Least Polluted Routes: algorithms and application

Enhao Sun

Master of Artificial Intelligence
School of Informatics
University of Edinburgh
2019

Abstract

Air pollution in cities is becoming a serious environmental problem with increasing concern in public. Pollutants in air are harmful to health especially for certain groups of people who have cardiovascular disease, breathing problems such as asthma and bronchitis. This project aims to develop an Android mobile application to help people plan journeys which takes air pollution as concern in finding the least polluted routes between places. Algorithms including Dijkstra, A* and Bidirectional A* were investigated. In addition, modifications were made to heuristic functions of A* in order to find the target faster. Dummy pollution data which covers the large area of the city of Edinburgh were generated by Generative adversarial network (GAN) with sample pollution data around the Meadows covered by six stationary AirSpeck air quality monitors. Algorithms were tested with planning different distance of routes in different areas in the city of Edinburgh and an analysis of the results is presented. An Android mobile application has also been developed for visualising the map as well as planning the cleanest routes. A server has been implemented and deployed on Google Cloud App Engine for running path planning algorithms and sending results to the Android mobile application.

Acknowledgements

I would like to express my appreciation for the support and guidance of my supervisor, Professor D. K. Arvind.

Additionally, I am grateful to Zoë Petard for sharing her experience and advice over the course of many meetings.

Finally, I would like to thank my friends and family for their encouragements and insights over this summer.

Table of Contents

Chapter 1

Introduction

This project aims to develop an Android mobile application to serve as an route planner to avoid poor air quality road areas. This route planner will be beneficial for everyone, but especially certain groups who are more vulnerable to air pollution including elderly people, pregnant women, children and people who are suffering from respiratory diseases (e.g. asthma)[7].

Big cites around the world have high levels of air pollution. For example, Beijing in china such as Beijing has PM2.5 concentration with 61 $\mu g/m^3$ reaching the AQI (Air Quality Index) level 153 on 8^{th} of August in 2019 (provided by Air-Matters [1]). According to U.S EPA PM 2.5 AQI [8], the air is unhealthy when it's AQI level exceeds 150. AQI level between 95 and 125 is unhealthy for sensitive groups (mentioned as above). According to [15], a global study which analysed medical hospital admissions in 204 counties from 1999 to 2002 reported that PM2.5 could cause heart failure with a 1.28% increase in risk per 10 $\mu g/m^3$ PM2.5. Air pollution in Edinburgh is causing a public health crisis. Streets including Queensferry Road, St John's Road and Nicolson Street in Edinburgh had nitrogen dioxide exceeding $45\mu g/m^3$ last year according to Friends of the Earth Scotland [5].

Most of the time, people care about going to their destination via the shortest path. However, with ambient air pollution as a growing concern, it would be very helpful to people who are more vulnerable to air pollution to travel with a cleanest route. This project was motivated the development of an Android mobile application to help people plan the least polluted route in daily life. It is important and helpful that people can use this application to check and compare the air pollution levels (e.g PM2.5 concentration) of the least polluted routes and the shortest routes so that they can choose the most suitable path for themselves. There are a few important assumptions about

the idea of generating least polluted routes. First, when mentioning 'pollution', this report is referring to the corresponding PM2.5 value. Second, the person using this Android mobile application is a pedestrian which means that the route planned by the application is for walking. In addition, since the main user interface is the Android mobile application, it is important to prioritise fast solutions in developing algorithms. This will ensure a smooth user experience. The main contributions of this project are listed as follows:

- Map data were extracted from Open Street Map (OSM) for the large area of Edinburgh which covers the areas from Newhaven to Braid Hills (North to South) and Murrayfield to Duddingston (West to East) (See Figure 4.1b in section 4.2).
- A python version of a third-party Matlab library was developed to process the OSM data and get nodes and edges from it to form a graph. This python program can process large OSM data files which solves the problem where a previous project [18] was unable to process big OSM data files using Matlab.
- Four algorithms were developed to run on the graph: Dijkstra, A*, dynamic weighting A* and Bidirectional A*; Several experiments in planning least polluted routes with different distance (Long distance (> 3km), middle distance(2-3km) and short distance (<2km)) were implemented to test and validate the algorithms against each other, as well as analysing their runtime and the number of expanded nodes during the search.
- A generative adversarial network was used to generate dummy pollution data for a larger area in Edinburgh from the pollution data of a smaller area around The Meadows in Edinburgh. This was done in order to test and validate the performance of the four algorithms in planning paths with different distances even when live pollution data for the larger area is not available.
- The algorithms were tested with pollution data around The Meadows during a day on 2019-07-24 generated by EGAN Ryan [26]. Many insights were observed by analysing the changes of the least polluted routes during a day.
- An Android mobile application with an intuitive user interface was developed for people to use. A server which can run path planning algorithms was developed and deployed on the Google Cloud App Engine so that the Android mobile application can request and receive the least polluted routes from the server efficiently with the latest pollution data.

Chapter 2

Background

2.1 Description of air pollution

This section delivers a brief introduction and explanation on air pollution. Pollutants are a mix of particles and gases which are not always visible in the air, coming from different sources. In order to measure the air pollution, an air quality index (AQI) that rates the air condition based on concentrations of five main pollutants: ground-level ozone, particulate matter, carbon monoxide, sulfur dioxide, and nitrogen dioxide [9] is used by government agencies to publish air pollution levels [30]. Particulate matter (PM) is the term for a mixture of solid particles and liquid droplets in the air [6]. It includes PM2.5 and PM10. PM2.5 are inhalable particles with diameters that are 2.5 micrometers and smaller. Inhalable particles with diameters that are 10 micrometers and smaller are called PM10 [4]. Particles with diameters smaller than 10 micrometer can get deep into human's lungs and maybe even bloodstream [4] posing great health risks.

This project aims to use PM2.5 as the main pollutant of concern in measuring the AQI level of the air. Meanwhile, PM1 and PM10 of the planned routes will be displayed in the Android mobile application for reference.

2.2 AirSpeck static wireless air quality monitor

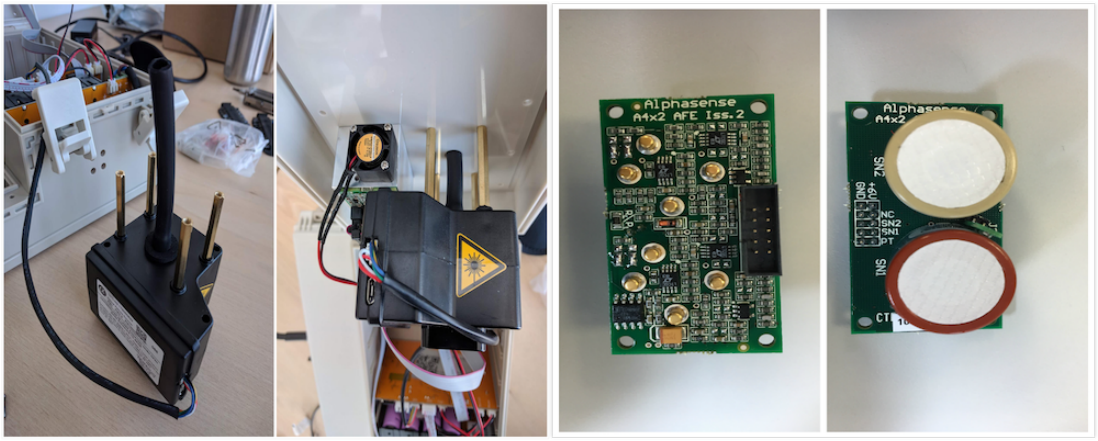
The air pollution data used in this project are collected by sensors called AirSpeck static wireless air quality monitor. Last year, six stationary air quality monitors were deployed around The Meadows in Edinburgh. In order to collect more air pollution data, another four stationary air quality monitors have been deployed around The

Meadows in June this year (See the position of sensors on the map in Figure 2.1). This section will explain how the stationary air quality monitors were set up.

To begin with, the data collected by stationary air quality monitor were sent to a server called AirSpeck platform through Global System for Mobile Communications (GSM) network. According to [10], both stationary air quality monitors and AirSpeck platform were equipped with General Packet Radio Service (GPRS) radio for data transmission. The stationary air quality monitor has two sensors which are responsible for collecting air pollution data. One is called Optical Particle Counter (OPC) which ouputs the detected concentration of PM1, PM2.5 and PM10. Another one is gas sensor which can measure two gases (NO_2 and O_3). (See Figure 2.2a and 2.2b)

Since the stationary air quality monitor will be deployed outside, they were designed to be resistant to different weathers and self-sufficient. A rain cover is attached on the top of the monitor and a solar cell is attached to the side of the monitor (See 2.3a, 2.3b, and 2.4a). Additionally, a rechargeable battery pack is charged by the solar panel during high solar incidence for power supply and a battery thermometer is installed to monitor the battery charge level (See 2.4b).

Figure 2.1: Ten sensors were available in this year around The Meadows and the campus; XXE001 - XXE006 were installed last year; XXE007 - XXE010 were installed this year; Figure provided by Sitthinut Kumpalanuwat [19]

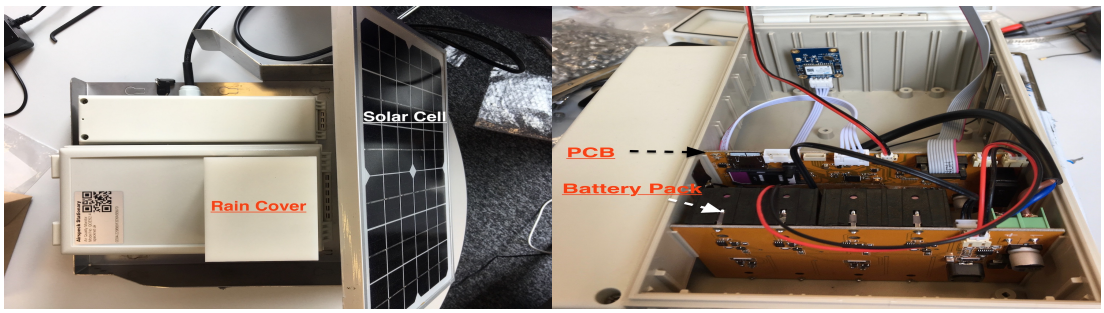


(a) OPC (b) GAS

Figure 2.2: OPC and gas sensor

(a) Rain cover (b) Solar cell

Figure 2.3: Rain cover for OPC and solar cell



(a) Attached rain cover and solar cell (b) PCB and battery pack

Figure 2.4: Attached rain over and solar cell; PCB and battery pack

2.3 Open Street Map

Open Street Map (OSM) is a project that contributes to create free geographic data for the world [35]. The data from OSM can be used in many ways including production of electronic maps which are similar to Google Maps. One popular map framework called Mapbox is developed based on OpenStreetMap data. Mapbox will be used to develop the Android mobile application which will be discussed later in section 4.4.2

OSM provides free editable maps with OSM format based on the XML format and can be downloaded by specifying the range (latitude and longitude) [2]. The map created by OSM presents physical features on the ground (e.g, roads, buildings) by attaching tags to its basic data structures (nodes and ways and relations)[12]. A node contains geo-spatial coordinates of a point with a unique ID. A way consists of a series of nodes. It has an unique ID, a name as well as the type of the way. Such features are very helpful when we want to extract the information that we need. For instance, a classification tag such as "highway=footway" means the current road is walkable for pedestrians. Tags and features can be used to filter ways in map files to obtain the specific roads.

2.4 Pathfinding

2.4.1 Problem Description

Pathfinding is related to the shortest path problem and is always popular in graph theory. Many fields including computer games [13] and transportation networks [11] use pathfinding methods to identify the best path which meets some criteria (cheapest, shortest etc) between two points in a large network [36]. This section will introduce the definition of pathfinding problem and discuss commonly used pathfinding methods.

The simplest pathfinding problem is to find the shortest path between two points in a graph. A directed graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Each edge $e = (u, v)$ between two points u and v has a corresponding non-negative weight $w(u, v)$. Normally, the weight $w(u, v)$ means the distance between u and v for finding shortest path problem. Hence, the total length of a path is the sum of each edge's weight belonging to the path. To find the shortest path from source s to a target t is to minimise the length of the path from s to t in a graph G where $s \in V$, $t \in V$ and $V \in G$.

Similarly, finding the cleanest between two points in a graph would be minimising the sum of the PM concentration of each node along a path. In this case, weight of the edge between two nodes would be the sum of the PM concentration of two nodes. Therefore, a new definition would be given a graph $G = (V, E)$ with a set vertices V and a set of edges E, an edge $e = (u, v)$ has an associated weight $w(u, v) = p(u) + p(v)$ where $p(u)$ and $p(v)$ means the PM concentration of node u and node v respectively. As mentioned before, PM2.5 will be the main criteria to measure the air quality. Thus, in this project, $p(u)$ and $p(v)$ are the value of the concentration of PM2.5 ($\mu g/m^3$).

Eventually, commonly used shortest path finding methods can be applied to solve cleanest-path problems by simply changing the definition of some parameters.

2.4.2 Baseline algorithms

Dijkstra is a commonly used graph-based pathfinding algorithms. Since in Dijkstra algorithm, the weight of edges will never be negative [36], it is very suitable for the solving many pathfinding problems including the cleanest-path problem. See the pseudocode of Dijkstra algorithm below where E is a set of edges and V is a set of vertices in a graph G.

Algorithm 1 Pseudocode of Dijkstra algorithm [38]

Input: G = (E,V) ▷ Weighted graph
Input: Source vertex $s \in V$
Input: all vertices $v \in V$
$dist[s] \leftarrow 0$ ▷ distance to source vertex is zero
for all v ∈ V-{s} **do**
    $dist[v] \leftarrow \infty$ ▷ set all other distances to infinity
$U \leftarrow \emptyset$ ▷ U, the set of visited vertices is initially empty
$Q \leftarrow V$ ▷ Q, the queue initially contains all vertices
while $Q \neq \emptyset$ **do**
    $u \leftarrow MinimumDistance(Q, dist)$
    $U \leftarrow U \cup \{u\}$ ▷ add u to list of visited vertices
    for all $v \in neighbors[u]$ **do**
        if $dist[v] > dist[u] + w(u, v)$ **then** ▷ if new shortest path found
            $d[v] \leftarrow d[u] + w(u, v)$ ▷ set new value of shortest path
return dist

A* is a variant of Dijkstra. The difference between A* and Dijkstra is that A*

additionally assigns the approximate distance between the current node v and the target node t to the weight w. So the weight of a edge $e = (u, v)$ would be $w(u, v) = d(u, v) + h(v, t)$ where $d(u, v)$ is the distance between u and v, $h(v, t)$ is the estimated distance between v and t, often known as heuristic function.

Algorithm 2 Pseudo-code of A* algorithm [21]

```
1:  Initialise: openList ← ∅, closedList ← ∅  ▷ Initialise both open and closed list
2:  starteNode.f = 0; openList.add(startNode)  ▷ Add the start node and leave it's f at zero
3:  while openList ≠ ∅ do                      ▷ Loop until we find the end
4:      currentNode ← leastFNode               ▷ the node with the least f value
5:      openList.remove(currentNode)
6:      closedList.add(currentNode)
7:      if currentNode == goal then
8:          return true                        ▷ Target found, backtrack to get path
9:      currentNode.children ← adjacentNodes   ▷ Generate children using adjacency nodes
10:     for child ∈ currentNode.children do
11:         if child ∈ closedlist then
12:             continue
13:         child.g = currentNode.g + distance(child, currentNode)
14:         child.h = distance(child, end)
15:         child.f = child.g + child.h
16:         if child.position ∈ nodes.position where nodes ∈ openList then
17:             if child.g > nodes.g then
18:                 continue
19:         openList.add(child)
```

The A* search algorithm is an example of a best-first search (BFS) algorithm which explores a graph by expanding the most promising node using cost function $f(n) = g(n)+h(n)$ where $g(n)$ is length of the path from the root to n and $h(n)$ is an estimate of the length of the path from n to the target node [32]. Greedy BFS uses the cost function $f(n) = h(n)$, which only contains heuristic function $h(n)$, estimating the closeness of n to the goal [28]. Thus, greedy BFS tries to expand the node that appears to be closest to the goal, without taking into account previously obtained knowledge (i.e. $g(n)$).

Chapter 3

Related Work

3.1 Heuristic function

The heuristic function $h(n)$ provides A* an estimate of the minimum cost from any node n to the target. This allows the heuristic function to control A*'s behaviour [23]. There are five possible situations that could happen:

- If $h(n) = 0$ through out the search, then A* will turn into the Dijkstra since only $g(n)$ plays a role. It would guarantee that A* can find a shortest path.
- if $h(n) < cost(n, target)$ through out the search, where $cost(n, target)$ is the cost of moving n to target, then A* will be able to find a shortest path. The smaller $h(n)$ is, the more node A* expands, making search progress slower[23].
- if $h(n) = cost(n, target)$ through out the search, then A* would only follow the optimal path without expanding any other node. This is the best situation. However, in the real world, it is unpractical to always find such $h(n)$. Still, there is the assurance that, given ideal information of $h(n)$, A* will behave perfectly.
- if $h(n) > cost(n, target)$ at some point during the search, then a shortest path is not guaranteed to be found by A*. As a trade off, A* could run faster to reach the target.
- The last extreme situation is that, if $h(n) \gg g(n)$ dominating the cost function, A* would turn into Greedy Best-First-Search(BFS).

Therefore, choosing proper heuristic functions for different cases becomes one of the most important part of developing A*.

3.2 A* variants

Since real-world problems are always complex, many variants of A* are derived for different use cases. Iterative deepening A*(IDA) is a variant of A* where the depth of the graph is a cutoff when the total cost $f(n) = g(n) + h(n)$ exceeds a given threshold. Normally, this threshold starts with zero which is the cost at the initial state, and would increase for each iteration during the runtime. Every time before next iteration, the threshold used for the next iteration will be updated with the minimum cost among all values that exceeded the current threshold [29]. Like A*, IDA* is guaranteed to find a optimal path, if the heuristic function $h(n)$ is admissible: $h(n) \leq cost(n, target)$. Since IDA does not remember any node except those belonging to the current path, the memory usage of IDA* is lower than A* [29]. In map path planning, the nodes are simply the coordinates which does not require much memory to store. However, a smaller algorithm runtime is very important for this project. Thus IDA* is not a appropriate choice for this project. But IDA* is still very useful for many special circumstances where the memory is limited but the optimal solution should be guaranteed. Another variant of A* is dynamic weighting A* which behaves like BFS at the beginning of the search by weighting the heuristic $h(n)$. This allows dynamic weighting A* to get anywhere quickly. At the end of the search, it becomes equally important to consider the $g(n)$ to get to the goal [24]. The total cost function of dynamic weighting A*:

$$f(n) = g(n) + w(n) * h(n)$$

where $w(n)$ is the dynamic weighting function. Normally, the value of $w(n)$ should decrease as the algorithm approaches to the target. This decreases the importance of $h(n)$ while increasing the relative importance of $g(n)$ which is the actual cost of the path. Dynamic weighting A* could explore less nodes than A* if $w(n)$ is set properly. This project will use dynamic weighting A* for planning least polluted route and compare its results with A*.

3.2.1 Pollution data

The pollution data used in this project were predicted by convolutional neural network (CNN) from Zoe Petard's previous work [25] as well as the CNN from Ryan Egan's project which aims to use online spatio-temporal model to predict air pollution with mobile and stationary data. [26].

Chapter 4

Methodology

4.1 Extracting OSM data in Python

A Previous project by Stefan lvanov [18] used Matlab to extract OSM data.

The Matlab library, developed by Ioannis Filippidis [3], can load osm data, parse OSM data and produce a connectivity matrix. At the beginning, the Matlab library was tried for this project. However, the function used for parsing OSM data was to slow to parse large OSM file which covered the main area of the city of Edinburgh. Even a smaller campus map was tried, but exporting the adjacency matrix to a text file consumed much memory.

Therefore, similar functions were implemented in Python. A library called Pyosmium was used in this project. Pyosmium can process OSM files in different formats. This library is a wrapper of the C++ library osmium and benefits from its fast implementation.

The process was fairly simple:

- Create an OSMHandler class which can extract OSM node and OSM ways from osm map file and append them to osm_node, osm_way lists respectively.
- Apply the osm map file to an OSMHandler object.
- Export osm_node list to a text file.
- Create an $N \times 1$ connectivity matrix where N means number of nodes and 1 means an empty list.
- Iterate through the osm_way list and append the index of the adjacency nodes to the K^{th} list in connectivity matrix where K means the index of the current node

in connectivity matrix.

- Export the index of all the nodes following with their lists in connectivity matrix (adjacency list) to text file. (E.g. 0: 4 5 means the 0^{th} node has adjacency nodes with index of 4 and 5)

Using an adjacency list was suggested by Stefan Ivanov [18]. However, in his project, the list was $N \times N$ binary matrix. When exporting the adjacency list to a text file, the non-zero indexes of each row needed to be found which required iterating through the whole matrix. This operation may not cause any problems when the size of map is small. However, as the map size grows, so does the number of nodes and the size of the matrix. Searching a large matrix using operations like ($numpy.where()$) is time consuming and will lead to an out of memory exception. Therefore, in this project, an $N \times 1$ matrix was used. Instead of using a binary matrix, storing the index directly to the list saves a lot of searching operations in large matrix. Exporting the adjacency list is just like exporting the nodes without any searching operation.

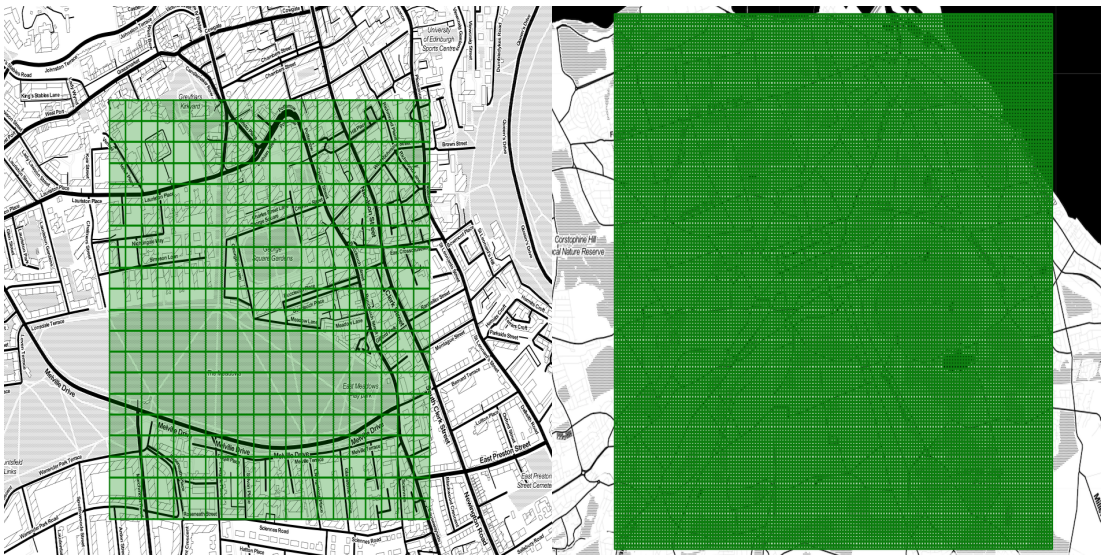
4.2 Dummy air pollution data

At the beginning of this project,the air pollution data used were the PM 2.5 concentration levels predicted by the CNN model from Zoe Petard's previous work [25].

The values of PM 2.5 concentration level were predicted based on six stationary air pollution sensors deployed last year around The Meadows as shown as in Figure 2.1. The prediction result covers the main area around the campus shown as in Figure 4.1a. Such air pollution data is enough for planning the path around the campus but not for a larger area like the city of Edinburgh. The area used in this project is covered by a 160×160 grid with approximate dimensions $9km \times 7km$, as seen in Figure 3.3. Thus, it is necessary to obtain more air pollution data which can approximately cover larger area.

4.2.0.1 Randomly generate the data

The most straightforward way to obtain more data is to randomly generate the air pollution data. However, one obvious problem for randomly distributed air pollution data is that the PM concentration level along the same road can have totally different value which is unreasonable. The path generated may suddenly turn right or turn left while it should go forward.



(a) 20×20 grids area around The Meadows (b) 160×160 grids area in Edinburgh

Figure 4.1: (a) Grids area around The Meadows; (b) Grids area cover the main city area of Edinburgh

4.2.0.2 Generative Adversarial Networks

Another method is to use Generative Adversarial Networks (GAN) [14]. Since the air pollution data for a small area is available, it is suitable to use existing data to generate fake data. The data generated by GAN preserves the distribution of original data for the most part. To do this, the first thing is to build generator (G) and discriminator (D) networks.

Table 4.1: Structure of G and D

Table 4.2: Structure of G

Layer (type)	Output Shape	Param
InputLayer	(None, 10)	0
Dense	(None, 200)	2200
Activation	(None, 200)	0
Dense	(None, 20)	4020

Table 4.3: Structure of D

Layer (type)	Output Shape	Param
InputLayer	(None, 20)	0
Reshape	(None, 20, 1)	0
Conv1D	(None, 16, 50)	300
Dropout	(None, 16, 50)	0
Flatten	(None, 800)	0
Dense	(None, 50)	40050
Dense	(None, 2)	102

The input for generator G is noise data which means that the dimension of the input layer of G is independent of the real original data. The output for G in this structure is 20 because the original PM data has dimensions with 20×20. The dimensions of the input layer in D match the dimensions of the output from G which is also (None, 20).

Then, a Generative Adversarial Network (GAN) can be built by combining G and D. Finally, training GAN:

1. Use G to sample and generate the data by feeding original real data to G; Pre training the discriminator D by fitting the data generated by G;

2. Use G to sample and generate the data by feeding original real data to G; Train the discriminator D with data generated by G.

3. Sample the noise uniformly between 0 and 1 with predefined dimension. Use the noise as input to train the GAN

4. Repeat the step 2 and 3 for $epoch$ times.

The data used for training has 19 samples predicted from Zoe's CNN model with kernel size 1 and learning rate 0.0005 [25]. Generating the fake data: Sample the noise uniformly between 0 and 1 with predefined dimension 20×20. Use the noise as input to trained generator G. The predicted result would be the fake data with dimension 20×20.

4.2.0.3 Comparing the fake data: GAN vs Random

Figure 4.2 shows the distribution of the original PM 2.5 concentration in a 20×20 grid area. Figure 4.3a shows the distribution of PM 2.5 generated by GAN. Figure 4.3b shows the distribution of randomly generated PM 2.5. Apparently, PM 2.5 generated by GAN has a similar distribution as the original PM 2.5 in both space and values. In Figure 4.2, the concentration level of PM 2.5 in the grids with latitude 10 to 20 and longitude 0 to 20 are fairly low while the grids with latitude 0 to 10 has higher PM2.5 concentration. This is reasonable because the the grids with latitude 0 to 10 are residential and commercial areas while the grids with latitude 10 to 20 are The Meadows which has better air quality. In contrast, the values of randomly generated PM 2.5 are spread sparsely from 0 to 1 independently of their location in the area making spacial data irrelevant.

Finally, after comparing the fake data generated by GAN and randomly generated data, it is easy to see that using GAN to generate fake data is more admissible. In order

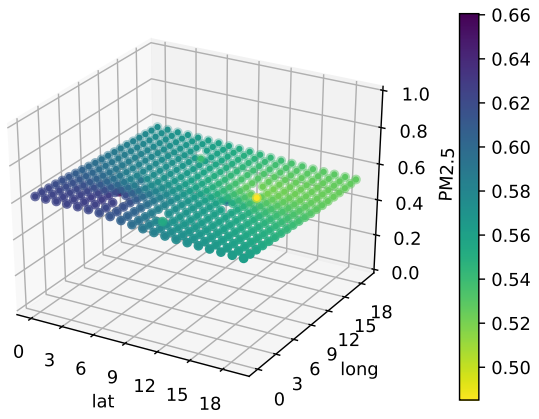
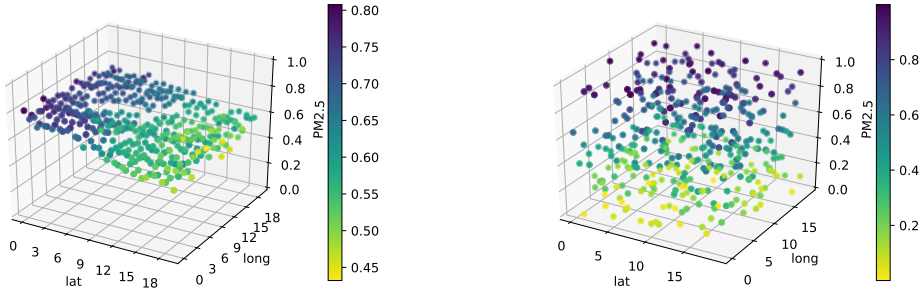


Figure 4.2: original PM 2.5 concentration level of 20×20 grids area



(a) GAN (b) Random

Figure 4.3: GAN/Randomly generated PM 2.5 concentration level of 20×20 grids area

to generate the PM 2.5 data for 160×160 grids, 64 of 20×20 grids of PM 2.5 data are needed. Using GAN, such 20×20 grids of PM 2.5 data can be generated for 64 times without repeating exactly the same one while a proper distribution of PM 2.5 data is guaranteed.

4.3 Algorithm implementation

In this section, five path planning algorithm are going to be discussed. These are Dijkstra, A*, Weighted A*, Dynamic weighting A* and Bidirectional A*. Before diving into algorithms, some important objects needs to be explained as below.

Since nodes and adjacency matrix have been extracted and stored in a text file as mentioned in section 4.1, they can be read from file and used to build a graph before

running algorithms. Thus, several objects are designed to read and store map data.

- Object "Point" is used to store the information of a node. A Point has several attributes: ID, Latitude, Longitude, PM1 value, PM2.5 value, PM10 value and weight. The weight of a Point is used to compare with other Points in a priority queue and can be set to either distance or pollution values.
- Object "Route" is used to store a list of points. A Route has several attributes: a list of points, routeString and distance. The routeString is mainly used for storing the JSON format of the list of points and will be returned to the client.
- Object "IO" can read data from files. The data types that IO object can handle includes nodes, adjacency matrix, grids and pollution data.
- Object "RoutePlanner" is responsible for:
    1. Initialising the graph: using IO to read data, initialising the cover tree space(a tree object used for finding the nearest points;)
    2. Running the path planning algorithms
    3. Returning the route string in JSON

Before running any path planning algorithm, the RoutePlanner is used to initialise the graph. Then the RoutePlanner will be able to receive the start point and the destination point to run algorithms as described below.

4.3.1 Shortest route planning: Dijkstra & A*

In this section, the implementation of Dijkstra and A* for shortest route planning will be described. In the meantime, routes generated by Google Maps will also be used to compare with the routes planned by Dijkstra and A*.

4.3.1.1 Distance Calculation

Normally, the distance in a straight line between two points can be seen as Euclidean distance in two dimension space. In this case, the distance between two points $p_1 = (x_1, y_1)$, $p_1 = (x_2, y_2)$ is given by:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

which is also equivalent to the Pythagorean theorem. In our case, x_1, x_2 can be seen as latitude and y_1, y_2 can be seen as longitude. If two points are very close to each other on the map, they can be seen as on a single plane. And thus, using Euclidean distance to estimate the distance is reasonable. However, such a calculation is not appropriate for two points that are not lying close to each other. To be more exact, the two points in our scenario are on a sphere (which is, Earth) or curved surface. Therefore, another formula called the Haversine formula [34] is presented to calculate the great-circle distance [33] between two points on the surface of a sphere. So, given two points $p_1 = (lat_1, long_1)$ and $p_2 = (lat_1, long_2)$ on a sphere S, the great-circle distance between them can be calculated as:

$$d(p_1, p_2) = 2r \times \arcsin\sqrt{\sin^2(\frac{lat_1 - lat_2}{2}) + \cos(lat_1)\cos(lat_2)\sin^2(\frac{long_1 - long_2}{2})}$$

where r is the radius of the sphere S (the Earth).

4.3.1.2 Implementing Dijkstra and A*

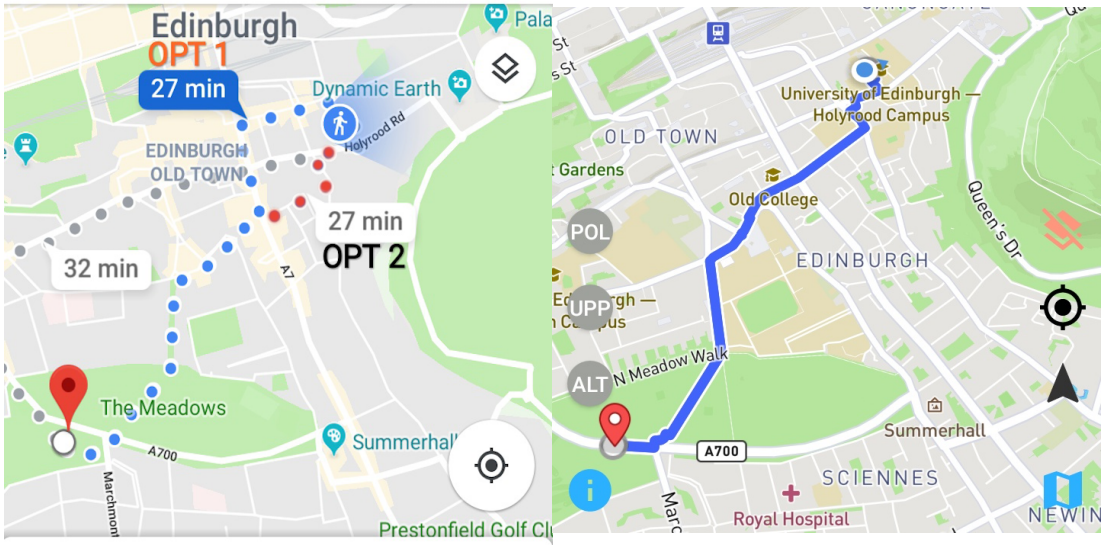
For shortest route planning, we assign the edges with distance calculated by Haversine formula during the run time. For A* algorithm, the heuristic function can be defined as the great-circle distance between the current point and the target point:

$$h = d(p, target)$$

where p is the current point and target means the destination point. So the heuristic function would never overestimate the cost from current point to the goal. One important method that is used in both Dijkstra and A* is a priority queue. Implementing Dijkstra based on a priority queue can save run time to $\mathcal{O}(|E| + |V|\log|V|)$ while the original Dijkstra algorithm has run time $\mathcal{O}(|V|^2)$ (where $|V|$ is the number of nodes, $|E|$ is the number of edges). Then, the steps described in section 2.4.2 in Background were followed to implement both Dijkstra and A*.

Once Dijkstra and A* were implemented, they were used to plan a route from a current location (9D Holyrood road) to a point near The Meadows. In order to evaluate the route generated by algorithm, two other routes marked as $OPT1$ and $OPT2$ in Figure 4.4a were generated by Google Maps for comparison. The $OPT1$ and $OPT2$ have the same duration as indicated in the Figure. The $OPT2$ is almost the same as the route generated by Dijkstra. This means that the algorithm perform nearly the same as Google Maps when planning for this route. However, this does not mean that

our algorithm can always generate the optimal routes. For now, the Dijkstra and A* algorithm succeed in planning a shortest route path.



(a) Google's

(b) My Dijkstra and A*

Figure 4.4: Google Map generated routes and my Dijkstra and A* generated route

4.3.2 Least polluted route planning: Dijkstra & A*

In this section, Dijkstra and A* are used to plan least polluted route. To evaluate our algorithms, a heat map for PM values will be used to show that the generated route avoids the areas where the the concentration of PM is high.

4.3.2.1 Dijkstra

Instead of assigning distance to edges, PM level is used as an alternative to distance. The weight of each edge is assigned by the sum of PM concentration level of two nodes.

To validate that Dijkstra can generate "least polluted" routes, a heat map was used to visualise the pollution. Additionally, a comparison was made between the least polluted route and shortest route. The main reason for doing this comparison is that in real-world conditions, the area where pollution level is high cannot always be avoided. So, one proper way is to inspect the difference between "least polluted" route. It is desirable to check the shortest route and whether "the least polluted" route has lower pollution level than the shortest route. Two points around The Meadows were chosen to generate the routes shown as in Figure 4.5.

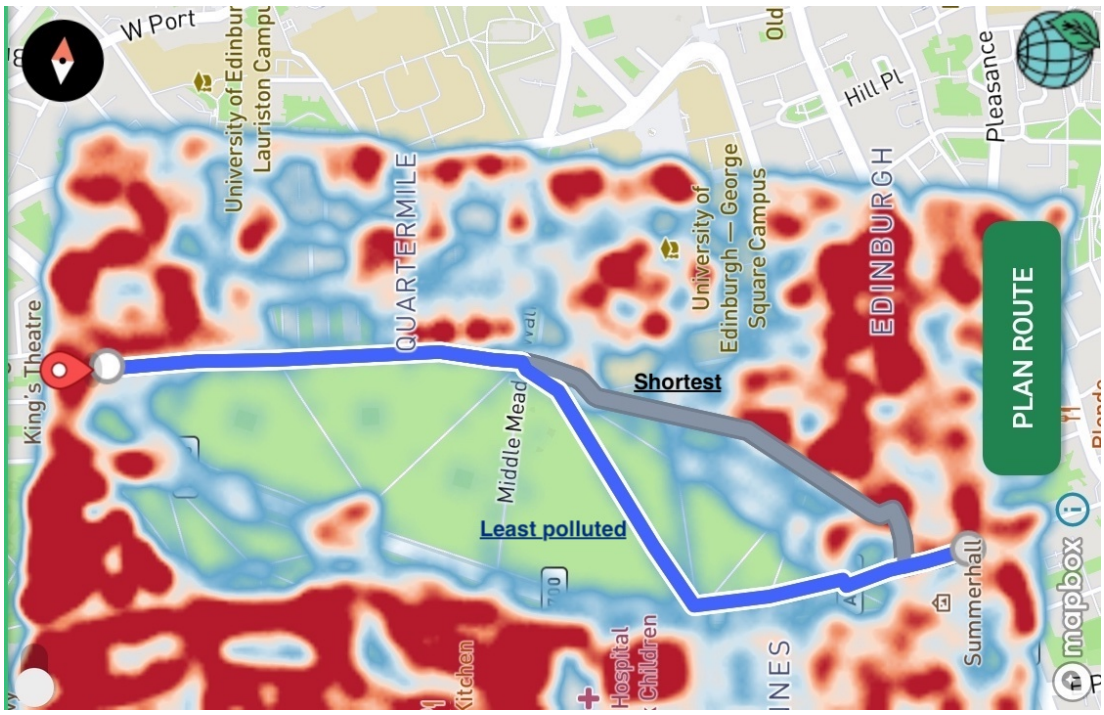


Figure 4.5: Least polluted route vs Shortest route in heat map

	Least polluted	Shortest
PM 2.5 ($\mu g/(m^3)$ per mile)	10.87	14.88
Distance (mile)	1.27	1.05

Table 4.4: Average PM 2.5 concentration level per mile and distance for least polluted route and the shortest route

In Figure 4.5, the two routes are different at the beginning and then converge at a point near Middle Meadow Walk. The rest of the route is the same after the intersection. It is obvious that the shortest route (in gray) passes an area where the pollution level is relatively higher than the area in The Meadows that the least polluted route passes. It is reasonable that the air condition in The Meadows would be better than the routes outside. For a more precise comparison, average PM concentration level per mile and the distance of two routes were listed in Table 4.4. The least polluted route has less PM 2.5 concentration level per mile than the shortest route. In the meantime, a longer travel distance is required for better air (even if you multiply the concentration per mile through with the miles, it's still less overall exposure to take the long route).

4.3.2.2 A*

Implementing A* algorithm for least polluted route planning has the same process as the implementation of shortest route as described in last section except for the cost function $g(n)$. Additionally, the dynamic weighting was added to h in order to investigate whether dynamic weighting improves the performance of A*.

The cost function of A* for least polluted route is:

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

where $g(n)$ is the accumulation of PM level from the current node n to the root and h(n) is the estimated accumulation of PM level from n to the target. In order to estimate $h(n)$ admissibly, it is necessary to find out how many nodes would remain for further steps to the target. The method suggested by the previous project [18] was used. It calculates the ratio between the number of nodes and the great-circle distance. Then the rest of number of nodes can be estimated by the remaining great-circle distance. Although, this method is not very accurate since different areas and distances may vary the result, this is the best solution for now. Eventually, the estimated ratio between the great-circle distance and the number of nodes in map is 1:42 after testing hundreds of least polluted routes by running Dijkstra algorithm. Once the ratio r and the average PM level p were obtained, the heuristic function is as follows:

$$h(n) = p \cdot r \cdot d(n, target)$$

where $d(n, target)$ is the great-circle distance between node n to target.

4.3.3 Weighted A*

Weighted A* allows A* to find a bounded optimal solution with less computational effort. The node evaluation function is defined as $f(n) = g(n) + w \cdot h(n)$, where the weight $w \geq 1.0$ is a parameter for scaling the value of the heuristic function.

According to [16], weighted heuristic search is most effective for search problems with close-to-optimal solutions, and can often find a close-to-optimal solution in a small fraction of the time it takes to find an optimal solution. In most cases, we want to find an optimal solution. However, in this project we want to suggest to the user alternative 'less polluted' routes which have higher pollution level but are shorter than the least polluted route. In this case, weighted A* may have pollution levels in between the shortest route and the least polluted routes.

4.3.4 Dynamic weighting A*

A* expands nodes in order of $f(n) = g(n)+h(n)$. For large problems, A* could quickly run out of memory because of a memory cost of $\mathcal{O}(n)$. Weighted A* expands nodes in the order of $f(n) = g(n) + \epsilon h(n)$ values, where $\epsilon > 1$ means A* tends to choose nodes that are closer to the target to expand. Therefore weighted A* can easily ignore some potential nodes to save time. Essentially, this algorithm trades off optimality for speed [20]. However, ϵ can be changed during the search. According to [27], the weighting factor ϵ is applied to the heuristic function to stimulate the search to be greedy. When the search is closer to the goal, the factor ϵ will be reduced. This is called dynamic weighting (see formula as below):

$$f(n) = g(n) + (1 + \epsilon \cdot (1 - \frac{D_n}{D_{goal}})) \cdot h(n)$$

where D_n is the cost of the current node and D_{goal} is the estimated cost from the root node to the goal. In this way, dynamic weighting A* initially behaves very much like a Greedy Best First Search (BFS), but as the search depth increases, the algorithm takes a more conservative approach, behaving more like the traditional A* algorithm. Although such an approach may improve the weighted A* or A*, it is not always guaranteed to find the optimal path. Instead, like weighted A*, dynamic weighting A* also trades off the optimality for speed. Therefore, setting the initial value of the weight factor ϵ becomes very important for dynamic weighting A*. For this project, different values of ϵ were tested for different roads in order to determine whether dynamic weighting actually made improvements and how the factor ϵ affects the search. See results and analysis in section 5.2.

4.3.5 Bidirectional A* search

The main reason to investigate Bidirectional search in this project is that the Bidirectional A* implemented by a previous project [18] hasn't been tested on a larger map and this is also an unsolved problem in previous project. So, in this section, Bidirectional A* will be introduced briefly.

Bidirectional search runs two searches: one starting from the starting node s to the target node t, and one from the target to the initial state. When two the searches meet, a good path should be found [31]. The idea behind bidirectional search is that a large tree would spread over the map during the search. Normally, a big tree occupies much

more space than two small trees [24]. So, in most cases, bidirectional search could prevent the search from expanding too many nodes [17].

Bidirectional search merges the two searches together after finding the meeting point. If the forward and backward searches run simultaneously and independently (without any cooperation), the first meeting of two searches, in most of time, does not guarantee the optimal solution [18]. Although, for each search, it can find the optimal path to the "meeting point", when combining the results of two searches, the path may not be optimal from the initial state to the target. Therefore, retargeting the goal of each search is needed [24]. A simple retargeting approach is to update the searches after the first meeting [18], see steps as follows:

1. Run two searches based on Dijkstra simultaneously (forward search start from s, backward search start from t) to find a meeting point m.
2. Store the sub-optimal paths for two searches: $forward(s \text{ to } m)$ and $backward(t \text{ to } m)$. So current optimal path would be: $forward + backward$
3. Check the remaining nodes for both forward search and backward search whether a shorter (better) path exists.
4. Update the $forward$ and $backward$ if necessary

Another retargeting approach is to update the candidate meeting point P dynamically during the search [24]. To do this, performing a forward search towards the target first for several steps to find a best forward candidate point P_f, and then running a backward search towards the P_f for several steps so that it can find a best backward candidate point P_b. Then running the forward search again but towards the P_b. Repeating such process until two candidates P_f and P_b are at the same point. This method requires to choose candidate points carefully which could directly determine the results.

Eventually, the first retargeting approach discussed above was used for this project which is proposed by Stefan [18]. Again, the main purpose is to test this algorithm in a lager map which is an unsolved problem in previous project.

4.4 Server and Mobile application

So far, the implementation of path planning algorithms and the design of objects have been discussed at the beginning of section 4.3. This section introduces the implemen-

tation of the server which is responsible for running path planning algorithms and the design of a mobile application.

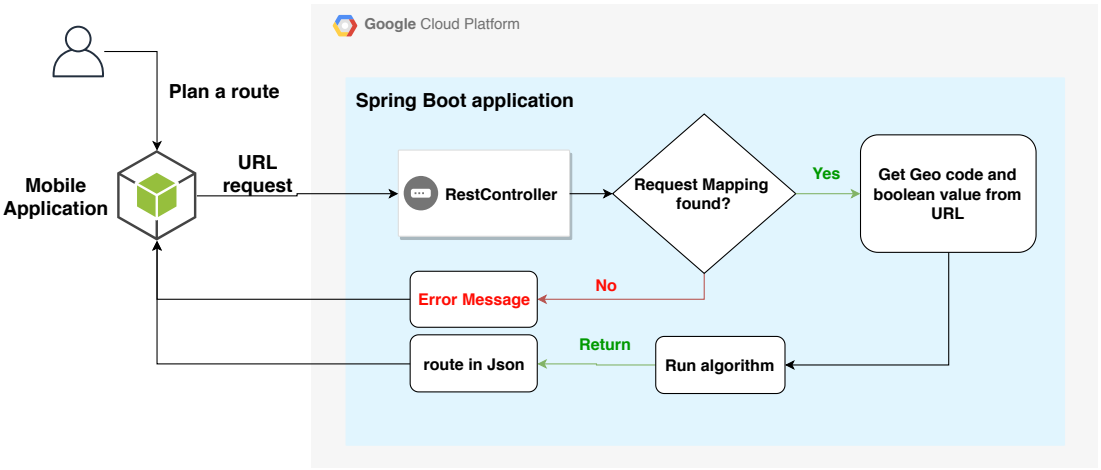


Figure 4.6: Overview of the work flow between server and mobile application

4.4.1 RESTful Web service

Representational state transfer (REST) is a architectural style that defines several constraints which can be applied to a web service and enable the service to work best on the web [37]. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links to the Web [22]. Thus, requesting resources deployed on cloud is convenient through RESTful service. Since the algorithm is developed by Java, implementing RESTful service using Spring Boot integrates well with algorithms.

Eventually, a Spring Boot application which provides a RESTful web service was developed. When starting the spring boot application, a RESTController will be initialised first. Then a global RoutePlanner will be created which will initialise the graph immediately. The RESTful web service provides a HTTP-request handler (currently) which is responsible for dealing with path-planning requests (See URL format as follow):

$$root/routes/sLat/sLong/tLat/tLong$$

where sLat and sLong in bracket means the latitude and longitude of the origin, tLat and tLong means the latitude and longitude of the target. When any request comes in, the HTTP-request handler (the RestController in Figure 4.6) checks the format of URL as well as the parameters of the URL. If the request matches, the parameters will

be parsed and sent to the RoutePlanner object which can run path planning algorithms. Otherwise, an error message will be returned. If the RoutePlanner finds some routes, the result will be returned in JSON to the client (See also in Figure 4.6).

Considering the server should be able to respond client quickly and consistently, the Spring boot application was deployed on Google Cloud App Engine. The Google Cloud App Engine provides a flexible Java environment which can automatically scale app up and down while balancing the load. It is convenient and efficient since the only concern is the development of the Spring boot application itself and the Google cloud App Engine will handle the whole life cycle of the Spring boot application. The spring application will be wrapped into a jar file which includes all resources and classes. The main application run on the Google Cloud App Engine is this jar file.

4.4.2 Android mobile application

An Android mobile application (as shown in Figure 4.6) was developed as a client for people to use. The main functionality of this application is displaying an interactive map which allows user to plan least polluted routes to different places.

The application works as follow:

1. Initialisation:
   (a) Initialise the map when the application is started: create a map view and enabling the map to access device's location
   (b) When map is ready, initialise the widgets (buttons, a search bar and information tables for air pollution)
   (c) Find the last known (current) location of the device and locate it. The application is ready to use
2. Send HTTP-request to server when a user requests least polluted routes
3. Wait for server's response. If a message is received in JSON format, parse the JSON to retrieve a list of points. Otherwise, an error message will be displayed to the user.
4. Construct the list of points to routes and displaying the routes on the map.

The user interface of the application as well as the functionality of widgets will be discussed in detail in section 5.3.

Chapter 5

Results

5.1 Least polluted route result

This section evaluates algorithms for planning least polluted routes. Since previous project [18] encountered problems of extracting nodes and adjacency matrix from large city map and ended up with running algorithms for small campus map, testing algorithms on a larger map becomes very important and necessary in this project. In order to compare algorithms more precisely, algorithms were tested by three experiments which include long distance, middle distance and short distance respectively. Each experiment contains three pairs of points which are distributed sparsely on the map so that algorithms can be tested in different areas in Edinburgh. More importantly, the performance of algorithms will be evaluated by run time, pollution level, distance, and the number of expanded node. So for each pair of points, each algorithm will be run for 100 times to calculate the average values of run time.

To begin with, 18 points (nine pairs) were selected and drawn on the map as shown in Figure 5.1 for easier observation. The largest points represent the long distance experiment and the smallest points for shortest distance experiments. In Figure 5.1, the name on the top of a point indicates the experiment information:

- The first character of the name indicates the distance(L: long distance experiment, M: Middle distance experiment, S: Short distance experiment)
- The second character of the name: S means the start point and T means the target
- The third character of the name means the i^{th} pair of the points in a experiment.

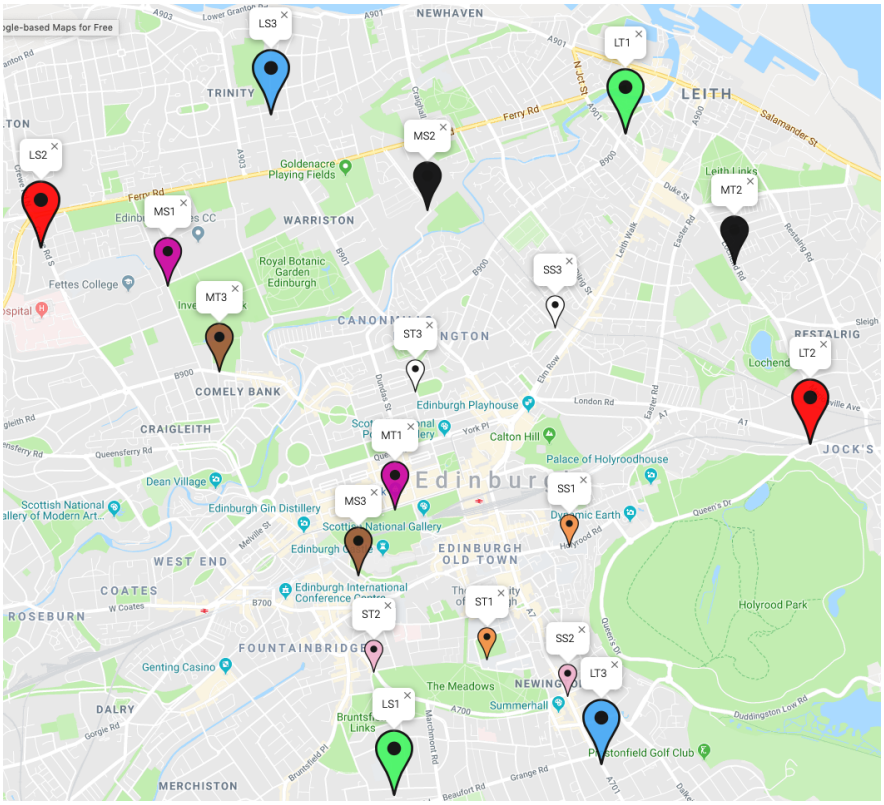


Figure 5.1: Paris of points for nine experiments in the map

5.1.1 Experiments

For the first experiment (Long distance), three pair of points which have red, blue and green colours shown in Figure 5.1 were used to test Dijkstra, A* and Bidirectional A* (Bi-A*). For each pair of points, the least polluted routes generated by the three algorithms are the same because they are deterministic. Such results are very similar to the previous project [18]. See PM 2.5 concentration results of three routes in Table 5.1.

By looking at the first figure in Figure 5.2, for three routes, the Dijkstra always performs better than A* based on runtime. This result is reasonable because A* needs to calculate the value of the heuristic function for every neighbour. In our case, the heuristic function is Haversine formula which contains lots of sin and cos calculations which are computationally expensive. As for Bidirectional search, the Bi-A* always performs better than A* based on runtime. As Bi-A* runs two A* algorithm simultaneously, it is reasonable that Bi-A* can run faster than A*. However, Dijkstra outperforms Bi-A* based on runtime. This can also be explained by the heuristic function which takes time to compute.

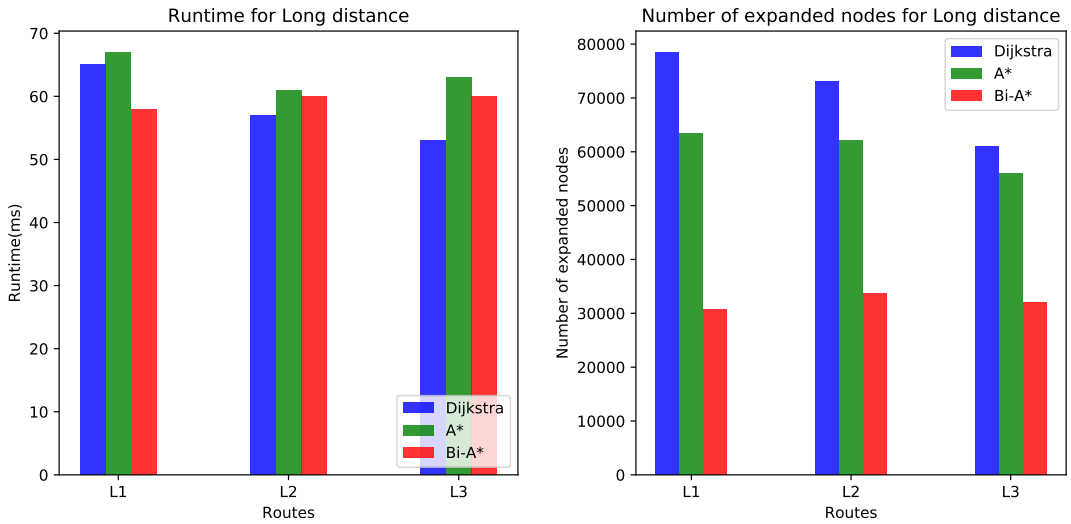


Figure 5.2: Comparison of the runtime and the number of expanded nodes of three algorithms for experiment 1: Long distance; L1 means the from LS1 to LT1, and etc

From	to	Dijkstra(PM2.5)	A*(PM2.5)	Bi-A*(PM2.5)	Distance(mile)
LS1	LT1	8.48	8.48	8.48	7.06
LS2	LT2	9.58	9.58	9.58	8.16
LS3	LT3	7.87	7.87	7.87	8.74
MS1	MT1	8.36	8.36	8.36	3.86
MS2	MT2	9.66	9.66	9.66	4.37
MS3	MT3	10.11	10.11	10.11	3.47
SS1	ST1	15.33	15.33	15.33	1.49
SS2	ST2	10.87	10.87	10.87	1.27
SS3	ST3	10.11	10.11	10.11	1.70

Table 5.1: Distance and PM 2.5 $((\mu g/m^3)/mile)$ value of least polluted routes for three experiments

Since the runtime of each algorithm can be affected by the performance of the computer while the the number of expand nodes remains same, evaluating the number of expanded nodes of the algorithms further becomes necessary. By comparing the second plot in Figure 5.2, A* expanded less nodes than Dijkstra for each routes. Dijkstra doesn't use a heuristic function to expand the node in the frontier. It looks only for the path that minimises the cost to reach an un-visited node directly connected to the nodes already visited. It will find the shortest path finally, but would have to explore all the

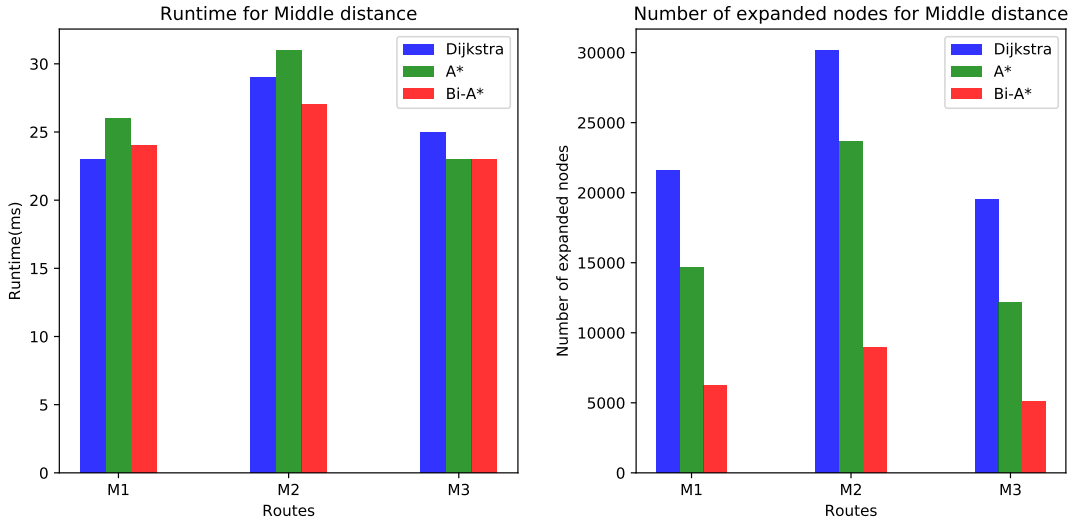


Figure 5.3: Comparison of the runtime and the number of expanded nodes of three algorithms for experiment 2: Middle distance; M1 means the from MS1 to MT1, and etc

nodes not directly connected to the sink. According to [20], A* with the help of a good heuristic function which never overestimates the cost to reach the goal always expands the correct node at the frontier. The Bidirectional search expanded fewer nodes than A*. If algorithms are running on a computer where the calculation of heuristic function costs only 1/100 of total runtime, under this circumstance, A* and Bi-A* would perform better than Dijkstra. On the other hand, A* and Bi-A* which expanded less nodes would also cost less memory than Dijkstra.

For the second experiment (Middle distance), three pairs of points with black, purple and brown colours shown in Figure 5.1 are used to test Dijkstra, A*and Bidirectional A* (Bi-A*). Similarly, for each pair of points, the least polluted routes generated by algorithms are the same because they are deterministic (See Table 5.1). The first plot in Figure 5.3 shows that the runtime of A* is very close to Dijkstra's and from MS3 to MT3, A* even performs better than Dijkstra based on runtime. As before, Bi-A* performs better than A*. As for the number of expanded nodes, Bi-A* expanded the least number of nodes while Dijkstra still expanded the most. It seems that if we evaluate three algorithms by combining runtime and expanded nodes we can see that, with the decreasing number of expanded nodes, the run time of A* would be closer to Dijkstra's. One potential reasons would be that the cost of calculating heuristic function does not dominate the runtime.

For the last experiment (Short distance), three pairs of points with white, orange and pink colours shown in Figure 5.1 are used to test Dijkstra, A*and Bidirectional A*

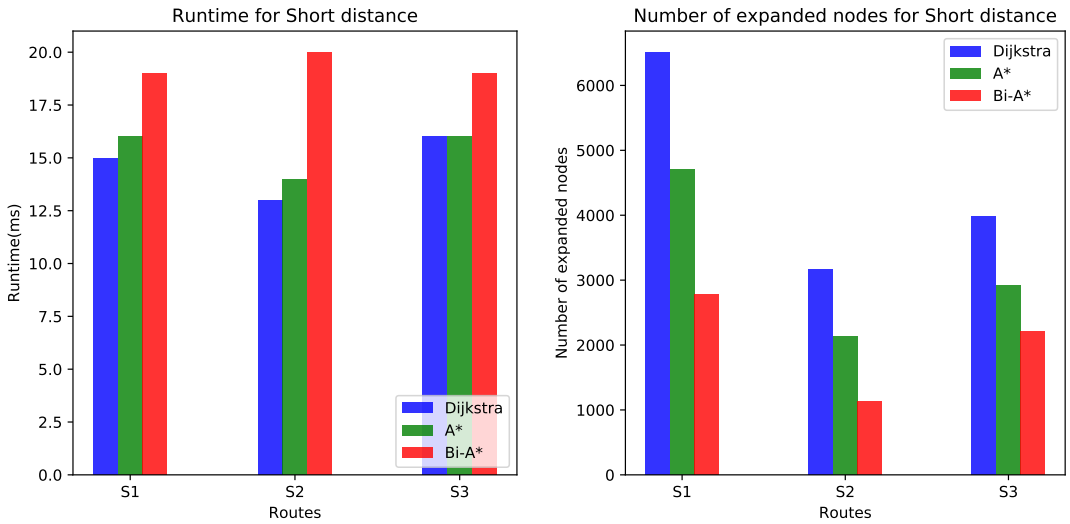


Figure 5.4: Comparison of the runtime and the number of expanded nodes of three algorithms for experiment 3: Short distance; S1 means the from SS1 to ST1, and etc

(Bi-A*). Again, for each pairs of points, the least polluted routes generated by the three algorithms are the same because they are deterministic. Based on runtime (in Figure 5.4), A* and Dijkstra performs very closely to each other while Bi-A* performs worse than A*, and of course, also worse than Dijkstra. And we can see that the number of expanded nodes of A* and Bi-A* are not significantly different anymore. Running two A* algorithm at the same time and expanding less nodes in short distance does not help Bi-A* perform better than A*.

5.1.2 Summary

To sum up, based on runtime, A* performs similarly to Dijkstra when the distance is short. When the distance increases, calculating the heuristic function which contains lots of computation on sine and cosine is likely to dominate the runtime of A*. So, the runtime of A* would be longer than Dijkstra when the distance increases. Bidirectional search could speed up the A* in search progress for long distance while the improvement is not significant when the distance is short. As for the number of expanded nodes, A* always expanded fewer nodes than Dijkstra with the help of an admissible heuristic function. Because of running two A* at the same time, Bi-A* actually expanded nodes in two smaller trees (as described before in section 4.3.5), thus, Bi-A* expanded fewer nodes than A*. Overall, Bi-A* performs better for all distances based on the number of expanded nodes and Dijkstra performs better based on runtime.

Since real-world problems are complex, the trade-off between run time and memory should be decided in many cases. Thus, investigating further in both bidirectional A* and Dijkstra is worthwhile for future development.

5.2 The result of Dynamic weighting A*

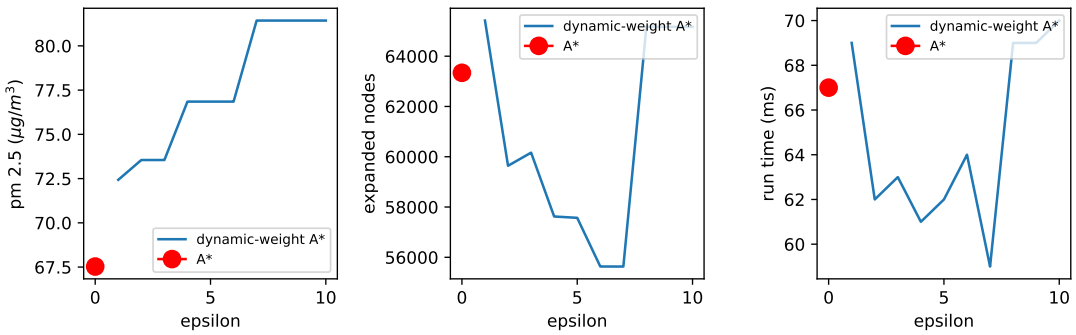


Figure 5.5: The test result of Dynamic weight A* planning from LS1 to LT1: Total PM 2.5 concentration, the number of expanded nodes and rum time

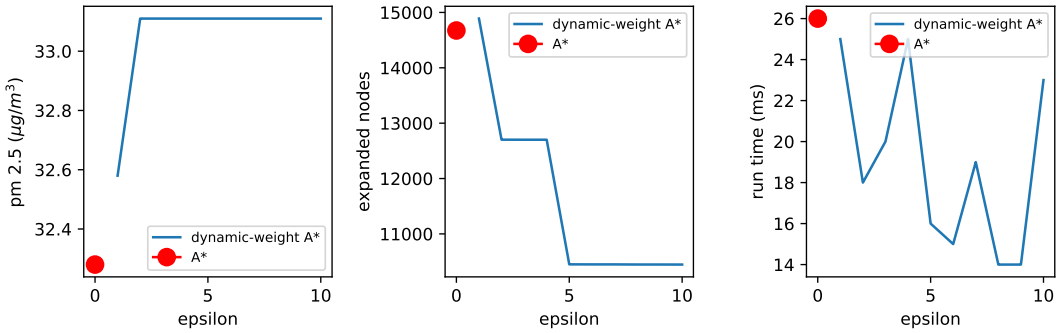


Figure 5.6: The test result of Dynamic weight A* planning from MS1 to MT1: Total PM 2.5 concentration, the number of expanded nodes and rum time

Figure 5.5, 5.6 and 5.7 above show how varying the value of ϵ from 1 to 10 affects the performance of dynamic weighting A* in planning three routes with different distance. The first Figure with y-label set as PM2.5 in Figure 5.7 show that for this short-distance case, dynamic weighting A* with any value of ϵ is guaranteed to find the optimal result. By looking at the third figure in Figure 5.7, there is no obvious trends of runtime. One point to note is that the smallest runtime is 9 when ϵ lie between 6 and 8. The second figure in Figure 5.7 shows that the ϵ between 6 and 8 also

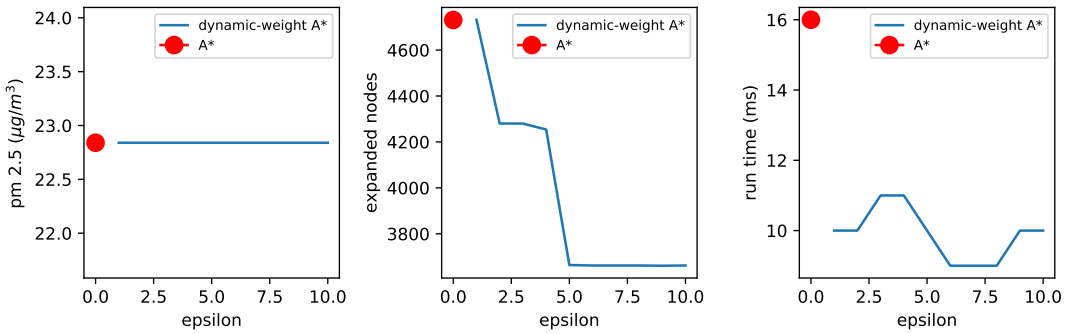


Figure 5.7: The test result of Dynamic weight A* planning from SS1 to ST1: Total PM 2.5 concentration, the numebr of expanded nodes and rum time

corresponds to the lowest number of expanded nodes. With increasing value of ϵ, the number of expanded nodes is going to decrease and always smaller than the number of nodes expanded by A*. As mentioned earlier in Section 4.3.4, dynamic weighting A* initially acts like BFS which speeds up the search progress. With larger ϵ, dynamic weighting A* would behave more greedily in order to reach the goal very fast. This is why fewer nodes are expanded by dynamic weighting A*.

However, as described in section 4.3.4, such an approach can easily miss out the optimal solution (See the first figures in Figure 5.5 and 5.6). For long distance and middle distance, dynamic weighting A* cannot find an optimal solution which has lowest pm 2.5 concentration with value 67.5 $\mu g/m^3$ and 32.28 $\mu g/m^3$ respectively for long distance and middle distance. This is because dynamic weighting A* trades off the optimal solution for speed, in this case, the number of expanded nodes. The second figures in Figure 5.5 and 5.6 shows the fact that with less expanded nodes, the value of PM2.5 concentration of sub-optimal solutions tend to increase. As for runtime, the runtime of long distances act similarly to the number of expanded node with some notable differences. The runtime shown in three figures are all smaller than the runtime of A* when the number of expanded nodes are smaller than A*'s.

To conclude, dynamic weighting A* could speed up the search by expanding fewer nodes during the runtime while the optimal solution can be guaranteed if ϵ is set properly. But when ϵ does not fit to the problem or is relatively high, dynamic weighting A* will trade off the optimal solution for increased speed like weighted A*. A strategy of setting ϵ can be summarised from three Figures (5.5, 5.6, 5.7):

- When estimated distance (great-circle distance) between two places is short (0km $\sim$ 3km), relatively higher $\epsilon > 5$ could help reduce the runtime and the

number of expanded node while the optimal solution is guaranteed

- when distance between two places is longer ($> 3km$), smaller $\epsilon < 1$ should be considered to guarantee the optimal solution. If close-to-optimal solution is acceptable, choosing ϵ between 5 and 6 is admissible for longer distance.

5.3 User interface

This section introduces the functionality of the widgets in this application. In Figure 5.8, the application contains seven buttons on two sides, a search bar on the top and a table along on bottom. On the right side, the first button (a pink icon) is used to clear the generated route on the map. The second black button (a GPS icon) is used to locate the device. The third button (a black navigation icon) can start planning the route when a destination is selected. The last blue button (a map icon) is used to show heat map of a route. On the left side, the first three buttons is mainly used for testing the functionality of the server. The buttons with texts "POL", "UPP" and "ALT" tell the server: "POL": Button ON tells server to plan the least polluted route; OFF means planning the shortest route. "UPP": Button ON means updating the pollution data; OFF means using current pollution data."ALT": Button ON means generating an alternative route.

When a location was selected in a list of results in the search bar, the view of the map will zoom in at the selected location (See the third picture in Figure 5.8); Once a destination is decided, a route will be generated by pressing the navigation button and the information of the route (PM, distance) will be displayed in a table on the bottom. This table can be shown/hidden by pressing the last blue button on the left(an exclamation mark icon).

5.4 Alternative 'less polluted' routes

As mentioned before in section 4.3.3, weighted A* can be used to generate close-to optimal routes. Thus, in this section, three routes along with alternative routes were generated with $\epsilon = 20$. The reason for setting ϵ to 20 is that with smaller ϵ, in short distance, the weighted A* generates the same route as A* in most of time. But it would be better that the alternative route is different to the optimal route. With higher ϵ, the weighted A* can guarantee to generate different route in most of time. In Figure 5.9, the gray routes are the alternative routes. The alternative routes from 9D

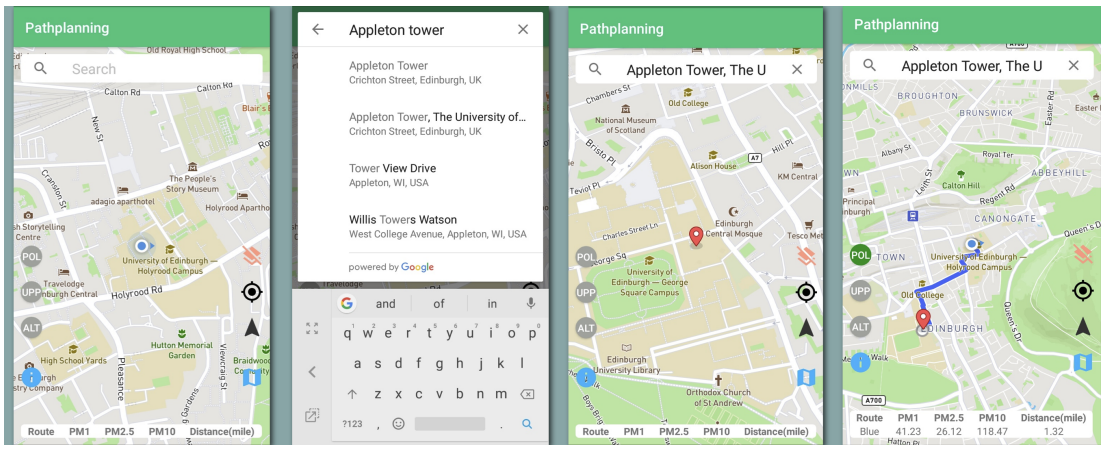


Figure 5.8: User Interface of the Android mobile application

Holyrood road to Princes Street and North Bridge have higher PM2.5 concentration but shorter distance than the optimal routes. Such alternative routes could provide the user another options: less polluted or shorter distance. But the alternative routes cannot guarantee the shorter distance. The alternative route from 9D Holyrood road to the Appleton Tower has higher PM2.5 concentration as well as long distance than the optimal solution. It is hard to ensure that the alternative routes are the sub-optimal routes. Generating the sub-optimal routes could be a potential area to be investigated in future. For now, the application aims to provide such service (function) which could be improved with better alternative route suggestions.

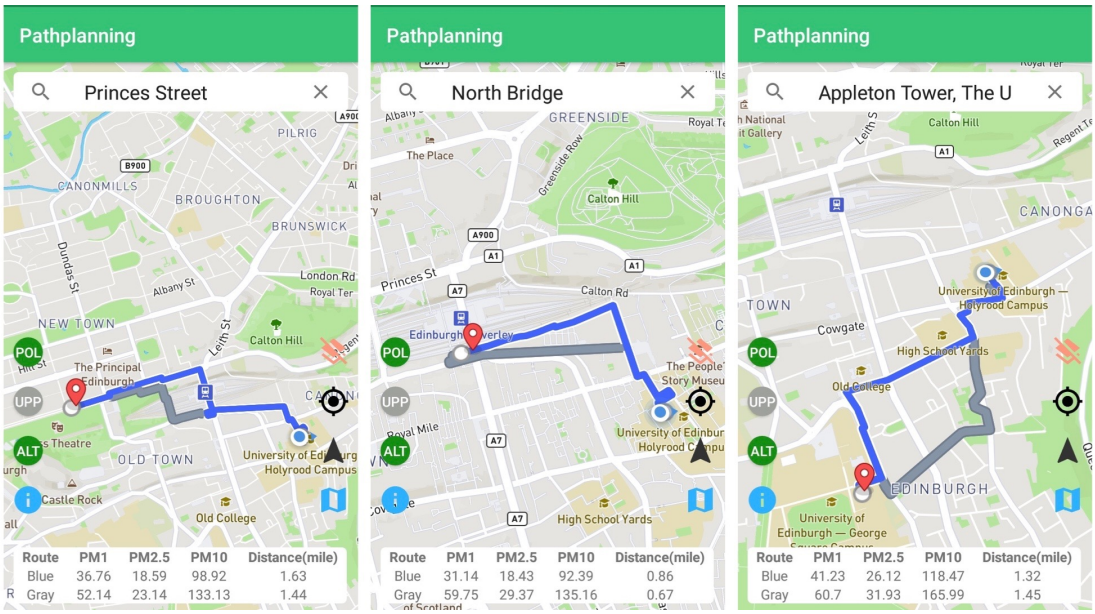


Figure 5.9: Least polluted routes with alternative routes generated from 9D Holyrood road to Princes Street, North Bridge and the Appleton tower

5.5 Using the latest pollution data

5.5.1 Server side preparation

Since the pollution data will be updated during the day, the server (least-polluted-route server) should be able to follow the latest data. Currently, the pollution data used by the least-polluted-route server is stored locally in the package (the jar file). Once the application is deployed on the Google Cloud App Engine, there is no other way to modify or operate the jar file. It turns out that the only feasible way is to implement another sever (pollution-data server) which can receive live pollution data and send it to the least-polluted-route server. Additionally, the pollution-data server should be able to run CNN models to predict pollution data in grids and use GAN to generate pollution data for larger areas if it is necessary. The interaction between two servers should be as follows:

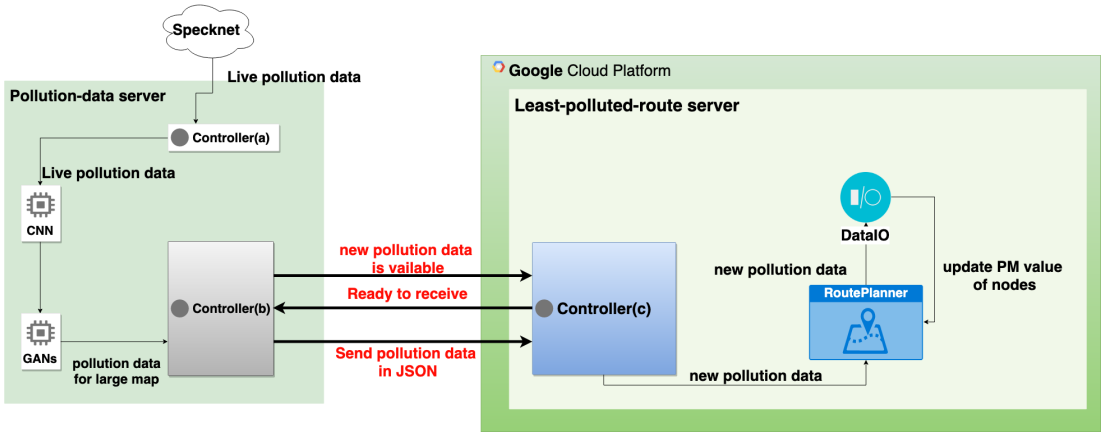


Figure 5.10: The interaction between pollution data server and least-polluted-route server

In short, when pollution-data server receives live pollution data from Specknet (a server which stores live data), it will process the data through CNN models and GANs to generate new pollution data for large map. If the new pollution data is ready, the pollution-data server send a message to notify the least-polluted-route server that the latest pollution data is available. The least-polluted-route server should then respond it whether or not to receive the new pollution data. If the respond is positive, then new pollution data will be sent in JSON to the least-polluted-route server. Eventually, the new pollution data will be applied to nodes.

Due to limited time, this project was unable to implement the pollution-data server. However an interface was implemented in least-polluted-route server for interacting

with pollution-data server so that in future, this project can be easily extended.

5.5.2 Using pollution data during a day

Although the pollution-data server is not available yet, it is necessary to test the algorithms with live data. As an alternative, the pollution data during a day on 2019-07-24 was predicted by convolutional neural network for a grid of 20×20 cells around The Meadows provided by Egan [26]. A 20×20 grid of pollution data were generated for each hour. This pollution data was stored locally and updated manually.

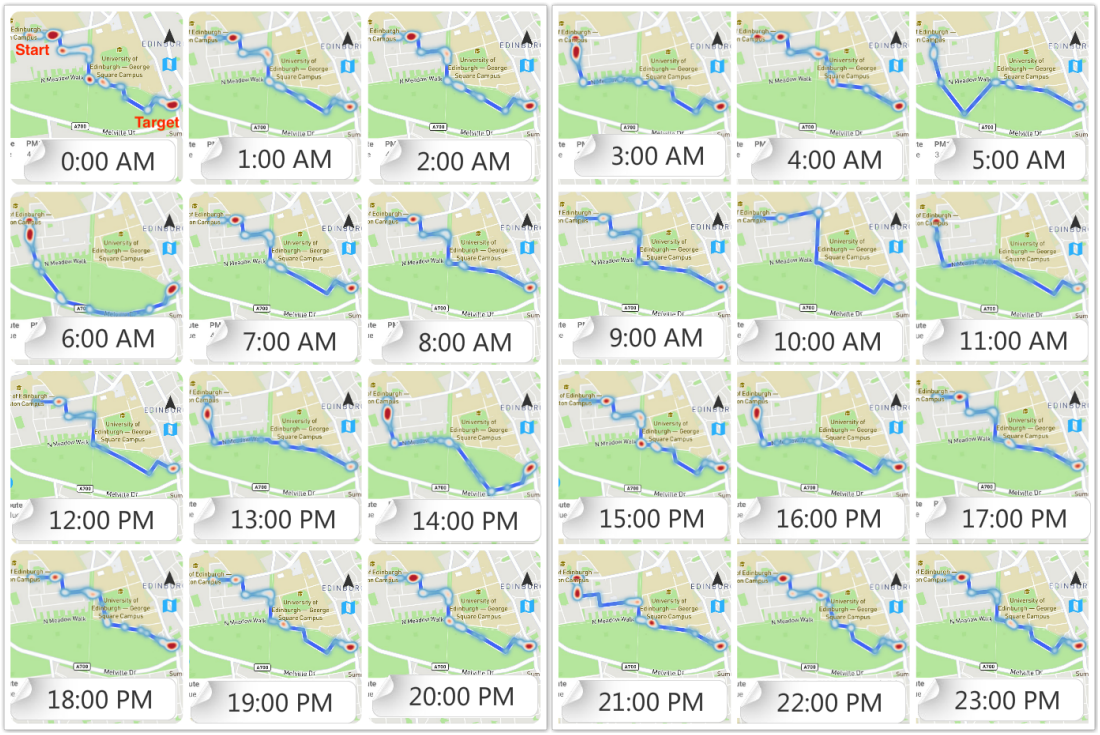


Figure 5.11: Least polluted routes (from The University of Edinburgh, Lauriston campus to Hope Park Cres (near by The Meadows)) with heat maps of the PM2.5 concentration of the routes generated during a day (2019-07-24)

To observe how the algorithms would react to the live pollution data, a single route from The University of Edinburgh, Lauriston campus to Hope Park Crescent (near by The Meadows) was generated at different times to check whether it would change from hour to hour. This can be seen in Figure 5.11. In addition, hourly pollution in PM2.5 particles of the least polluted route as well as the pollution for the shortest route during the day are plotted in Figure 5.12. Correspondingly, the distance of the least polluted route at each hour are shown in Figure 5.13.

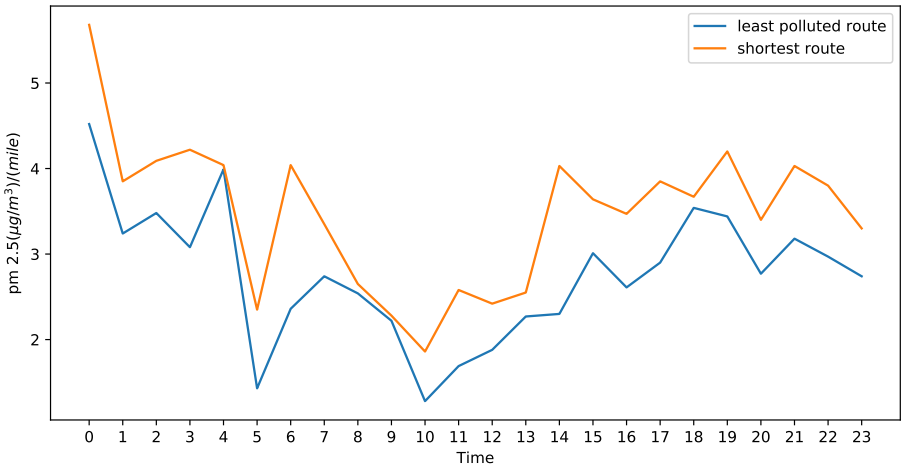


Figure 5.12: The hourly PM2.5 of the least polluted routes and the shortest route from The University of Edinburgh, Lauriston campus to Hope Park Crescent (near by The Meadows)

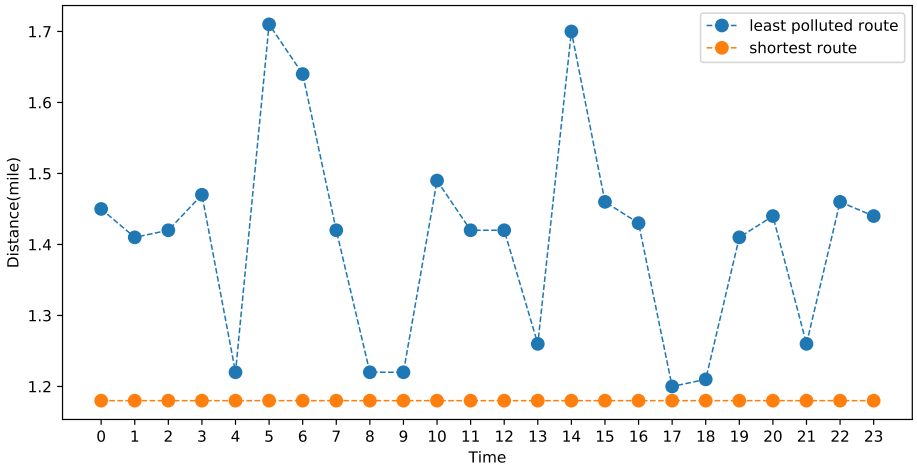


Figure 5.13: The hourly distance of the least polluted routes from The University of Edinburgh, Lauriston campus to Hope Park Crescent (near The Meadows); The distance of shortest route is 1.18 miles for reference

By analysing the three figures mentioned above, many insights can be observed. Obviously, the least polluted route is not constant. In Figure 5.11, the heat map of the least polluted routes shows that the most polluted places of the route are the road near The University of Edinburgh, Lauriston campus and Hope Park Cres near The Meadows during the day. It is reasonable that both high pollution points are near

traffic lights where many vehicles could wait at the stop light during the day. The change of the least polluted routes could reflect the pattern of pollution distribution. In Figure 5.12, from 5AM to 23PM, the average PM2.5 concentration of the least polluted routes reaches the peak at 18PM. This could be explained by that 18PM is rush hour and the least polluted route passes many roads where may have relatively heavy stream of people (near by the main library and The Meadows walk) and vehicles (See the route at 18PM in Figure 5.11.

The PM2.5 concentration of the least polluted routes and the shortest route are very similar at 4AM and 9AM in Figure 5.11. In order to analyse the shortest route and the least polluted route at 4AM and 9AM further, heat maps of PM 2.5 concentration along the least polluted routes and shortest route at 4AM and 9AM were generated and shown in Figure 5.14. At 4AM, the least polluted route and the shortest route are very similar. The different part of two routes is that the least polluted route passes the road near the George Square while the shortest route follows Middle Meadows walk. This causes the little difference of average PM2.5 concentration between two routes. At 9AM, the least polluted route and the shortest route are almost the same except that the least polluted route enters Middle Meadows walk a bit earlier than the shortest route. Such observation indicates that the shortest route could be very similar to the least polluted route sometimes during a day. However, most of time, the shortest route is much higher the least polluted route. In Figure 5.12, the PM2.5 concentration of the shortest route and the least polluted route have big gaps at 6AM and 14PM. By looking at the least polluted routes at 6AM and 14PM in Figure 5.11, the least polluted routes are significantly different with the shortest route which is constant (see the shortest route in Figure 5.14). The least polluted routes at 6AM and 14PM avoid the roads on Lauriston PI by passing through Chalmers St. It is possible that more vehicles pass Lauriston PI than Chalmers St at that time. This evidence not only shows that the least polluted route can avoid some places with relatively higher PM2.5 concentration, but also indicates that the air quality of Chalmsers St is better than Lauriston PI at 6AM and 14PM.

Another insight from Figure 5.12 is that the PM 2.5 concentration of two routes have similiar trends. As discussed above, the places near the library, Lauriston campus and Hope Park Cres (near the crossroads) have highest PM2.5 concentration. Those places are frequently passed by the least polluted routes and the shortest route. The PM2.5 concentration of those places might dominate the total PM2.5 concentration of the routes, thus, leading the trends of the total PM2.5 concentration of the routes during

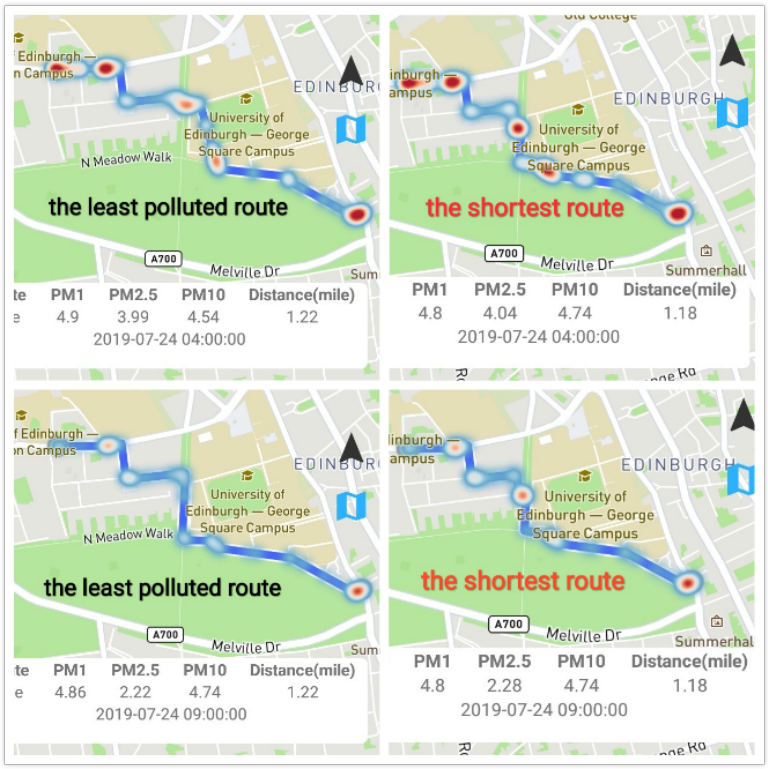


Figure 5.14: The least polluted routes and the shortest route at 4AM and 9AM

the day. The least polluted routes always have less PM2.5 concentration because it could try other longer but cleaner roads.

By looking at the distances of the least polluted routes in Figure 5.13, the distance increases rapidly from 4AM to 5AM. In Figure 5.11, the least polluted routes take different roads at 4AM and 5AM. This could be explained by a variety of factors which attribute to change the pollution in this area, e.g particular event taking near the university and traffic.

In conclusion, this section shows the difference between the least polluted routes and the shortest distance during a day. The former is dynamic, while the later is not. For someone who wants to check the least polluted route for a certain time of day, the Android mobile application can provide the cleanest route for the user by using the latest pollution data. The heat map function also helps the user to check the distribution of the PM 2.5 concentration along the route so that the user can prepare (e.g by wearing a mask) to pass those areas with relatively higher PM2.5 concentration.

Chapter 6

Conclusions

Various algorithms were investigated for planning the least polluted routes in a large map in Edinburgh. The pollution data for the large map was generated by GAN so that algorithms could be tested in planning long-distance least polluted routes. Experiments including different distances of path planning were implemented to test the performance of algorithms by evaluating the PM2.5 concentration and the distance of the route, run time of the algorithms as well as the number of expanded nodes. For each path, the average of run time of each algorithm was calculated for running 100 times. For a least polluted route, the optimal solution is deterministic. Thus, the comparison of Dijkstra, A* and Bidirectional A* in planning the least polluted route would focus on run time and the number of expanded nodes. For three algorithms, the results in section 5.1 show that the performance of Dijkstra and A* are mostly the same based on runtime. With increasing distance of the path, calculating the heuristic function is going to be the most signification portion of the run time which slows down A*. The Bidirectional A* takes less time in planning long-distance path than A*. This is because Bidirectional A* always expands fewer nodes than A* during the search with the help of expanding two smaller trees. Although Dijkstra runs that fastest among three algorithms in planning different paths, it expanded more nodes than A*, more than twice as Bidirectional A*. So, currently Bidirectional A* is the best choice in planning the least polluted route since it expands fewer number of nodes while the run time remains short.

The dynamic weighting A* adjusts the value of the heuristic function during the runtime in order to speed up the search and reach the target quickly. For paths with different distance, the ϵ should be adjust accordingly. The strategy of setting ϵ was summarised in section 5.2. For short-distance paths, higher ϵ in range (5,10) can reduce

the runtime and the number of expanded nodes of the algorithm as well as guaranteeing the optimal solution. For long-distance paths, ϵ in range (5,6) is most admissible which could help generating close-to-optimal solution.

An Android mobile application was developed for checking the least polluted route at any time of day. A server called least polluted route was deployed on the Google Cloud App Engine which is responsible for running algorithms and generating least polluted route for the Android mobile application. The Android mobile application allows the user to plan the least polluted routes between any places. The heat map generated for each route can help the user checking the distribution of PM2.5 concentration along the route. The last section 5.5 in Results discussed that the least-polluted-route server is prepared to interface with another server called pollution-data server which is responsible for processing the live data and send it to the least-polluted-route server. In addition, the pollution data in this year during one day (2019-07-24) was used to test the algorithms. The least polluted routes changed at different times of day. By comparing to the shortest route generated on that day, it could be seen that the least polluted routes avoided some places with high concentrations of PM2.5. This observation indicates that the Android mobile application and the algorithms on server are reliable in planning the least polluted route.

In conclusion, the aim of future work should be to investigate the algorithms further especially the heuristic function of A* (e.g try different methods to estimate the remaining PM concentration of the routes during the search). The design and functionality of application still need to be improved (e.g show the information of the destination like post code). The least-polluted-route server should be developed further for dealing with heavy requests. A pollution-data server should be developed as well for updating the live data for the least-polluted-route sever.

Bibliography

[1] Beijing real-time air quality inex(aqi) & pollen report - air matters. https://www.unenvironment.org/news-and-stories/blogpost/young-and-old-air-pollution-affects-most-vulnerable. Accessed Aug. 2019.

[2] Openstreetmap export. https://www.openstreetmap.org/export. Online; accessed 30-July-2019.

[3] Openstreetmap functions. https://www.mathworks.com/matlabcentral/fileexchange/35819-openstreetmap-functions?s$_tid = prof_contriblnk. AccessedJun.$2019.

[4] Particulate matter (pm) basics. https://www.epa.gov/pm-pollution/particulate-matter-pm-basics. Accessed Mar. 2019.

[5] Revealed: Scotland's most polluted streets in 2018. https://foe.scot/press-release/scotlands-most-polluted-streets-2018/. Accessed July. 2019.

[6] Source and effects of pm2.5. https://laqm.defra.gov.uk/public-health/pm25.html. Accessed Mar. 2019.

[7] Young and old, air pollution affects the most vulnerable. https://www.unenvironment.org/news-and-stories/blogpost/young-and-old-air-pollution-affects-most-vulnerable. Accessed Aug. 2019.

[8] Air quality index (aqi). https://www.epa.gov/sites/production/files/2014-05/documents/zell-aqi.pdf, April 2012. Accessed 16 Mar. 2019.

[9] Air pollution, facts and information. https://www.nationalgeographic.com/environment/global-warming/pollution/, Feb 2019. Accessed Mar. 2019.

[10] DK Arvind, Janek Mann, Andrew Bates, and Konstantin Kotsev. The airspeck family of static and mobile wireless air quality monitors. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 207–214. IEEE, 2016.

[11] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.

[12] OpenStreetMap Wiki contributors. Map features, 2019. [Online; accessed 30-July-2019].

[13] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[15] Valerie B Haley, Thomas O Talbot, and Henry D Felton. Surveillance of the short-term impact of fine particle air pollution on cardiovascular disease hospitalizations in new york state. *Environmental Health*, 8(1):42, 2009.

[16] Eric A Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[17] Robert C Holte, Ariel Felner, Guni Sharon, Nathan R Sturtevant, and Jingwei Chen. Mm: A bidirectional search algorithm that is guaranteed to meet in the middle. *Artificial Intelligence*, 252:232–266, 2017.

[18] Stefan Ivanov. Sensing Spaces: Least Polluted Routes. Master's thesis, University of Edinburgh, 2017.

[19] Sitthinut Kumpalanuwat. The healthy city-tour route planner for cyclists in the area of central Edinburgh. Master's thesis, University of Edinburgh, 2019. unpublished thesis, in progress.

[20] Maxim Likhachev. A* and weighted a* search. University Lecture, 2010.

[21] Nils J Nilsson and Nils Johan Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann, 1998.

[22] Oracle. What are restful web services, 2013. [Online; accessed 5-Aug-2019].

[23] Amit Patel. Heuristic function, 2019. [Online; accessed 19-July-2019].

[24] Amit Patel. Variants of a*, 2019. [Online; accessed 19-July-2019].

[25] Zoe Petard. High Resolution Spatial Predictions of Air Pollution Levels in Edinburgh. Master's thesis, University of Edinburgh, 2018.

[26] EGAN Ryan. An Online Spatio-Temporal Model to Predict Air Pollution With Mobile and Stationary Data. Master's thesis, University of Edinburgh, 2019. unpublished thesis, in progress.

[27] Jordan Tyler Thayer and Wheeler Ruml. Using distance estimates in heuristic search. In *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[28] Max Welling. Informed search algorithms. University Lecture, 2009.

[29] Wikipedia. Iterative deepening a*, 2018. [Online; accessed 19-July-2019].

[30] Wikipedia. Air quality index, 2019. [Online; accessed 30-July-2019].

[31] Wikipedia. Bidirectional search, 2019. [Online; accessed 5-Aug-2019].

[32] Wikipedia. Breadth-first-search, 2019. [Online; accessed 5-Aug-2019].

[33] Wikipedia. Great-circle distance, 2019. [Online; accessed 19-July-2019].

[34] Wikipedia. Haversine formula, 2019. [Online; accessed 19-July-2019].

[35] Wikipedia. Openstreetmap, 2019. [Online; accessed 30-July-2019].

[36] Wikipedia. Pathfinding, 2019. [Online; accessed 5-Aug-2019].

[37] Wikipedia. Representational state transfer, 2019. [Online; accessed 5-Aug-2019].

[38] Melissa Yan. Dijkstra's algorithm. *Massachusetts Institute of Technology. Regexstr*, 2014.