graph within a certain distance from a certain geocode, we use graph_from_point function from osmnx package. The distance 'dist' that we calculated

previously for geofencing and the mode of travel are taken as the other two parameters for the function. This means that from the center point 'orig' , the graph will be constructed such that the nodes within the distance 'dist' and the edges that can be travelled using the defined mode of travel will be retained. Hereafter, we add two edge attributes or weights to our graph G. The first one being the speed and the second is travel time. The functions that are utilised for imputing these values is from 'speed' module of osmnx package. We use 'add_edge_speeds function which assign weights of the edges, a speed that is based on the mean maxspeed value of edges of each highway type [C4]. If none of the edges of a highway type has a maxspeed value allocated, the function will determine the weight by taking the mean of all maxspeed values in the graph. The assigned weight would always be converted to kmph even if a maxspeed of an edge is given in mph. The graph G, plotted in 4.10, is a multidigraph with 10238 nodes and 22794 edges.

Further, 'add_edge_travel_times' returns the graph with travel time assigned to its edges where the travel time is determined by the length and speed(kph) attributes of the edge. Hereafter, we project the graph using 'project_graph' from projection module of osmnx. By default, the function projects the graph to UTM CRS (Coordinate Reference System). We do this since the function that we are using for for finding nearest node of the locations require a projected graph for an accurate result. The projected graph is plotted in the figure 4.10.

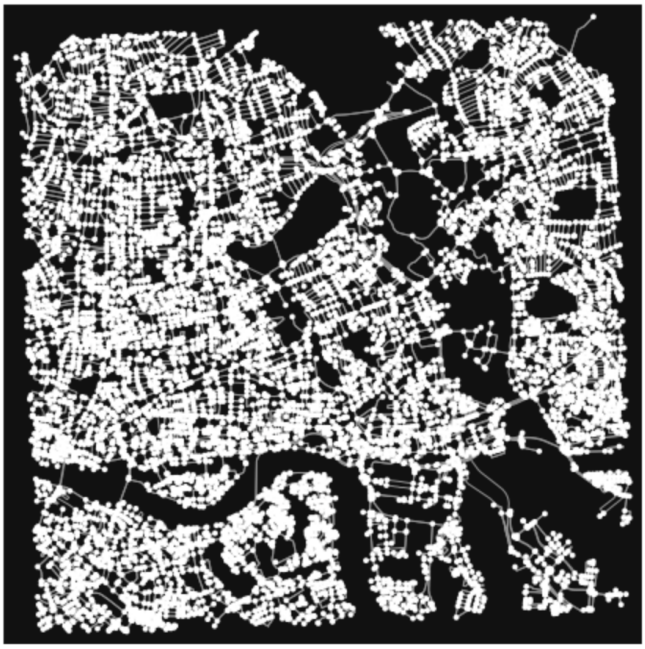


Figure 4.10: Graph model of the input location

Since the projected graph is in UTM coordinate system, we need to change the coordinates of our initial locations and the data points. we change the latitude/longitude to UTM for our data points and save it as a new column 'loc' in our data frame. Along with this, we create a new column 'route' where we save the shortest path from our origin point to each of the data points. In order to do this, we introduce a function 'path' whose parameters are the graph, the initial point and the destination(each data point). Nearest nodes of the origin and destination points are found first and then this is used to obtain the shortest path which is found using 'shortest_path' function from networkx package. The default algorithm for this function is Dijkstra's. we can alternatively use A* algorithm with 'astar_path' function. However, 'shortest_path' with Dijkstra's algorithm was faster compared to 'astar_path'. A detailed explanation of this is given in Appendix A.

Another column 'time' is introduced in the datafrome which given take the sum of all weights (travel-time) of the edges in the path. 'get_route_edge_attributes' from utils_graph module of osmnx package returns a list of weights and its sum is rounded to an integer. This value is taken as the resultant time.

	id	addr:postcode	name	opening_hours	outdoor_seating	geometry	long	lat	loc	route	time
1962	node/3831795563	E2 9LH	Wild Bean Cafe	None	None	POINT (-0.05607 51.52925)	-0.056072	51.529249	(704195.0245311586, 5712790.800642301)	[961587776, 961587760, 961587771, 720900903, 1...	227
3181	node/6018051912	EC3N 4AB	Raven's Cafe	None	None	POINT (-0.07601 51.50743)	-0.076014	51.507430	(702909.0658014127, 5710309.447831936)	[961587776, 961587760, 961587771, 720900903, 1...	367
3882	node/7248621568	E14 2BB	Sandwich Plus	None	None	POINT (-0.00588 51.50949)	-0.005883	51.509486	(707765.4814585033, 5710734.926421213)	[961587776, 961587760, 961587771, 246860043, 1...	253
1821	node/3567118899	SE1 2RS	Costa	Mo-Fr 06:00-19:30, Sa 07:00-19:00, Su 07:30-19:00	None	POINT (-0.08155 51.50422)	-0.081554	51.504220	(702538.9544197407, 5709937.228074911)	[961587776, 961587760, 961587771, 720900903, 1...	489
2167	node/4324478924	E1 2QE	DeLites	Mo-Fr 08:00-22:00; Sa 09:00-22:00; Su 09:00-21:00	None	POINT (-0.05645 51.51158)	-0.056453	51.511579	(704247.6883658497, 5710825.208472768)	[961587776, 961587760, 961587771, 720900903, 1...	269

Figure 4.11: Data points with its shortest route from current location & travel time

The dataframe 4.11 shows the 'route' and 'time' columns as the last two columns. The time for the first entry is given as 227. This means that it take 227 seconds to reach the destination from the origin through the corresponding shortest route. Since our maximum travel time is 5 min (300 seconds), all the entries with time greater than 300 can be removed from our list.

	id	addr:postcode	name	opening_hours	outdoor_seating	geometry	long	lat	loc	route	time
56	node/279249388	E1 2DL	Cafe Donatella	None	None	POINT (-0.06037 51.51542)	-0.060367	51.515420	(703958.9831435235, 5711241.353879541)	[961587776, 961587760, 961587771, 720900903, 1...	230
285	node/443418105	E3 5LX	Hiland	None	None	POINT (-0.03441 51.53209)	-0.034413	51.532088	(705684.2061303565, 5713167.116833195)	[961587776, 961587760, 961587771, 246860043, 1...	115
317	node/470362541	E14 4QT	Brera	None	None	POINT (-0.02219 51.50513)	-0.022191	51.505128	(706653.8191039551, 5710204.246379174)	[961587776, 961587760, 961587771, 720900903, 1...	275
326	node/475381384	E1 1JX	Casablanca Cafe / Alauddin Sweets	None	None	POINT (-0.06661 51.51719)	-0.066614	51.517192	(703517.7490175099, 5711420.932927357)	[961587776, 961587760, 961587771, 720900903, 1...	205
493	node/652850246	E3 2AN	Mighty Bite	None	None	POINT (-0.02263 51.52782)	-0.022627	51.527818	(706520.8409588585, 5712725.61753162)	[961587776, 961587760, 961587771, 246860043, 1...	57
...	...	...	...	...	...	...	...	...	...	...	...
4706	node/9661248645	None	Knot	None	None	POINT (-0.00386 51.54109)	-0.003860	51.541088	(707761.8561575535, 5714254.11355128)	[961587776, 961587760, 961587771, 246860043, 1...	293
4707	node/9661248647	None	Caffè Nero	None	None	POINT (-0.00292 51.54164)	-0.002918	51.541639	(707824.6574936745, 5714318.095418482)	[961587776, 961587760, 961587771, 246860043, 1...	288

Figure 4.12: Filtered data points

The table shown in 4.12 is the filtered list of cafes which can be reached within the input time. If we examine the 'time' column, we can see that all the values shown are less than 300.

Chapter 5

Results and Discussion

5.1 Results

The filtered list will contain all the locations that can be reached within the maximum time. The names of these places are then converted to a list and displayed in a dropdown widget. This becomes the input for displaying the corresponding shortest path on the map.

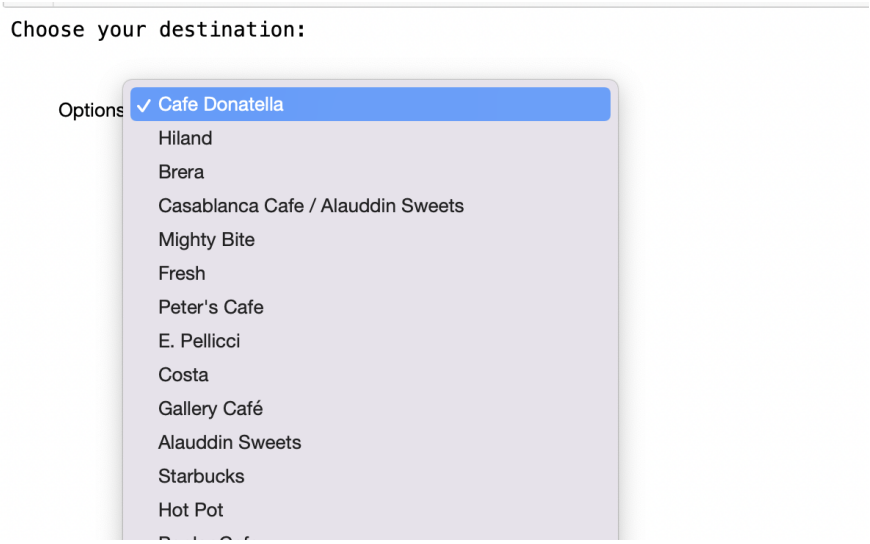


Figure 5.1: List of all target points that can be reached within the input time

The figure 5.1 shows the choice of destinations and in our particular example, we needed to find a cafe which can be reached within 5 minutes from Mile End. This list is not sorted based on the travel time. We have done this later when developing the interactive dash board to demonstrate how it can be developed into a web application.

In order to give an illustration of where the origin and the chosen destination point is, a

map is shown here in the report. This is not included in the project and the code is given in Appendix B. The popup contain the status of the location if it is the Origin or the target point along with the latitude/longitude coordinates. 'Cafe Donatella', which was selected from the set of choice, is shown by the red point and 'Mile End' is the one in blue colour in 5.2.

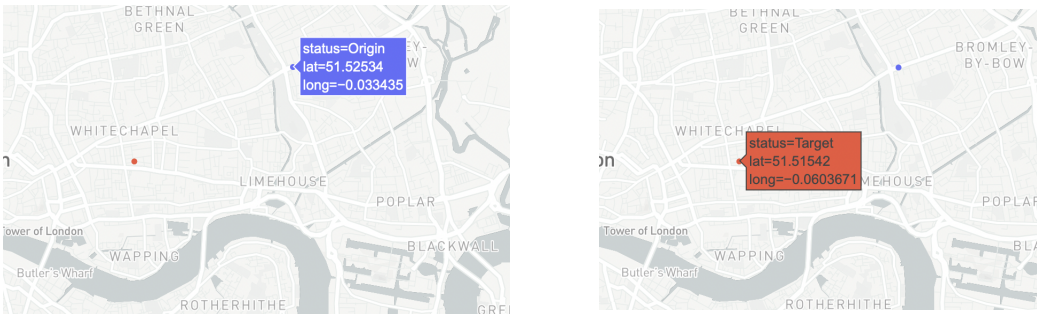


Figure 5.2: Map showing origin and selected target point

5.1.1 Shortest Path to the destination

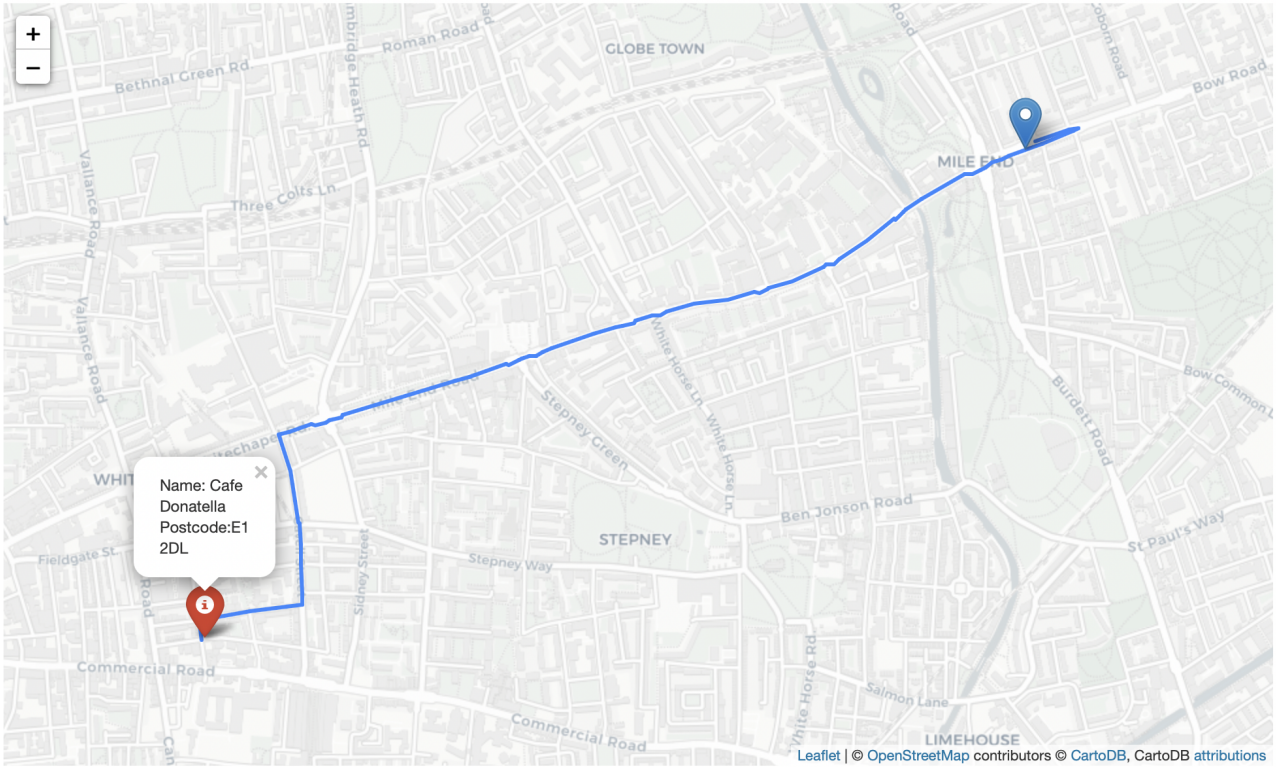


Figure 5.3: Shortest path from origin to selected target point

On selecting our target point, the map will be displayed with the origin point, target point and the shortest path between the two points. In the figure 5.3, origin is represented with the blue marker, destination with red and the blue line is the shortest path between them. The popup in the red marker will show the destination name, its postcode and the opening hours (if given in the data).

5.1.2 Interactive Dashboard using Mercury

The project was modified to construct an interactive dashboard from .ipynb file created with jupyter notebook. The most popular option to generate such a dashboard is mercury and voila. voila converts the jupyter notebook file as a standalone application and it is slow in processing the files. There are two other packages in python that aid developing such a front end. One is Flash and the other is dash. dash is best when used for specific requirements and Flash is the one to develop generic web applications. However, mercury framework was used since we can create an aesthetically pleasing application with minimal efforts.

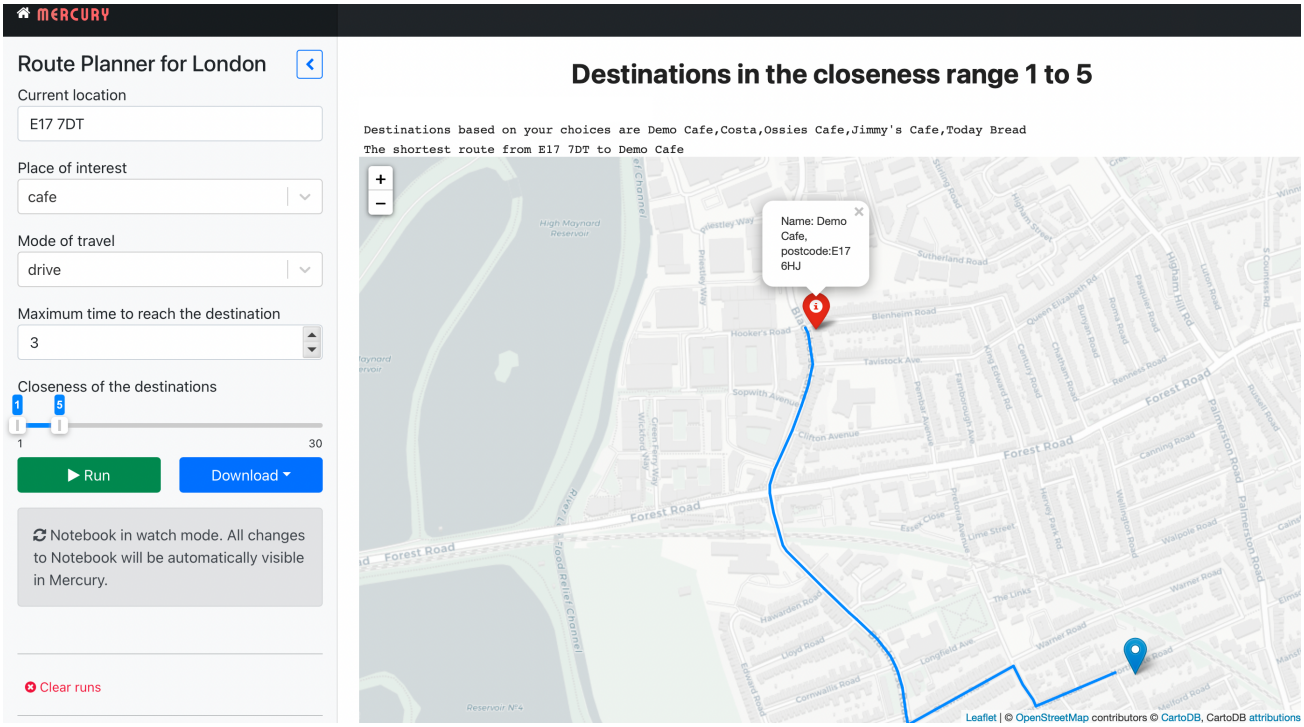


Figure 5.4: Interactive Dashboard

In the web framework, one more additional option was included. The python file was modified so that the user can select the destination to be viewed based on how close the destination is to the user. If the user wants to see the nearest 5 place of interest, then he/she should set

it the range as 1 to 5. This will give the name of top nearest places along with the maps. All the maps can be viewed by scrolling down.

5.2 Discussions

5.2.1 Selecting Dijkstra's algorithm

The execution time for 'path' function with'shortest_path' (using Dijkstra's) and an alternative to the function using 'astar_path' were found in the case where we take 'drive' as mode of travel for varying value of travel time. The functions were used to find shortest path of all the data points (in the filtered data frame) from the origin consecutively. The number of data points after filtering, graph nodes and edges along with the execution time of functions were noted in 5.5.

Travel time	No. of data-points	Graph Nodes	Graph Edges	Dijkstra's	A*
1	12	364	761	0.035507	0.041730
3	135	2110	4900	0.129851	0.356222
5	475	10238	22794	15.863240	26.610161
8	1109	21428	49036	46.157913	85.576082

Figure 5.5: Execution times for Dijkstra's and A* based on change in travel time

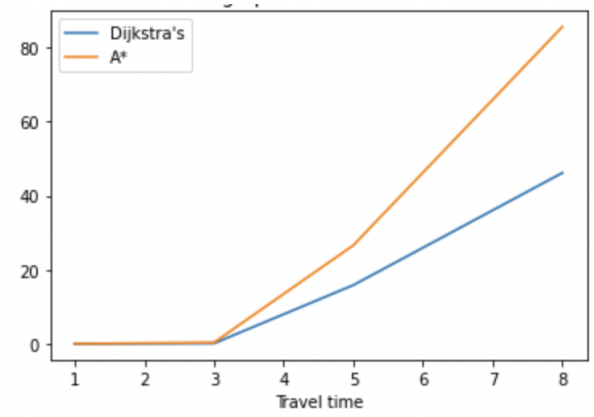


Figure 5.6: Line graph showing execution times for Dijkstra's and A* against travel time

In table 5.5, cafe was selected as the place to visit. There are 12 cafes that one can reach within 1 minute by drive from 'Mile End' and the resultant graph G contains 364 nodes and 761 edges. Under these circumstances, we used 'path()' with Dijkstra algorithm and 'path1()' with A* algorithm to find the shortest path from origin to all these points. path 1

took 0.035507 seconds to run and A* found all the shortest paths in 0.041730 seconds. As the travel time increases, the number of cafes increases and so does the number of nodes and edges of the graph. Hence, the time of execution increases along with it since the number of shortest path which needs to be determined also increases. A graph was plotted to show how the execution time of the algorithms deviates as the travel time increases. Initially, both the algorithms performed identically. However,with increase in travel time, path() function with Dijkstra's outperformed that with A*. Hence, Dijkstra's was chosen in our project to increase the efficiency.

5.2.2 Data Flow Diagram

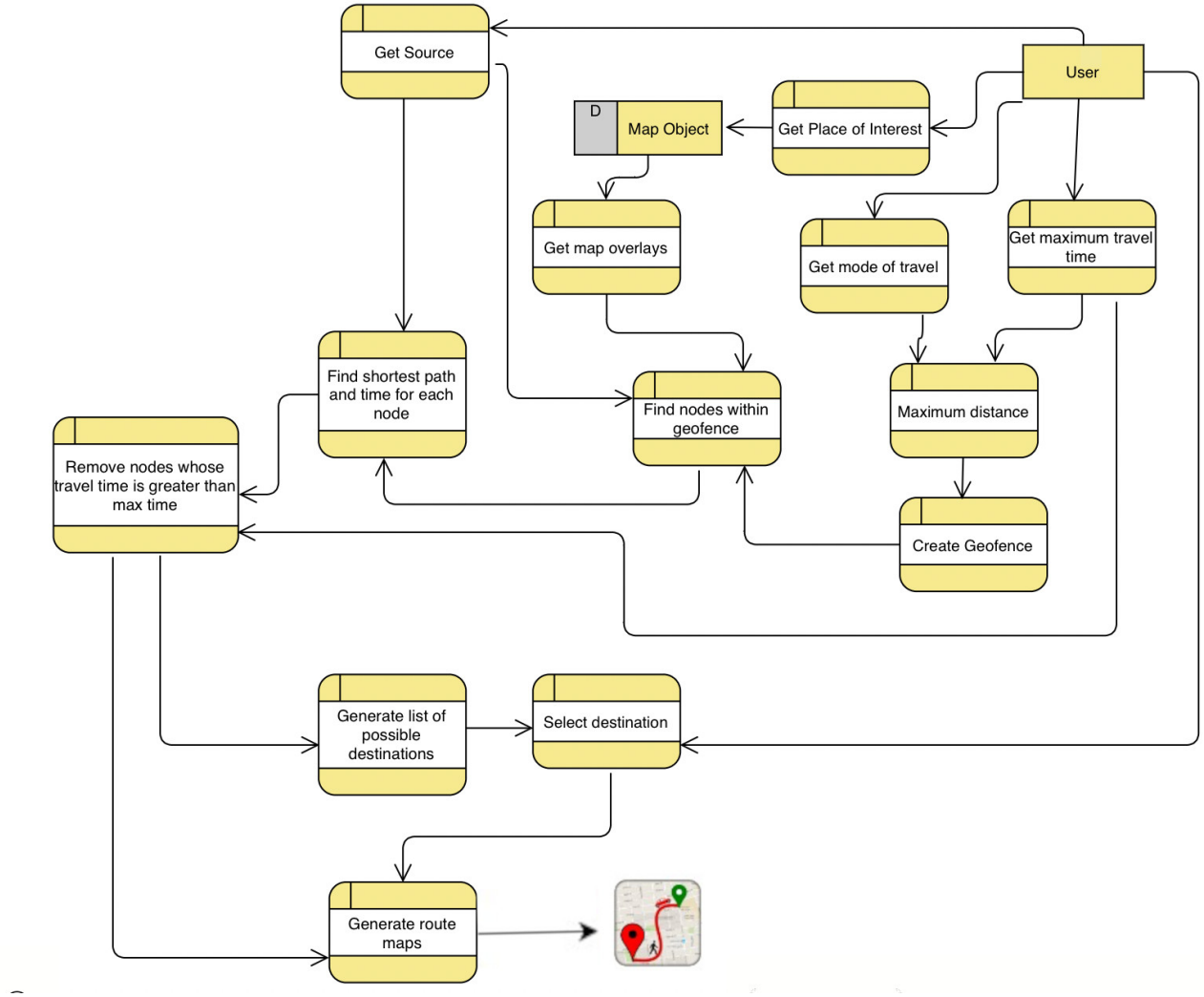


Figure 5.7: Data Flow Diagram

Chapter 6

Conclusion

The project was designed as to give the user the option to filter the nearest locations by limiting the locations based on the maximum travel time along with certain other conditions. Even though, google do provide services on showing the nearest restaurants or gas stations. It is not specifically designed for the task and there are limited options that we can choose from.

The scope of this project is tremendous. The project can be extended to include other amenities and can be advance so that it integrates live traffic data when determining the shortest path. A survey was conducted among the peers and close acquaintances to analyse the scope of the project. The majority claimed that it will be beneficial if this project was developed into a mobile application since it is specifically created according to the user's preferences.

Chapter 7

AppendixA

7.1 Project Planning and design

7.1.1 Project Initiation

The project idea was built upon analysing what a user requires compared from other applications that utilises with shortest path algorithms. A market research was conducted on existing technologies and platforms were identified which works with finding meeting places or nearest place. The data required for the project was downloaded from Overpass API. Focused on which packages and modules were needed for the project and the input methods required in the project.

7.1.2 Planning

Designed a plan to reach the filtered data by finding the shortest path from the current location to the data points and filtering the data based on the travel time. Milestones for completing the project was set and algorithms were analysed to decide on which algorithm to choose.

7.1.3 Execution

There were changes done in the execution procedure. Since it was a relatively large data finding shortest path for each data points was not feasible, the data points in the proximity of the current location was needed to be considered. Checked for a possible solution to set up such a boundary depending on the distance that can be reached based on the time and mode of travel and decided to try geofencing. After getting the result, an interactive dashboard was developed for a better user experience with an addition to includes a new

input variable to determine how many nearest locations should be shown and how close the locations should be.

7.1.4 Management

Each weekend was set as a target to complete each tasks in the project and started developing the project report after the completing the rough draft for coding

7.1.5 Review

The project achieved it initial goals on filtering the required data points and plotting the shortest path with initial and target locations marked. Initially, it was planned to design the project in Flask. However, the application was created with mercury due to time constraints and since mercury was efficient for our particular task.

Chapter 8

AppendixB

Implementing Dijkstra's and A* algorithms on the graph in 3.1a

```python
#import the required modules
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
%matplotlib inline
```

```python
#initiate the graph
G = nx.Graph()

#add nodes in appropriate positions
G.add_node("x1",pos=(0,3))
G.add_node("x2",pos=(0,0))
G.add_node("x3",pos=(3.5,0))
G.add_node("x4",pos=(3.5,1.4))
G.add_node("x5",pos=(3.5,3.1))
G.add_node("x7",pos=(9,1.43))
G.add_node("x6",pos=(9,3.2))
G.add_node("x8",pos=(9,0))


#Add edges along with the edge weight
G.add_edge("x1", "x5", weight=0.186)
G.add_edge("x1", "x2", weight=0.1775)
G.add_edge("x2", "x3", weight=0.182)
G.add_edge("x3", "x4", weight=0.094)
G.add_edge("x4", "x5", weight=0.09)
G.add_edge("x3", "x8", weight=0.222)
G.add_edge("x4", "x7", weight=0.217)
G.add_edge("x5", "x6", weight=0.221)
G.add_edge("x6", "x7", weight=0.0946)
G.add_edge("x7", "x8", weight=0.0953)
```

Figure 8.1: Creating the graph

```
#get the dictionary with nodes as keys & its position coordinate as the values
pos=nx.get_node_attributes(G,'pos')

#draw the nodes with its name and edges connecting them
nx.draw(G, pos, node_color='lightgreen',with_labels=True, node_size=600)

#get a dictionary noting the edges as key and weights as its value
edge_labels = nx.get_edge_attributes(G, "weight");

#add the edge labels in the graph
nx.draw_networkx_edge_labels(G, pos, edge_labels);
```

Figure 8.2: Plotting the graph

```
#finding shortest path from x1 to x7 using dijkstra's algorithm in networkx
path_nodes = nx.dijkstra_path(G, source="x1", target="x7") #or nx.astar_path(G, "x1", "x7")

#draw the nodes with its name and edges connecting them
nx.draw(G, pos, node_color='lightgreen',with_labels=True, node_size=600)

#add the edge labels in the graph
nx.draw_networkx_edge_labels(G, pos, edge_labels);

#highlight the nodes corresponding to the shortest path usin 'red'
nx.draw_networkx_nodes(G,pos,nodelist=path_nodes,node_color='r')

#create the edges connecting the path_nodes
path_edges = list(zip(path_nodes,path_nodes[1:]))

#draw edges using red colour and with width 2
nx.draw_networkx_edges(G,pos,edgelist=path_edges,edge_color='r',width=2)
```

Figure 8.3: Finding and Plotting the shortest path

```
from datetime import datetime as dt

#dt.now() gives the present time
start_time = dt.now()

#finding shortest path from x1 to x7 using dijkstra's algorithm in networkx
nx.dijkstra_path(G, source="x1", target="x7")

#printing the execution time
print(dt.now() - start_time, "time of execution for dijkstra's")

#initiate start time for A*
start_time = dt.now()

#finding shortest path from x1 to x7 using A* algorithm in networkx
nx.astar_path(G, "x1", "x7")

#printing the execution time
print(dt.now() - start_time, "time of execution for A*")
```

```
0:00:00.002895 time of execution for dijkstra's
0:00:00.000910 time of execution for A*
```

Figure 8.4: Execution time for the algorithms

Bibliography

[Deo] Narsingh Deo, *Graph Theory in Application with Applications to Engineering and Computer Science*, 1974, Prentice-Hall, Inc., Eaglewood Cliffs, New Jersey.

[Shekar] Shekar, Pavithra C., *Applications of Graph theory on Google map Application*, 2019, CIKITUSI Journal For Multidisciplinary Research, Volume 6, Issue 4.

[Pearson] Pearson, Helen. *THE TIME LAB, WHY DOES MODERN LIFE SEEM SO BUSY?*, 2015, Macmillan Publishers Limited.

[MD] D'souza, Melita., D'souza, Dwayne Dexter. , *An Analysis of Bellman-Ford and Dijkstra's Algorithm*, 2019, CIKITUSI Journal For Multidisciplinary Research, Volume 6, Issue 4.

[HD] Hu, Shunfu., Dai, Ting., *Online Map Application Development Using Google Maps API,SQL Database, and ASP.NET*, 2013, International Journal of Information and Communication Technology Resarch, Volume 3 No. 3.

[ENGM] Edward He, Natashia Boland, George Nemhauser, Martin Savelsbergh *Time-Dependent Shortest Path Problems with Penalties and Limits on Waiting*, 2020, INFORMS Journal on Computing 33(3): 997-1014.

[Schrijver] Schrijver, Alexander, *On the History of Shortest Path Problem*, 2010, Documenta Math.

[Wilson] Wilson, Robert J., *Introduction to Graph Theory*, 4th Edition, 1996, Prentice Hall.

[SW] Sedgewick, Robert., Wayne, Kevin. *https://algs4.cs.princeton.edu/42digraph/Algorithms*, 4th Edition.

[Berkeley] *Lecture Notes- Graph Theory*.

[Ralphs] Dr. Ted Ralphs *Lecture Notes- Algorithms in Systems Engineering*.

[MIT] *Lecture Notes - Directed Graphs*, Mathematics for Computer Science, MIT Open Course.

[HR] Otto Huisman, Rolph A. de, *Principles of Geographic Information System*, 2001, The International Institute for Geo-Information Science and Earth Observation.

[AD] Anthony .C. Ijeh, David .S. Preston, et al. *Geofencing Components and Existing Models*, 2014, International Journal of Computer Applications, Volume 107 - No. 16.

[GKS] Gupta, C. B., Kumar, Sandeep., Singh, S. R., *Advance Discrete Structure*, 2010, I.K. International Publishing House Pvt. Ltd.

[CA] Chapuis, G., Andonov, R., *GPU-accelerated shortest paths computations for planar graphs*, 2017, Advances in GPU Research and Practice.

[PD] Panahi, S., Delavar, M. R., *A GIS-based Dynamic Shortest Path Determination in Emergency Vehicles*, 2008, World Applied Sciences Journal 3, 88-94.

[WPF] Peng, Wei., Hu, Xiofeng., et al. *A Fast Algorithm to Find All-Pairs Shortest Paths in Complex Networks*, 2012, International Conference on Computational Science, ICCS 2012, Procedia Computer Science 9 (2012) 557 – 566 .

[SAL] Sharma, H., Alekseychuk, A., Leskovsky, P. et al. *Determining similarity in histological images using graph-theoretic description and matching methods for content-based image retrieval in medical diagnostics*, Diagn Pathol 7, 134 (2012).

[Aeden] Pettit, Aedan D., *Developing a Web-Based Application for Finding Meeting Places* (2020). Senior Independent Study Theses. Paper 9080.

[OWM] *Open Web Mapping*, Department of Geography, PennState College of Earth and Mineral Sciences.

[C1] *NETWORKX-Network Analysis in Python*,NETWORKX documentation (Viewed - 10 July, 12:30 PM)

[C2] *Geopandas 0.11.0*, Geopandas documentation(Viewed - 10 July, 2:30 PM)

[C3] *Geopy Documentation* Viewed - 11 July, 3 PM)

[C4] Abdishakur, *The Art of Geofencing in Python*, 2022 (Viewed - 12 July, 2 PM)

[C5] *Generating polygon representing rough 100km circle around latitude/longitude point using Python?*, Stack Exchange (Viewed - 13 July, 11:30 AM)

[C6] *OSMnx 1.2.2*, OSMnx documentation (Viewed - 10 July, 3 PM)

[C7] *Shapely Documentation* (Viewed - 13 July, 2:30 AM)

[C8] ferhatmetin, *Map Visualization with Folium*, 2021, Data Science Earth (Viewed - 19 July, 12:30 AM)

[C9] *Mercury Docs*, Mercury documentation (Viewed - 29 July, 12:30 AM) (Viewed - 29 July, 12:30 AM)

[JN] Janakiev, Nikolai. *Loading Data from OpenStreetMap with Python and the Overpass API*,

[WH] *whatshalfway*. (Viewed - 6 July, 2:45 PM)

[MW] *meetways*. (Viewed - 6 July, 3 PM)

[Sandy] WrittenHouse, Sandy, *The 5 Best Sites to Find Halfway Points Between Two Places*, 2022 (Viewed - 12 Aug, 10 AM)

[Verdi] *Working With Graphs*, 2019 (Viewed - 15 Aug, 2 PM)

[CMS] *Lecture Notes - Carnegie Mellon University* (Viewed - 19 Aug, 7 PM)

[Greef] Lilian de Greef, *Lecture Notes- Dijkstra's Algorithm*, 2017, Data Structures and Algorithms. (Viewed - 19 Aug, 9 PM)

[Isaac] *A* search algorithm*, Isaac Computer Science (Viewed - 22 Aug, 4 PM)

[TC] *Understanding Time Complexity Calculation for Dijkstra Algorithm*, 2021, Baeldung (Viewed - 24 Aug, 9 PM)

[AI] *Exercises: Artificial Intelligence*, 2017, Data Structures and Algorithms. (Viewed - 24 Aug, 11:30 PM)

[USGS] *What does the term UTM mean? Is UTM better or more accurate than latitude/longitude?*, 2017, Data Structures and Algorithms. (Viewed - 24 Aug, 11:30 PM)

[Sale] *What Is Geofencing Marketing? How to Add It to Your Marketing*. (Viewed - 31 Aug, 12:45 PM)

[ons] *Average actual weekly hours of work for full-time workers*, 2022.