NANYANG TECHNOLOGICAL UNIVERSITY

SCE16-0446
A Machine Learning-Based Approach to Time-Dependent Shortest Path Queries

Submitted in Partial Fulfillment of the Requirements
for the Bachelor of Computer Science
of the Nanyang Technological University

by

Wei Yumou

School of Computer Science and Engineering

SCE16-0446

A Machine Learning-Based Approach to Time-Dependent Shortest Path Queries

by

Wei Yumou

Submitted to the School of Computer Science and Engineering
on March 23, 2017, in partial fulfillment of the
requirements for the degree of
Bachelor of Computer Science

Abstract

Road traffic is known to be time-dependent. The travel time of a road varies at different times of the day. Many algorithms have been proposed for finding a shortest path in a time-dependent road network. In this project, I explored an alternative approach that leveraged on GPS trajectories collected from thousands of taxis. Each GPS trajectory was mapped to a set of real road segments. An abstract landmark graph was built to represent the city's road network and a machine learning-based approach was proposed to estimate the travel time of each edge. The estimates made by this approach were compared against real-time estimates made by existing online mapping services to evaluate its accuracy. A modified Dijkstra's algorithm was presented to calculate a shortest path in a time-dependent landmark graph, based on the travel time estimates.

FYP Supervisor: Xiao Xiaokui
Title: Associate Professor, Assistant Chair (Strategic Research)

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Assoc Prof. Xiao Xiaokui, who gave me this great opportunity to work on GPS trajectory mining and offered valuable insights to help me tackle challenges.

I would also like to thank my parents and friends who have been constantly providing me with lots of emotional support.

Lastly, I would appreciate the Computational Sensing Lab at Tsinghua University, Beijing, China, for their generosity in sharing the data set used in this project and also thank Baidu for providing wonderful Baidu Maps APIs which constitute an essential part of this project.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Finding a route with the least travel time on a large road network in a metropolis is not only of algorithmic interests, but also of economic and environmental values. Less travel time means less fuel consumptions and less carbon emissions. However, finding the *shortest* route can be a challenging task, especially when the road traffic is known to be *time-dependent* or *dynamic*, namely, when the travel cost changes with respect to time. It may take 10 minutes on average to traverse a particular road at 10 a.m., but it is possible that the expected travel time increases to 20 minutes at 5 p.m. Moreover, the travel costs of two different roads may have different time-varying patterns. For instance, one road may have a peak travel time at 12 p.m. but the other may have two peaks at 8 a.m. and 6 p.m., respectively. Definition 1 gives a formal description of a dynamic road network, based on which the generalised time-dependent shortest path problem is defined in Definition 2.

Definition 1 (*Dynamic road network*)**.** A dynamic road network is a weighted, directed graph $G = (V, E)$ where E represents a set of road segments and V denotes the set of intersections amongst the road segments. It has a weight function $w : E, t \to \mathbb{R}$, where t represents a moment in time.

Definition 2 (*Generalised time-dependent shortest path problem*)**.** In a dynamic road network $G = (V, E)$, given a source node u, a destination node v and a departure time t from u, find a path p that satisfies:

$$w(p) = \delta(u, v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\rightsquigarrow} v\right\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases} \quad (1.1)$$

where $w(p)$ is the weight of the path p and defined as sum of the weights of its constituent edges, and $\delta(u, v)$ is known as the **shortest-path weight** from u to v.

A typical Bellman-Ford [1] or Dijkstra's algorithm [2] for finding shortest paths assume the cost of traversing each edge in a graph is constant with respect to time and therefore, do not work on time-dependent road networks without appropriate modifications. Fortunately, most online mapping services such as Google Maps or Baidu Maps are able to recommend shortest routes by interpolating real-time traffic information. This project seeks to investigate an alternative approach for finding shortest routes on a dynamic road network based on mining a GPS[1] trajectory database aggregated from thousands of taxis in Beijing, China.

Chapter 2 describes the steps taken in preliminary data processing, where outliers in the data set are removed and each GPS trajectory is mapped to a set of streets. Chapter 3 introduces the approach for constructing a landmark graph that represents a city's road network in an abstract way. Chapter 4 discusses the method to estimate the travel cost of each edge in a landmark graph. Chapter 5 presents an algorithm for finding a shortest path given the dynamic travel costs.

[1]Global Positioning System

1.1 Motivations for Mining Taxi GPS Trajectories

Taxi drivers or any experienced car drivers, more often than not, possess some *implicit* knowledge or intuitions about which route from a source u to a destination v is the best in terms of travel time at a particular moment. Such knowledge or intuitions stem from everyday experiences. For example, a taxi driver may observe that there are always traffic jams from 6 p.m. to 7 p.m. on a particular street and hence avoid travelling on that street during that period of time whenever possible. But observations of this kind, albeit valuable, are too subtle to be captured by any general algorithms and oftentimes, even drivers themselves may not be fully aware of that.

However, mining their GPS trajectories can be an excellent way of revealing such knowledge. In a metropolis such as Beijing or New York, taxi drivers are required by regulations to install GPS devices on their cars and to send time-stamped GPS information to a central reporting agency periodically, for management and security reasons. The GPS information typically includes latitudes, longitudes, instantaneous speeds and heading directions. Therefore, such GPS data is readily available and little effort is needed to collect it. By means of mapping to a real road network all GPS data points pertaining to a particular taxi during a specific period of time, a GPS trajectory can be obtained to represent the driver's intelligence. Definition 3 formally defines taxi trajectories.

Definition 3 (*Taxi Trajectory*)**.** A taxi trajectory $T = (p_1, p_2, \ldots, p_n)$ is a set of chronologically ordered GPS data points pertaining to one taxi. The time span between p_1 and p_n is defined as the *duration* of the trajectory.

1.2 Practical Limitations

Some practical limitations are worth mentioning.

Arbitrary sources and destinations

In a typical map-query use case, a user is able to select an arbitrary source to start with and an arbitrary destination to go to. But this may not be possible for the approach proposed in this project, since the taxi GPS trajectories do not necessarily cover every part of a city's road network. It is likely that there are no trajectories passing through one particular source and one particular destination.

Low sampling rate

Taxis report their locations to the central reporting agency at a relatively low frequency to conserve energy. For the data set used in this project, the expected sampling interval is one minute. But oftentimes, the GPS device may not be working properly or may be occasionally shut down due to various reasons, which causes the actual sampling interval to fluctuate.

Even if the sampling interval *is* strictly kept at one minute, for a taxi moving at a typical speed of $60km/h$, it means the distance between two consecutive sample points is $1km$. Such a large distance increases the uncertainty about the *exact* trajectory that the taxi has moved along. Figure 1-1 [15] demonstrates a problem caused by low sampling rate and long inter-sample-point distance.

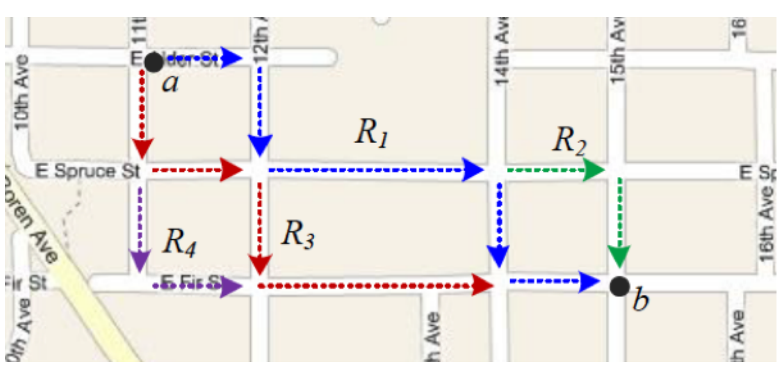


Figure 1-1: An example of low-sampling-rate problem

The taxi is known to have traversed from point a to point b. But there are four possible trajectories from a to b. In this case, the exact route cannot be determined without additional information.

Limited GPS accuracy

After decades of development, the GPS has achieved great accuracy, but it is not completely error-free. A report [11] in 2015 showed that GPS-enabled smartphones typically have an accuracy of 5 metres *under open sky*. But in a metropolis like Beijing, the actual accuracy may be lower due to the reflection of signals amongst high buildings. Moreover, the data set used in this project was collected in 2009 when GPS devices had lower accuracy than they have today.

The limited accuracy in GPS devices makes it difficult to map a GPS data point to a real street when there are two streets located very near to each other. In Beijing, there is usually a side road running in parallel with a main road. Due to the limited-accuracy problem, a taxi might be *actually* on the side road when some of its GPS data points are mapped to the main road, or vice versa.

1.3 Related Work

The incentive for carrying out this project comes from a similar project [15] by Microsoft Research Asia. A number of concepts and procedures should be credited to them. But this project has also adopted some innovative strategies that cater to its own unique situations.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Preliminary Data Processing

2.1 Data Collection and Cleaning

The taxi GPS data used in this project is collected from the Computational Sensing Lab [16] at Tsinghua University, Beijing, China. The data set contains approximately 83 million time-stamped taxi GPS records collected from 8,602 taxis in Beijing, from 1 May 2009 to 30 May 2009. Originally, the data set consists of seven fields as shown in Table 2.1. Longitude and latitude in the data set are defined in the WGS-84[1] standard coordinate system, which is the reference coordinate system used by the GPS.

The original data set came in a binary file format. After it was decoded and imported into a MySQL database, the first step in data cleaning was **to delete all records with zero value in the SPEED field**, since when a taxi is stationary it yields no valuable information about the *trajectory* it is moving along. While being stationary could be due to a traffic jam, this kind of information is well captured by the time span between the last and the next *non-stationary* data point.

[1] World Geodetic System

Field	Explanation
CUID	ID for each taxi
UNIX_EPOCH	Unix timestamp in seconds since 1 January 1970
GPS_LONG	Longitude encoded in WGS-84 multiplied by 10^5
GPS_LAT	Latitude encoded in WGS-84 multiplied by 10^5
HEAD	Heading direction in degrees with 0 denoting North
SPEED	Instantaneous speed in metres/second (m/s)
OCCUPIED	Binary indicator of whether the taxi is hired (1) or not (0)

Table 2.1: A summary of the seven original fields

In addition, all records must have a *unique* pair of **CUID** and **UNIX_EPOCH** fields, since it is not possible for a taxi to appear in two different locations at the same moment in time. This kind of error is likely due to some errors in aggregating the original data set.

2.2 Reverse Geocoding

After the data set was cleaned, the next step was to map each GPS data point to a real street based on its longitude and latitude, which is also known as *reverse geocoding*. A number of algorithms [7] have been proposed for this purpose, but most of them require an additional GIS[2] database of the road network in Beijing. This project adopted an alternative strategy which leveraged on the existing public API[3]s for reverse geocoding.

Currently, a number of online mapping platforms provide reverse geocoding services as part of their developer APIs. Amongst others, Google Maps and Baidu Maps

[2] Geographic Information System
[3] Application Programming Interface

offer relatively stable and fast reverse geocoding services. However, due to the "China GPS shift problem" [13] where longitudes and latitudes encoded in WGS-84 format are required by regulations to be shifted by some variable amounts when displayed on a street map, Google Maps is not able to display a GPS data point correctly because it only supports WGS-84 formats. Figure 2-1 illustrates the effect of such shift, where the correct location is displayed on the **right**.

Figure 2-1: An example of China GPS shift problem

Baidu Maps, on the other hand, has been using their own coordinate system, BD-09, which is an improved version of the Chinese official coordinate system, GCJ-02. Baidu Maps does provide a set of APIs to convert WGS-84 coordinates into BD-09 ones. To reverse-geocode GPS data points, longitudes and latitudes must first be converted into BD-09 format. Then another set of APIs from Baidu Maps can be used for reverse geocoding. To store the converted longitudes and latitudes as well as the street names obtained from reverse geocoding, four new fields were added to the original data set as shown in Table 2.2.

In order to use Baidu Maps APIs for coordinate conversion, the following system architecture was set up as shown in Figure 2-2. The Apache HTTP server hides the MySQL database and sends HTTP POST requests containing the original coordinates

Field	**Explanation**
DataUnitID	Nominal primary key for each record
UTC	UNIX_EPOCH in human readable format
BD09_LONG	Longitude encoded in BD-09 format
BD09_LAT	Latitude encoded in BD-09 format
STREET	Street name

Table 2.2: A summary of additional fields

to Baidu Maps Web API to get the converted coordinates. Then it updates the database through PHP *mysqli* utility.

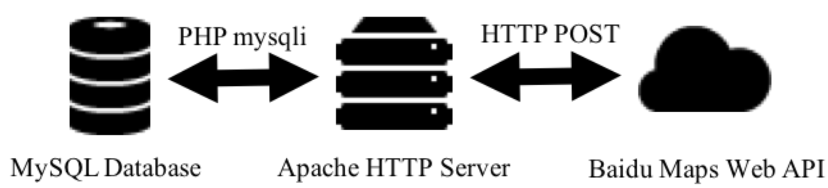


Figure 2-2: System architecture for coordinate conversion

After longitudes and latitudes were converted from WGS-84 format to BD-09 format, Baidu Maps Web API was used to reverse geocode all GPS data points. However, the system architecture was slightly changed, to accommodate the change in technology used. For reverse geocoding, AJAX[4] was used to communicate with the Baidu Maps Web API for speed and unlimited number of requests per day. Therefore, one additional layer was added to the existing system architecture as shown in Figure 2-3.

Executed in a web browser environment, AJAX sent HTTP POST requests to the Apache HTTP server to fetch the converted coordinates in BD-09 format which were

[4] Asynchronous JavaScript and XML

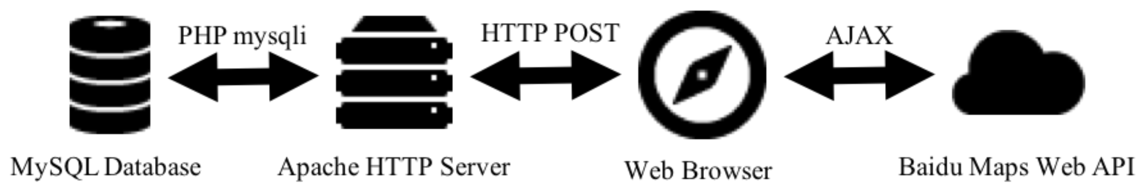


Figure 2-3: System architecture for reverse geocoding

subsequently sent to the Baidu Maps Web API server *asynchronously* via HTTP GET requests. Once the server responded with the name of the street, AJAX updated the database by sending another HTTP POST request to the Apache HTTP server. The asynchronous nature of this architecture, however, caused a few problems which are addressed in Section 2.3.

2.3 Outlier Detection

2.3.1 Motivations for Outlier Detection

The Baidu Maps Web API for reverse geocoding is stable and fast, but does not produce no errors. Sometimes, a GPS data point may be mapped to a main road but actually it is on the side road, which is one of the limitations mentioned in Section 1.2, or it is actually mapped to a street that Baidu Maps does not recognise. In neither case will Baidu Maps produce a correct reverse geocoding. Moreover, the reverse geocoding process is asynchronous, which means that it is being performed in the background in parallel with the main application thread. Therefore, it is inevitable that some street names may get lost when the records are being updated or a record is updated with a wrong street name. Figure 2-4 shows a drastic example.

In this example, Baidu Maps believes that all data points plotted belong to a particular street. But when plotted on a 2-D plane, these data points almost represent

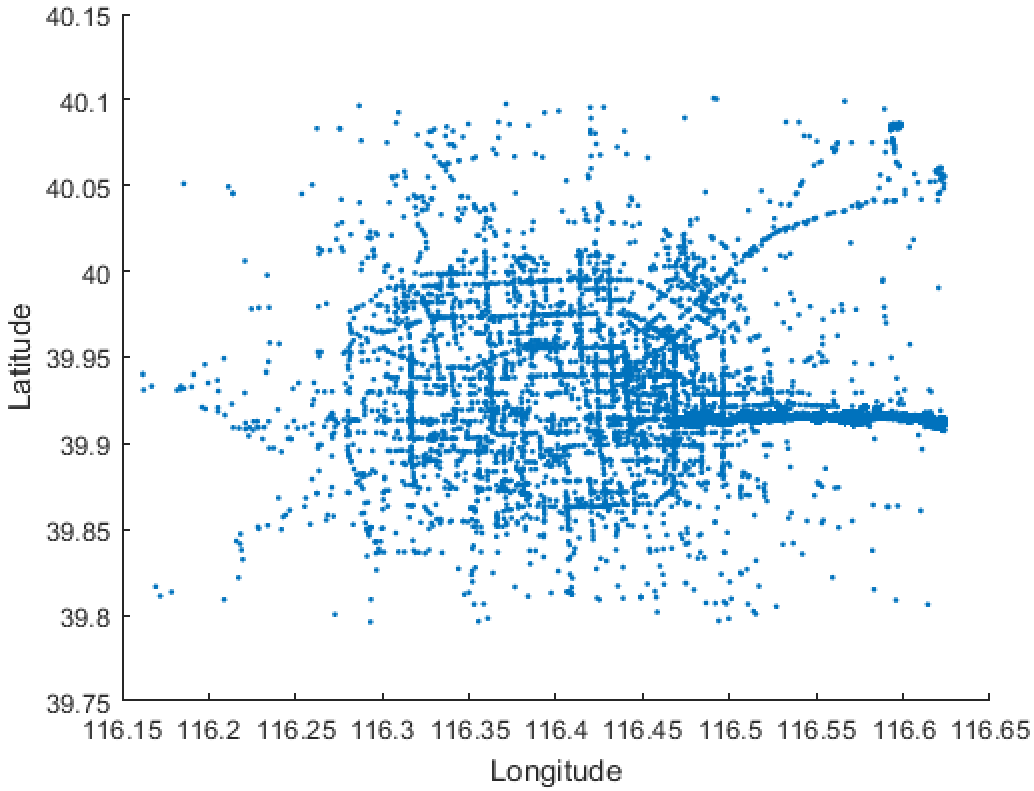


Figure 2-4: An example of outliers

the *entire* road network in Beijing. The actual, correct street is represented in the figure as the *thickest* blue line on the right half of the figure with a longitude ranging from 116.45°E to 116.65°E. Erroneous records like those not on the thickest line are known as *outliers* and must be properly identified and removed. This project proposes a novel outlier detection approach based on unsupervised learning whose principle behind is based on Theorem 1.

Definition 4 (*Reasonable reverse geocoder*)**.** A reasonable reverse geocoder always gives its best matching from a GPS data point to a street whenever possible and has an accuracy more than 50%.

Theorem 1 (*Majority Clustering Theorem*)**.** If a *reasonable reverse geocoder* is used to reverse-geocode a set of GPS data points which are mapped to a particular street *in reality*, then, when plotted on a 2-D plane, majority (more than 50%) of the points must be clustered together to form a rough shape that is similar to the shape of the street that they are supposed to be mapped to.

Proof. Proof by contradiction. Assume, for the purpose of contradiction, majority (more than 50%) of the data points that are *indeed* located on the same street are scattered arbitrarily on a 2-D plane after being reverse-geocoded by a reasonable reverse geocoder. In particular, when plotted on a 2-D plane, majority of them *do not* form any similar shape to that of the street they are supposed to be mapped to. Then, the majority must have been erroneously mapped to some other streets because no single street covers the whole city's road network. Thus, the reasonable reverse geocoder has only achieved an accuracy that is less than 50%, which contradicts Definition 4 of a reasonable reverse geocoder. □

2.3.2 Outlier Identification

Apparently, Baidu Maps provides a reasonable reverse geocoder because it is of industrial-grade quality and has an accuracy larger than 50%. Therefore, if a set of points belong to a particular street, after reverse-geocoded by Baidu Maps, majority of them should be clustered to assume a rough shape of that street according to Theorem 1. Based on that, an unsupervised learning technique — clustering can be used to separate the correctly mapped data points from the outliers.

Many clustering techniques are available [10]. Since each record can be represented graphically as a point on a 2-D plane with longitude as the x axis and latitude as the y axis, a **self-organising feature map** (SOFM) [6] seems to be an appropriate technique to use.

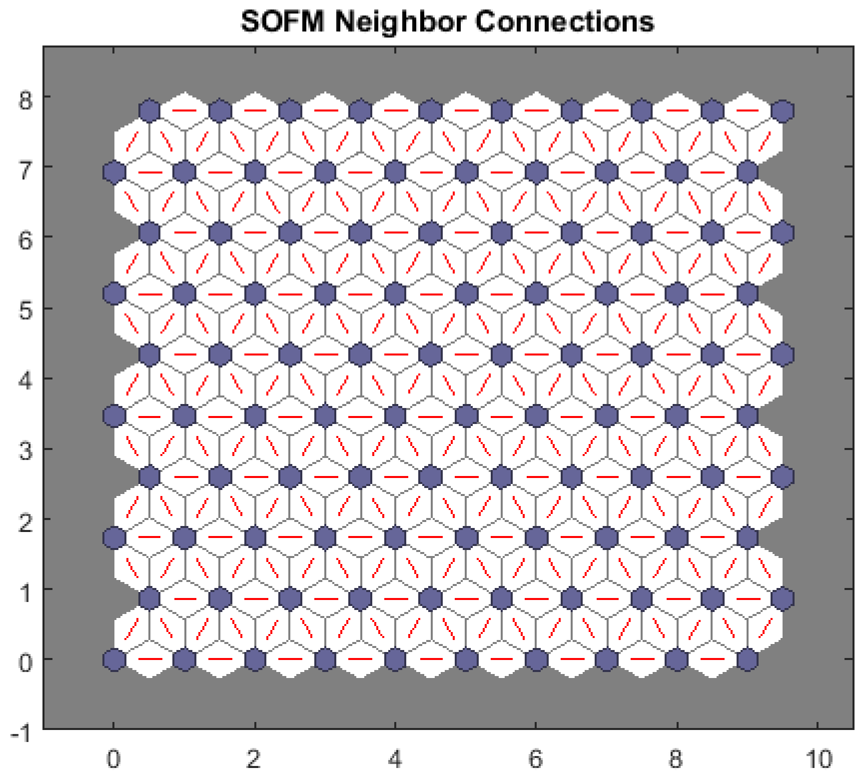


Figure 2-5: An example of SOFM

A self-organising feature map is a form of artificial neural network. It consists of a pre-defined number of interconnected neurons distributed over a 2-D plane as shown in Figure 2-5. Prior to training, the neurons are randomly scattered among the data points and gradually move to the *centroids* of the data clusters they represent as they learn the *features* of the training data. Upon termination of the training, all data points near to a particular neuron, in terms of Euclidean distance[5], are assigned to the cluster that neuron represents. Figure 2-6 shows the results after the clustering is completed.

It is clear from the figure that while some neurons represent the clusters of outliers,

[5]Other distance measures are also possible.

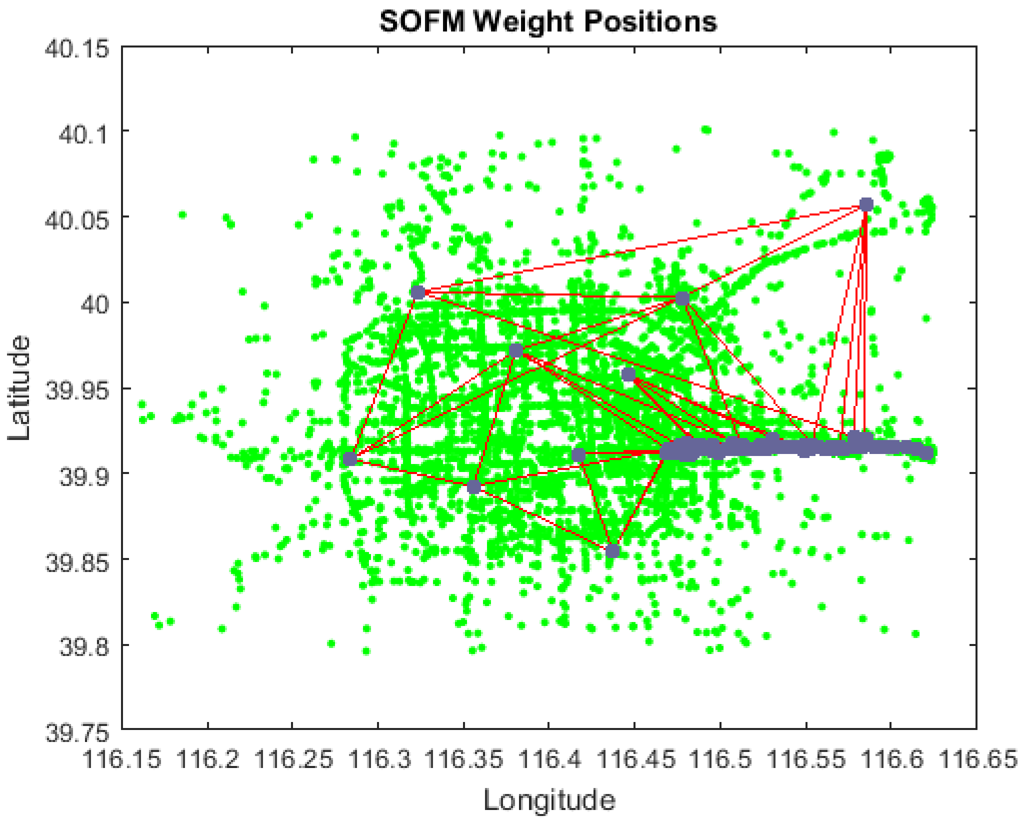


Figure 2-6: A plot of neuron positions after training

majority of the neurons are clustered to *cover* the correct street they should represent. A 10×10 SOFM was used in this project, so there were at most 100 neurons or equivalently, 100 clusters. Each cluster had a different size. To ensure a thorough removal of the outliers, **only the top 50% largest clusters were considered as clusters of correct data points which are called “legal clusters”. All other clusters were deemed as clusters of outliers**.

2.3.3 Outlier Removal

Once the legal clusters were identified, a distance threshold was set to remove outliers so that **whenever the minimum distance between a data point and all cen-**

troids of the legal clusters was above the threshold, that data point would be considered as an outlier and removed. The python-like pseudocode in Listing 2.1 describes this idea with more clarity. Source code is listed in Appendix A.2.

Listing 2.1: Pseudocode for outlier removal

```
Input: a set of GPS records, a set of centroids and a distance threshold
    d_max
Output: a set of GPS records with all outliers removed

for recd in records:
    min_dist = ∞
    for ctrd in centroids:
        min_dist = min(min_dist, get_distance(recd, ctrd))
    if min_dist > d_max:
        records.remove(recd)
```

However, the *distance* between a data point and a centroid is not as straightforward as Euclidean distance. A centroid, to some extent, can be imagined as a *real* point on the Earth's surface. To calculate the distance between a data point and a centroid is to calculate the **spherical distance** between them, which is given by the Haversine Formula in Theorem 2.

Theorem 2 (*Haversine Formula*)**.** Given two points $P(\lambda_1, \varphi_1)$ and $Q(\lambda_2, \varphi_2)$ on the surface of a sphere, where λ and φ represent longitude and latitude in radians, their spherical distance (the distance along a great circle of the sphere) is given by [9]

$$d = 2R \arcsin \sqrt{\sin^2 \frac{\varphi_2 - \varphi_1}{2} + \cos\varphi_1 \cos\varphi_2 \sin^2 \frac{\lambda_2 - \lambda_1}{2}} \tag{2.1}$$

where R is the radius of the sphere.

Since the Earth is not a perfect sphere but a spheroid, R varies with latitude.

Theorem 3 suggests how to calculate the Earth radius at any latitude.

Theorem 3 (*Radius at any Latitude*)**.** Given a latitude φ in radians, a polar radius R_p and an equatorial radius R_e, the spheroid's radius at that latitude is given by [12]

$$R_\varphi = \sqrt{\frac{(R_e^2 \cos\varphi)^2 + (R_p^2 \sin\varphi)^2}{(R_e \cos\varphi)^2 + (R_p \sin\varphi)^2}} \tag{2.2}$$

It is known that $R_e = 6,378,137$ metres and $R_p = 6,356,752$ metres on the Earth [14] and that Beijing's latitude is about 39°N. Therefore, the distance between a data point and a centroid can be calculated. For this project, two distance thresholds were selected: 30 metres and 50 metres. The thresholds were set in a way that it ensured there was sufficient data for subsequent machine learning tasks while the estimates of travel time were as least affected as possible by outliers. Should the thresholds be set to a too small value, the remaining data could not have been sufficient; on the other hand, however, if the thresholds were set to a too big value, the accuracy of the final results would have been subject to outliers.

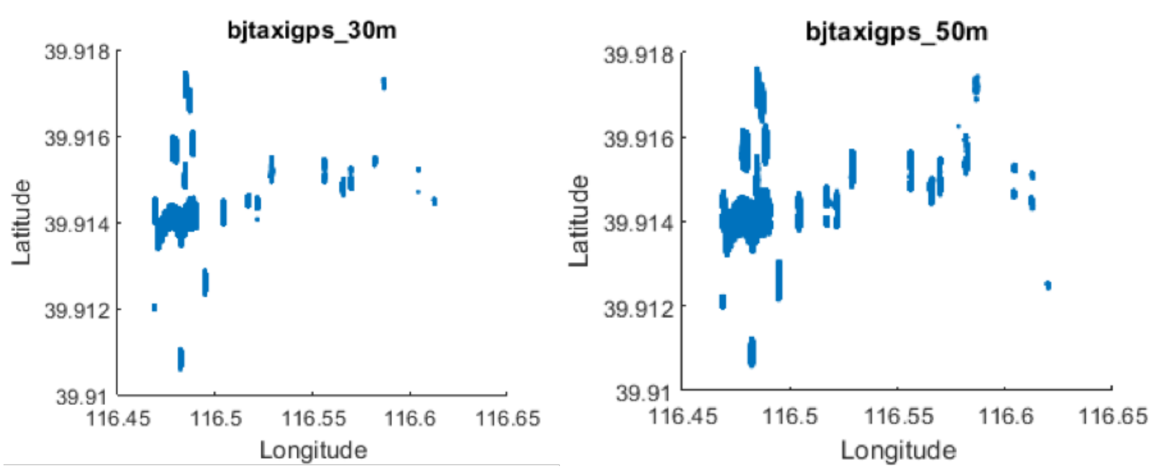


Figure 2-7: A plot of data points after outlier removal

After the outliers were removed, two data sets remained. They are hereinafter referred to as **bjtaxigps_30m**, where outliers were filtered by a distance threshold of 30 metres and **bjtaxigps_50m**, where outliers were filtered by a distance threshold of 50 metres, respectively. bjtaxigps_30m contains approximately 51 million records while bjtaxigps_50m has 59 million. All algorithms and procedures described hereinafter are applicable to both data sets. Figure 2-7 gives a plot of the GPS data points after outliers were removed from both data sets. Clearly, the longitude and latitude now have a much smaller *range* and the data points roughly form a shape similar to that of the street they are supposed to be mapped to.

Chapter 3

Landmark Graph Construction

Definition 5 (*Landmark*)**.** A landmark is a road segment that is frequently traversed by taxi drivers according to the taxi GPS trajectory database. [15]

The concept of a landmark is proposed primarily for two reasons. First, as mentioned in Section 1.2, the taxi GPS trajectories do not necessarily cover every road segment in a city's road network and the low-sampling-rate problem makes it difficult to determine the exact route on which a taxi traversed. Therefore, it is not possible to estimate the time-dependent travel cost for each road segment. However, ignoring the length of a road segment and considering it as an abstract "landmark" make estimating the travel cost between two *landmarks* feasible.

Moreover, the notion of landmarks closely follows the way how drivers remember driving routes in daily life [15]. For instance, a driving route could be described as "go straight on the 4th Avenue, turn right at the 7th Street and exit at the Smith Road". Drivers tend to use familiar road segments as landmarks to guide their directions.

This chapter describes the procedures for constructing an abstract "landmark graph" in order to estimate the travel cost between two landmarks. But before that, the taxi GPS trajectories must be separated into a set of *trips*.

3.1 Trip Identification

Definition 6 (*Trip*)**.** A trip is a taxi trajectory $T = (p_1, p_2, \ldots, p_n)$ that satisfies either condition:

1. A passenger is on the taxi, namely,

$$\forall i \leq |T|, \quad p_i.OCCUPIED = 1; \tag{3.1}$$

2. No passenger is on board, but the time span between *any* two consecutive records is no longer than 3 minutes, namely,

$$\forall i \leq |T| - 1, \quad p_{i+1}.UTC - p_i.UTC \leq 3mins. \tag{3.2}$$

Table 3.1 gives a tiny example of trip identification.

CUID	UTC	GPS_LONG	GPS_LAT	OCCUPIED	TRIP_ID
1	1/5/2009 0:02:00	116.39616	39.81294	0	4552265
1	1/5/2009 0:04:00	116.39575	39.82296	0	4552265
1	1/5/2009 0:07:00	116.39567	39.82774	0	4552265
1	1/5/2009 17:08:00	116.30142	39.98105	1	1
1	1/5/2009 17:10:00	116.29514	39.98419	1	1
1	1/5/2009 17:11:00	116.28959	39.98289	1	1
1	1/5/2009 17:12:00	116.28087	39.97552	1	1
1	1/5/2009 17:16:00	116.26813	39.93537	1	1
1	1/5/2009 18:11:00	116.36537	39.95019	0	4552271
1	1/5/2009 18:12:00	116.36546	39.94886	0	4552271
1	1/5/2009 18:13:00	116.35927	39.94528	0	4552271

Table 3.1: An example of trip identification

Clearly, all records in Table 3.1 belong to one taxi (taxi with CUID = 1) and are sorted chronologically. The first three records, although no passengers are aboard, are considered to be in the same *trip* because the time span between any two consecutive records is no longer than 3 minute. The next five records constitute another trip, even though the last two records have a time span of 4 minutes, since a passenger is on the taxi (OCCIPIED = 1). Following the same reasoning as the first three's, the last three records are treated as one trip. Listing 3.1 gives the pseudocode for trip identification. Source code is shown in Appendix A.3.

Listing 3.1: Pseudocode for trip identification

```
Input: a set of GPS records and a time span threshold t_max
Output: a set of GPS records with each record assigned a trip ID

curr_tripid = 1
last_occup = curr_occup = records[0].OCCUPIED
last_unixepoch = curr_unixepoch = records[0].UNIX_EPOCH
for recd in records:
    curr_occup = recd.OCCUPIED
    curr_unixepoch = recd.UNIX_EPOCH
    if curr_occup != last_occup:
        ++curr_tripid
    elif (curr_occup == 0) and (curr_unixepoch - last_unixepoch > t_max):
        ++curr_tripid
    recd.TRIP_ID = curr_tripid
    last_occup = curr_occup
    last_unixepoch = curr_unixepoch
```

It is noteworthy that although the middle five records with OCCUPIED = 1 come chronologically after the first three records, they are nevertheless assigned a smaller TRIP_ID. This is due to an adjustment made to the actual implementation of the algorithm. The *actual* algorithm operates in two stages. It first identifies trips for all records with OCCUPIED = 1; only in the second stage does it identify trips for records with OCCUPIED = 0. The rationale for this arrangement is to give priority to the records with passengers aboard, because **these records are guaranteed to belong to one trip**. In fact, the second condition for identifying trips in Definition 6 is more of a heuristic rather than a theorem. It is meant to provide sufficient data for subsequent machine learning tasks with reasonable accuracy.

3.2 Landmark Frequency

CUID	UTC	GPS_LONG	GPS_LAT	Street	TRIP_ID
1	1/5/2009 0:02:00	116.39616	39.81294	A	4552265
1	1/5/2009 0:04:00	116.39575	39.82296	A	4552265
1	1/5/2009 0:07:00	116.39567	39.82774	B	4552265
1	1/5/2009 17:08:00	116.30142	39.98105	C	1
1	1/5/2009 17:10:00	116.29514	39.98419	C	1
1	1/5/2009 17:11:00	116.28959	39.98289	C	1
1	1/5/2009 17:12:00	116.28087	39.97552	A	1
1	1/5/2009 17:16:00	116.26813	39.93537	A	1
1	1/5/2009 18:11:00	116.36537	39.95019	B	4552271
1	1/5/2009 18:12:00	116.36546	39.94886	C	4552271
1	1/5/2009 18:13:00	116.35927	39.94528	C	4552271

Table 3.2: An illustration of frequency counting

Definition 6 states that whether recognising a particular road segment as a landmark or not is based on some notion of frequency. Only if a road segment is visited by drivers frequently enough is it recognised as a landmark. In this project, the notion of frequency is given by Definition 7.

Definition 7 (*Frequency of a Road Segment*)**.** The frequency of a road segment is the sum of *unique* occurrences of that road segment in each trip.

Table 3.2 illustrates how the frequency of a road segment is calculated. In Trip 4552265, Street A occurs twice but **its frequency increases only by one**. Similarly, Street C occur three times and Street A occur twice in Trip 1, but **their occurrences only add one to their frequencies**. Therefore, even though Street B *appears* less times than Street A or Street C, **all streets have the same frequency of 2**, based on this set of records.

The algorithm for counting frequency of a road segment is rather straightforward as in Listing 3.2. Source code is listed in Appendix A.4.

Listing 3.2: Pseudocode for counting road frequency

```
Input: a set of trips
Output: a set of frequencies for each street

frequencies = dict{}
for trip in trips:
    streets = unique(trip.streets) // get unique streets
    for street in streets:
        if street in frequencies:
            ++frequencies[street]
        else:
            frequencies[street] = 1
```

After the frequency of each road segment is determined, a parameter k is set to select the top k frequent road segments as *landmarks*. For this project, k was chosen to be 500 to ensure the road segments are significant enough to be landmarks. When plotted on a map, these 500 landmarks cover most of the main streets in Beijing as shown in Figure 3-1.

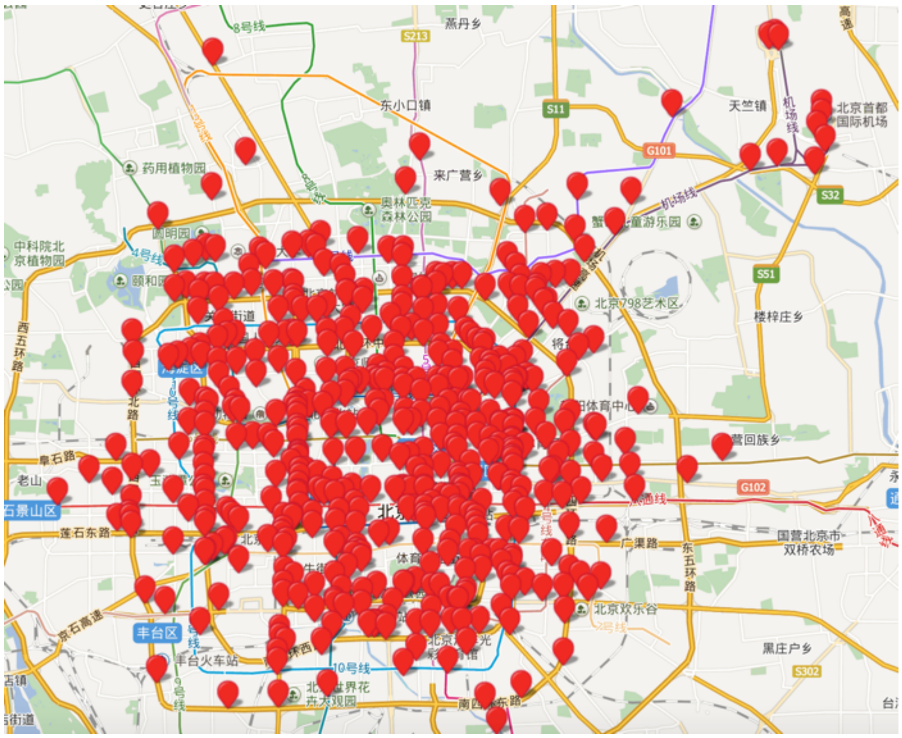


Figure 3-1: A plot of landmarks when $k = 500$

3.3 Landmark Graph

Definition 8 (*Landmark Graph*)**.** A landmark graph is a weighted, directed graph $G = (V, E)$ where V is the set of landmarks defined by parameter k and E is the set of taxi trajectories $T = (p_1, p_2, \ldots, p_n)$ that satisfy the following conditions:

1. p_1 and p_n must be landmarks;
2. $p_2, p_3, \ldots, p_{n-1}$ must **not** be landmarks and;
3. The duration of the trajectory must **not** exceed a threshold t_{max}.

The threshold t_{max} is used to eliminate trajectories with unreasonably long durations [15]. Sometimes, a taxi may consecutively traverse several landmarks but only the first and the last landmarks are recorded, due to the low sampling rate, which causes the time span between the two recorded landmarks to be unreasonably long. For this project, t_{max} was set to 20 minutes. The algorithm for constructing landmark graph is given in Listing 3.3. Source code is given in Appendix A.5. For each edge (u, v), the arrival time at u and the arrival time at v are also recorded for the subsequent task of estimating travel time.

Listing 3.3: Pseudocode for constructing landmark graph

```
Input: a set of trips
Output: a landmark graph G = (V, E)
for trip in trips:
   //get unique, chronologically ordered streets
   streets = unique_ordered(trip.streets)
   while j < len(streets):
        //loop until a landmark is found
        while j < len(streets) and not is_landmark(streets[j]):
            j = j + 1

        //find another landmark
        intermediaries = []
        k = j + 1
        while k < len(streets) and not is_landmark(streets[k]):
            intermediaries.append(streets[k])
            k = k + 1
        //insert edge
        E.insert(streets[j], intermediaries, streets[k])
        //next search starts from the second landmark
        j = k
```

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Time-Dependent Edge Cost Estimation

Chapter 3 describes the general procedures for constructing a landmark graph. To find a time-dependent shortest route on a landmark graph, the time-dependent edge cost must be calculated. In this project, the time-dependent edge cost specifically refers to travel time, but in theory, it can refer to any quantity that can be described as a time-dependent edge cost function of the form $w : E, t \to \mathbb{R}$. In practice, it sometimes also refers to fuel consumptions or taxi fares. This chapter introduces a machine learning-based approach to estimate the travel time of each *significant edge* in a landmark graph at a particular moment in time.

Definition 9 (*Support*)**.** A support for an edge e in a landmark graph $G = (V, E)$ is the number of times that e appears.

Definition 10 (*Significant Edge*)**.** A significant edge in a landmark graph $G = (V, E)$ is an edge $e \in E$ that has a support at least m, where m is a parameter specified in advance.

The purpose of defining significant edges is to eliminate those edges that are seldom traversed by taxi drivers, as estimating the travel time of those edges will not be very accurate. The parameter m also represents a level of *confidence*, that is, to what extent it is true that this edge *really* exists in the real world.

Everyday experiences show that the travel time of a particular road usually has different time-varying patterns in weekdays as compared to that in weekends or public holidays. For instance, it is likely that, on weekdays, the travel time of a particular road has one *peak* at 8 a.m. when people travel to work and the other peak at 6 p.m. when people return home after work. But when it is weekends or public holidays, the travel time of that road may have a peak at 10 a.m. when people go for holiday activities with families and the other peak at only about 8 p.m. when the whole day's celebrations are over.

Based on this intuition, two separate landmark graphs were built in this project, with one for weekdays and the other for weekends or public holidays. Moreover, as mentioned in Section 2.3.3, two data sets, bjtaxigps_30m and bjtaxigps_50m, remained after outlier removal based on different thresholds set for removing outliers. Therefore, in total, *four* landmark graphs were built in this project and they are summarised in Table 4.1, although their names are self-explanatory.

Landmark Graph	**Data Source**
wrkd_ldmkgraph_30m	weekday trajectories in bjtaxigps_30m
holi_ldmkgraph_30m	holiday trajectories in bjtaxigps_30m
wrkd_ldmkgraph_50m	weekday trajectories in bjtaxigps_50m
holi_ldmkgraph_50m	holiday trajectories in bjtaxigps_50m

Table 4.1: A summary of landmark graphs

4.1 Travel Time Distribution

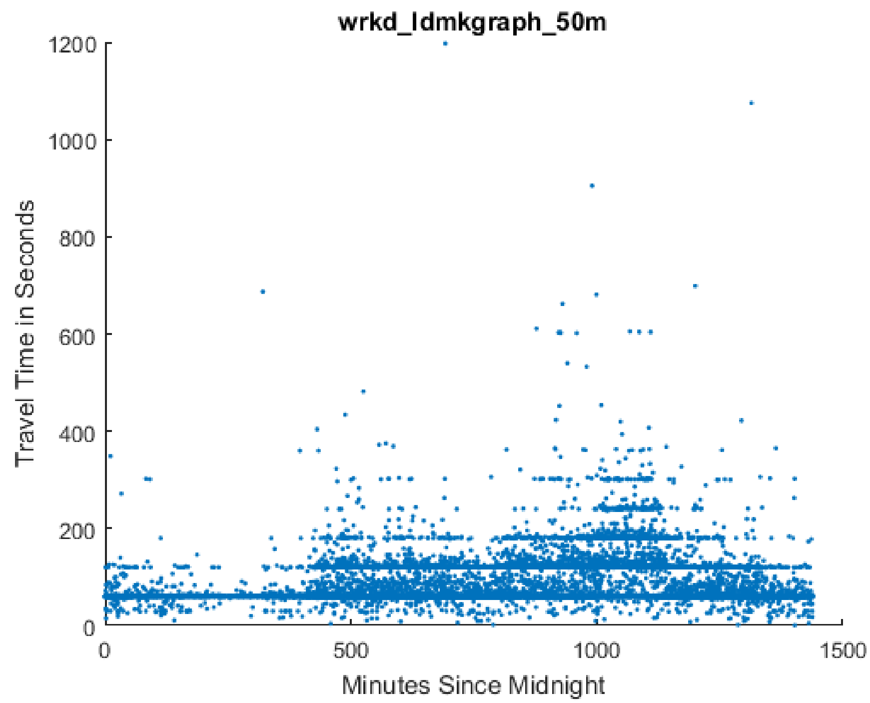


Figure 4-1: An example of travel time patterns

Figure 4-1 illustrates a scatter plot of the travel time of a particular landmark graph edge during the course of a weekday. It can be observed that the travel time does not seem to be a single-valued function with respect to time of the day, as one may expect; rather, the scatter points tend to gather around some values and form some *clusters*. For instance, when it is 500 minutes since midnight, namely 8:20 a.m., the travel time seems to have three main clusters which are represented by three horizontal lines formed by the scatter points. When it is 1,000 minutes since midnight, namely 4:40 p.m., there are about five such lines. This pattern is attributable to three possible reasons:

1. Drivers may actually choose different routes to travel between the two land-

marks, which cannot be captured by the landmark graph since it only knows a driver has traversed from one landmark to the other but not the exact route. Different routes have different traffic conditions and speed limitations, therefore the travel time varies;

2. Drivers have different driving skills, preferences and behaviours. Some drivers just drive faster than others, even if the road conditions are similar and;

3. The GPS devices on taxis reported locations *periodically*, therefore, durations like 60 seconds or 120 seconds are very commonly seen in the landmark graphs. Even if the *actual* travel time is 53 seconds, it is still recorded as 60 seconds. This corresponds to the low-sampling-rate problem mentioned in Section 1.2.

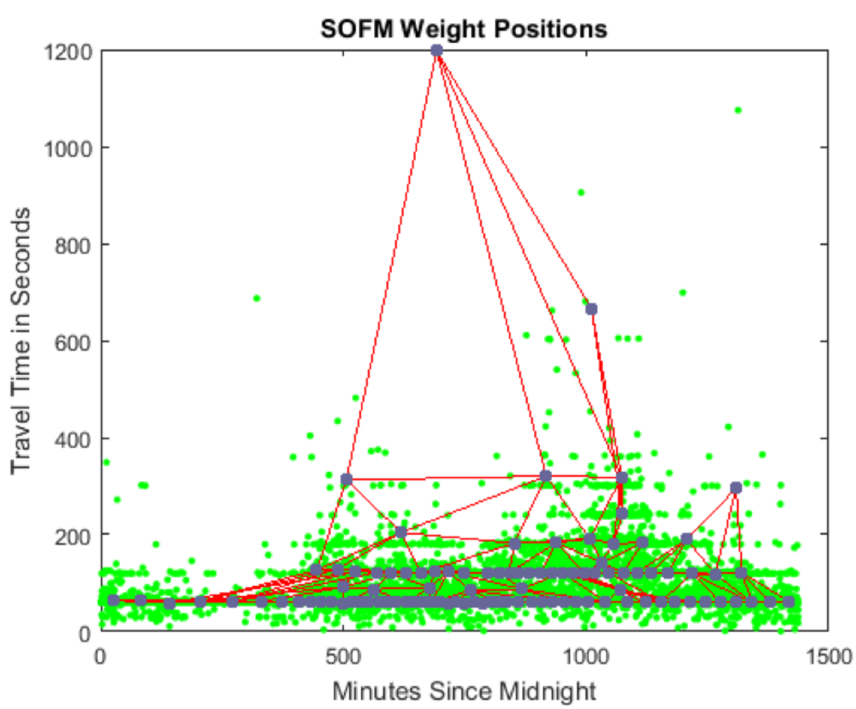


Figure 4-2: A plot of neurons' final positions

Therefore, it is not possible to fit the scatter points with a single-valued function.

Rather, the clustering technique should first be employed to identify the travel time clusters. Like in Section 2.3.2, a **self-organising feature map** is used to cluster the scatter points. Figure 4-2 shows the final positions of the neurons at the end of the training.

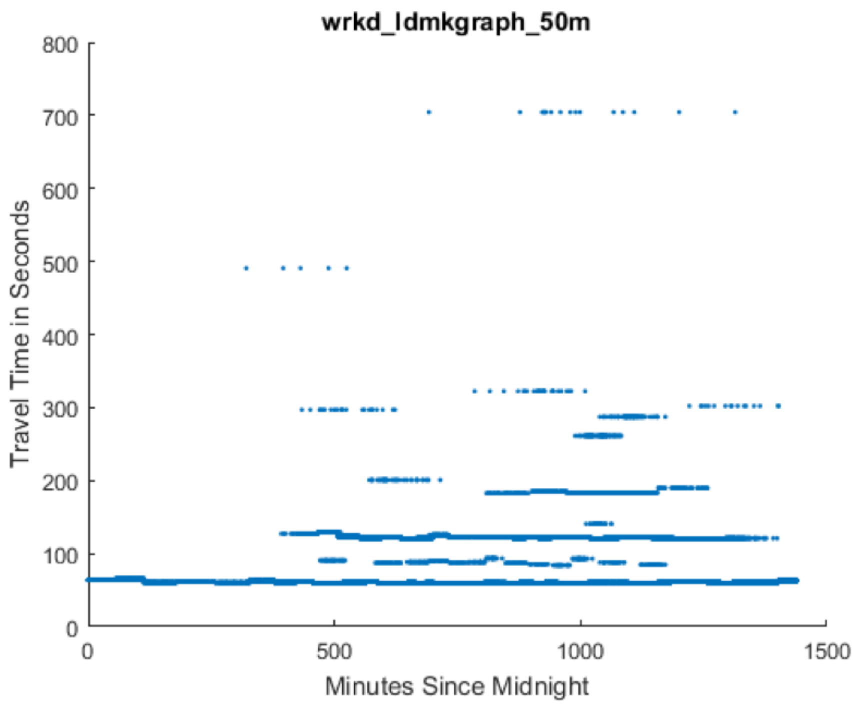


Figure 4-3: An example of representing data points with centroids

One merit of SOFM clustering is *feature extraction*. In this case, after the training is completed, the SOFM has learned some features about the travel time at different times of the day and expressed its understanding by moving its neurons to the centroids of the clusters. Now, **the data points can be represented by their respective centroids**. Figure 4-3 shows the effect of replacing each data point's value with their centroids'. The clusters are clearly shown by the horizontal lines formed by those centroids.

Representing data points with cluster centroids or features learned provides the benefit of *generalisation*. The data set used in this project, albeit large in size, is nevertheless a small *sample* of the *population* of taxi trajectories over time. In other words, these trajectory records are only the *observed* ones but there are potentially infinite number of trajectories that cannot be all observed. The only information known about them is that **they must fit into one of the clusters** after undergoing the same procedures described in the previous chapters. The use of centroids instead of real data points also takes into consideration the unobserved trajectories and reflects the real underlying patterns. In fact, generalisation is an advantage that all neural network-based methods share.

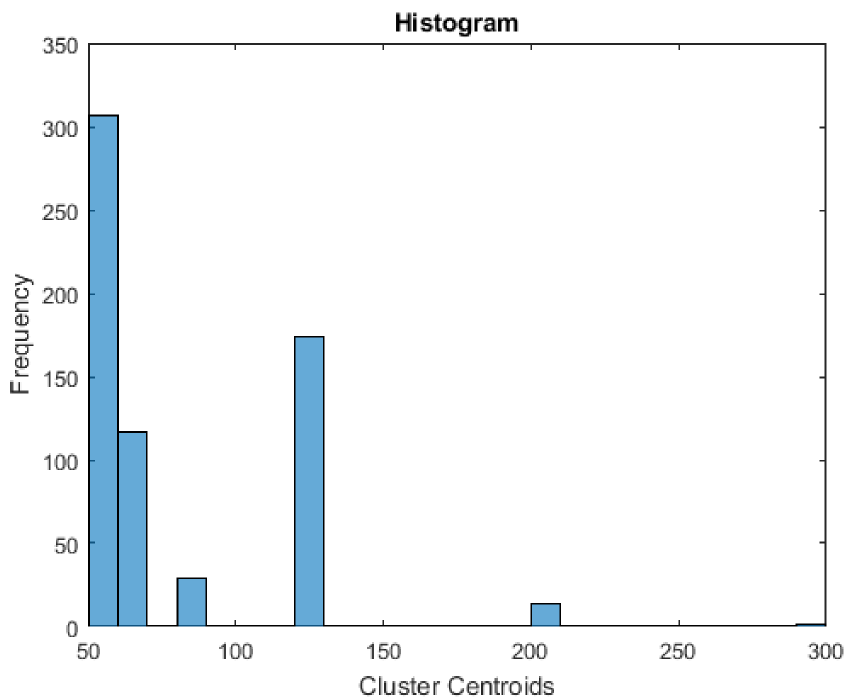


Figure 4-4: An example of distribution of clusters

Now that the clusters are identified, Figure 4-4 shows a histogram of clusters within the 10:00 a.m. to 10:30 a.m. interval. It can be observed that within a particular time interval, there are many clusters of travel time and each cluster has different sizes. To describe the travel time pattern within a particular time interval, the histogram is converted into a cumulative probability distribution of clusters which is then fitted with Weibull Distribution described in Theorem 4.

Theorem 4 (*Weibull Distribution*)**.** A Weibull Distribution is described by two strictly positive parameters, the scale parameter α and the shape parameter β, and has a **probability distribution function** of [5]

$$f(x|\alpha,\beta) = \frac{\beta}{\alpha} \cdot (\frac{x}{\alpha})^{\beta-1} \cdot e^{-(x/\alpha)^{\beta}}. \tag{4.1}$$

Correspondingly, its **cumulative distribution function** is given by [4]

$$F(x|\alpha,\beta) = \int_0^x f(t|\alpha,\beta)dt = 1 - e^{-(x/\alpha)^{\beta}}. \tag{4.2}$$

Figure 4-5 shows an example of fitting a cumulative probability distribution of clusters with Weibull Distribution. The red line represents the cumulative probability distribution of clusters and the blue line indicates the best Weibull Distribution fit estimated by maximum likelihood.

In the referencing paper [15], it uses Inverse Gaussian Distribution to fit the centroid distribution. However, as some experiments showed, Inverse Gaussian Distribution does not always work, especially when the centroids actually have a uniform distribution, namely, when only one cluster appears in the time interval. In that case, Weibull distribution will have the shape parameter $\beta = \infty$ but it still can give the correct result.

The cumulative distribution function of Weibull Distribution is good at calculating the probability whereby the travel time t is less than a particular value. For instance, the probability of the travel time for this landmark graph edge being less than or equal to 100 seconds is approximately $P(t \leq 100) = 0.7$ according to the Weibull

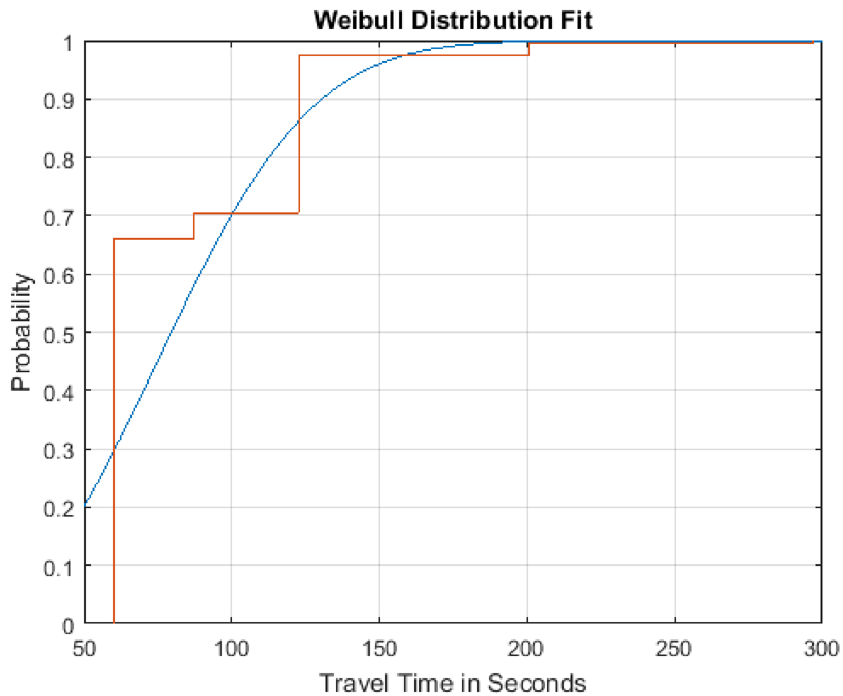


Figure 4-5: An example of fitting data with Weibull Distribution

Distribution curve in Figure 4-5. But to estimate the travel time of a landmark graph edge is in fact the *reverse*: given a known probability p, find the corresponding travel time t, which is exactly the inverse Weibull cumulative distribution function does.

Theorem 5 (*Inverse Weibull Cumulative Distribution Function*)**.** The inverse Weibull cumulative distribution function is defined as

$$F^{-1}(x|\alpha,\beta) = G(p) = \alpha(-ln(1-p))^{1/\beta} \tag{4.3}$$

where p is the given probability.

In practice, the probability p corresponds to some subjective measures of a driver's driving skill. It represents how fast a driver drives as compared to other drivers. If a driver has a p value of 0.2, it means that this driver usually drives faster than 80% $(1 - 0.2)$ drivers. This concept is formally defined in Definition 11.

Definition 11 (*Optimism Index*)**.** An optimism index p indicates how optimistic a driver feels about his or her driving skills [15]. A driver with an optimism index of p_0% usually drives faster than $(1 - p_0)$% drivers under similar road conditions.

The optimism index of a driver can be set by the driver in advance, although it may be subject to the illusory superiority problem[1]. It can also be set to the mean value by default or learned from the driver's trajectory history.

With optimism index defined, it is easy to estimate the travel time using Theorem 5. However, it is worth mentioning that for a cumulative probability distribution function $F(x)$, the probability of a single value is zero, namely, $F(x_0) = 0$; only a range of values can have positive probabilities. Therefore, the travel time calculated from optimism index should really be a *range* of travel time. For instance, if the optimism index is $p = 0.7$, the correct result should be $t \leq 100$ based on Figure 4-5, which means that for a driver with an optimism index of 0.7, his or her travel time is *at most* 100 seconds. No exact value can be determined. However, in the context of this project, only the *worst-case* travel time is considered; therefore, the driver is said to have an expected travel time of 100 seconds.

4.2 Travel Time Evaluation

After the travel time distributions for each significant edge were determined, an evaluation was conducted to ascertain the accuracy of the travel time estimates. In this project, the estimates made by Theorem 5 were compared against real-time estimates made by Baidu Maps at various times of the day.

[1]In a 1986 experiment, *80%* participants rated their driving skills *above-average* [8].

Some complications do arise, however, because the two vertices of each significant edge are actually two *landmarks*. Definition 5 states that a landmark is essentially a road segment with its length ignored. But in estimating the travel time between two landmarks, the length of the landmarks cannot be ignored and has profound effects on how the travel time should be calculated.

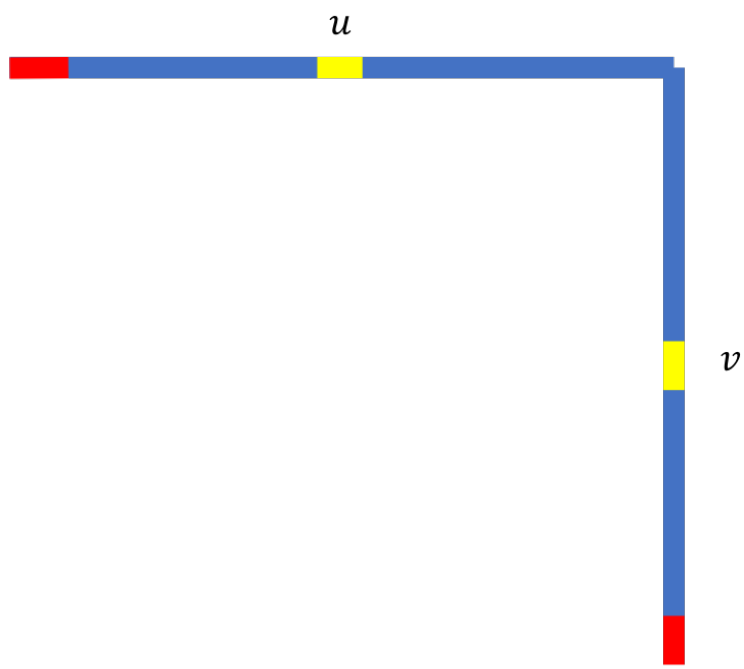


Figure 4-6: An example of different landmark travel time

Imagine there are two landmarks u and v, respectively, and each has a length of L as shown in Figure 4-6. Then, there are potentially infinite number of ways of 'travelling from u to v', because the starting point could be any points on u and the ending point could be any points on v. If a driver starts at the red point on u and stops at the red point on v, the total distance is $2L$ and the total time is T. But had the driver stopped at the yellow mid-point on v, the total distance would be $1.5L$ and the total time would be $3T/4$, assuming the speed is constant.

Therefore, it is difficult to give a *definite* estimate of the travel time between two landmarks. In this project, the **median travel time** is instead used to represent the travel time between two landmarks. It describes the *typical* travel time a driver should expect. An example of a median case is that the diver starts at the yellow mid-point on u and ends at the yellow mid-point on v. But there are in fact infinite number of median cases.

The most significant 150 edges were selected to be evaluated, from *each* of the four landmark graphs listed in Table 4.1. For each edge (u, v), 20% of the trajectories with u as starting point and v as ending point were randomly sampled. These trajectories represent some random trips between u and v with varying distances. Baidu Maps API was then used to estimate the travel time of each trajectory in real time, after which the median of all the estimates was selected as Baidu's estimate for the travel time between u and v. Finally, Theorem 5 was used to calculate its own the travel time estimate which was then compared against Baidu's. Table 4.2 summarises the evaluation results for the 150 significant edges selected from each landmark graph.

Landmark Graph	**RMSE**	**Mean Error Ratio**	**Mean No. of Samples Per Edge**
wrkd_ldmkgraph_50m	78.84	-0.009	1824.60
wrkd_ldmkgraph_30m	87.96	-0.065	1507.56
holi_ldmkgraph_50m	87.39	-0.16	832.96
holi_ldmkgraph_30m	76.41	-0.14	681.89

Table 4.2: A summary of evaluation results

Assuming a Baidu's estimate is b_i and the corresponding project's estimate is $\hat{b}_i$, then the Root Mean Square Error (RMSE) is given by:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(\hat{b}_i - b_i)^2}{N}} \tag{4.4}$$

where N is the total number of estimates; and the Mean Error Ratio (MER) is given by:

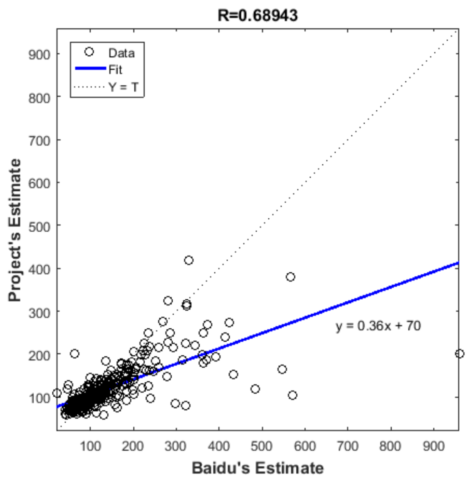
$$MER = \frac{1}{N}\sum_{i=1}^{N}\frac{\hat{b}_i - b_i}{b_i} \tag{4.5}$$

The RMSE gives an overall measure of accuracy. For instance, the RMSE for landmark graph *wrkd_ldmkgraph_50m* turned out to be 78.84, which means that on average, the difference between Baidu's estimates and Theorem 5's estimate was 78.84 seconds. But whether this difference is significant or not depends on the scale of Baidu's estimates. For a Baidu's estimate of 30 seconds, 78.84 seconds may be considered significantly high, but for an estimate of 200 seconds, it may be acceptable.

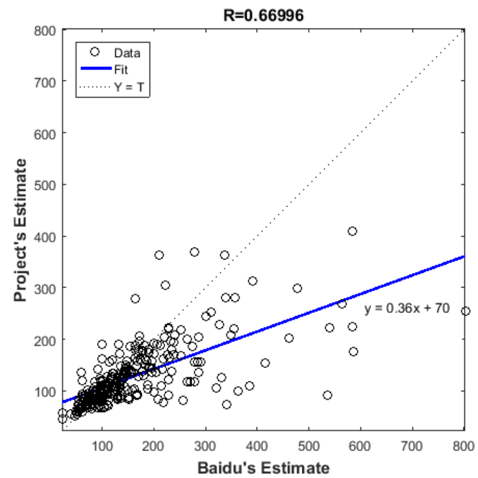
Therefore, MER is designed to take into account the scale of Baidu's estimates. It computes the ratio between the error and Baidu's estimate. A positive MER indicates that on average Theorem 5 tends to have larger estimates than Baidu's; on the other hand, a negative MER indicates that on average, Theorem 5 tends to have smaller estimates than Baidu's.

Another technique to gauge Theorem 5's performance is linear regression. An ordered pair of estimates $(b_i, \hat{b}_i)$ can be treated as a point in a 2-D Cartesian system. In the *ideal* case, $\hat{b}_i$ should be equal to b_i, therefore, all points should form a straight line with a slope of 1 and an intercept of 0 when plotted. In the general cases, the regression equation gives the relationship between b_i and $\hat{b}_i$. If the slope of the regression equation is less than 1, then $\hat{b}_i$ is expected to be less than b_i in *most* cases depending on the intercept. Figure 4-7 shows the regression plots.

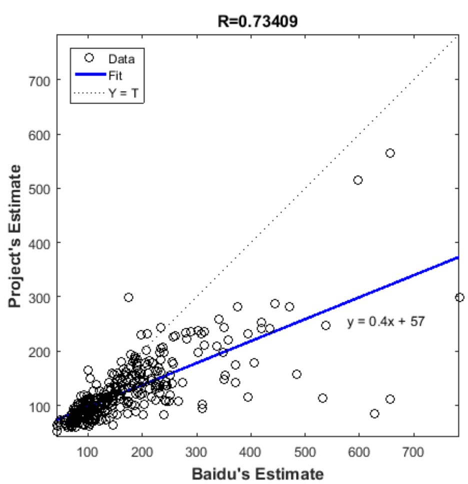
In summary, the estimates Theorem 5 gave were smaller than Baidu's estimates in most cases, which to some extent, was actually expected. The data set was collected in 2009 but Baidu Maps was giving *real-time* estimates. Eight years since then, the travel time in the entire road network should have increased in general, due to increase in car ownership. Nevertheless, it still provides reasonable estimates with a worst-case RMSE of 87.96 seconds.



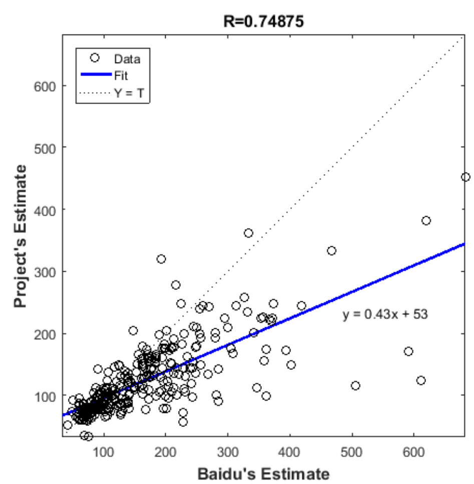
(a) wrkd_ldmkgraph_50m



(b) wrkd_ldmkgraph_30m



(c) holi_ldmkgraph_50m



(d) holi_ldmkgraph_30m

Figure 4-7: A plot of regressions

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Time-Dependent Shortest Path Calculation

Definition 12 (*FIFO Graph*)**.** A time-dependent graph $G = (V, E)$ with a dynamic weight function $w : E, t \to \mathbb{R}$ is a FIFO graph [3] iff for every edge $(u, v) \in E$

$$\forall \Delta t \geq 0, \quad w(u, v, t_0) \leq \Delta t + w(u, v, t_0 + \Delta t) \tag{5.1}$$

Chapter 4 discusses how to estimate the time-dependent travel cost of each significant edge. Once the time-varying patterns of edge costs are determined, calculating a shortest path in a landmark graph is straightforward, provided the landmark graph is a FIFO graph[1] as defined in Definition 12.

An equivalent way of expressing FIFO property is that, if a person leaves vertex u at time t_0, then any persons who leave vertex u at a later time $t_0 + \Delta t$ will not be able to reach vertex v earlier than the first person. In practice, transport networks are said to have FIFO property [15], therefore, a landmark graph can be considered as a FIFO graph. Then the Dijkstra's shortest path algorithm can be applied directly but

[1] If the graph is not a FIFO graph, the problem is NP-hard if waiting is not allowed at any vertices

with a modification that the edge costs are computed dynamically as the algorithm proceeds. Figure 5-1 shows an example of dynamically updating edge costs.

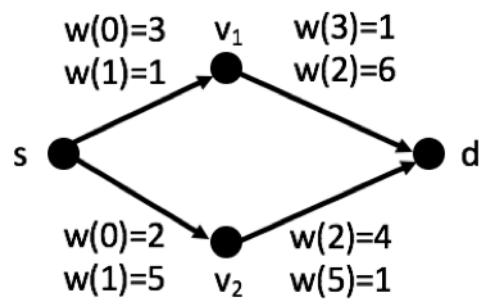


Figure 5-1: An example of updating edge costs dynamically

If a taxi leaves s at time $t = 0$, the edge (s, v_1) and (s, v_2) will have a time-dependent travel time of 3 and 2, respectively. If the taxi follows the current fastest edge (s, v_2), when it arrives at v_2 the time-dependent travel time of the edge (v_2, d) at time $t = 2$ will be 4, giving a total travel time 6. But should the taxi follow the edge (s, v_1), the time-dependent travel time of the edge (v_1, d) at $t = 3$ would be 1, giving a total travel time of 4. Therefore, path $s \rightsquigarrow v_1 \rightsquigarrow d$ is the time-dependent shortest path. The same process applies when the taxi leaves s at time $t = 1$, but the time-dependent shortest path will be path $s \rightsquigarrow v_2 \rightsquigarrow d$. This example is also a FIFO graph: path $s \rightsquigarrow v_1 \rightsquigarrow d$ would still be the shortest path, should another taxi *wait* at s until $t = 1$. Listing 5.1 gives the pseudocode for the modified Dijkstra's Algorithm.

Listing 5.1: Pseudocode for Modified Dijkstra's Algorithm

```
Input: a landmark graph G = (V, E), a source s and a destination d
Ouput: a predecessor graph G_p = (V_p, E_p)
// EAT = Earliest Arrival Time
predecessor = dict{s : None}
pq = queue.PriorityQueue()
s.EAT = 0
```

```
pq.put(s)
for vertex in G.V − {s}:
    vertex.EAT = ∞
    pq.put(vertex)
while not pq.empty():
    hd = pq.get() // get the vertex at the queue head
    for v in hd.neighbours:
        if hd.EAT + w(hd, v, hd.EAT) < v.EAT:
            v.EAT = hd.EAT + w(hd, v, hd.EAT)
            predecessor[v] = hd
```

Upon termination of the algorithm in Listing 5.1, a predecessor graph whose edges are *reversed* shortest paths from s to other vertices are constructed. Listing 5.2 provides a recursive method to print out the shortest path from s to d.

Listing 5.2: Pseudocode for Printing Shortest Paths

```
Input: a predecessor graph G_p = (V_p, E_p) and a destination d
Ouput: a shortest path s ⇝ d

def print_path(predecessor, dst):
    pred = predecessor[dst]
    if pred is None:
        print(dst.name)
    else:
        print_path(predecessor, pred)
        print('→' + dst.name)

print_path(predecessor, d)
```

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Conclusion

This paper presents an alternative approach for finding a shortest path in a time-dependent road network, based on GPS trajectories aggregated from thousands of taxis in Beijing, China. With the help of existing developer APIs from Baidu Maps, each GPS data point is mapped to a real road segment based on its longitude and latitude. An innovative SOFM-based approach is proposed for identifying and removing outliers. The trajectories are then separated into individual trips and frequently visited streets in each trip are chosen as landmarks. Several time-dependent graphs with landmarks as vertices are constructed as an abstract representation of Beijing's road network. SOFM and Weibull Distribution are employed to estimate the travel time of each landmark graph edge. The estimates are compared with real-time estimates made by Baidu Maps. The worse-case RMSE is 87.96 seconds and the worst-case MER is -0.16, which indicates that this approach provides reasonable estimates as compared with Baidu's and has the potential for everyday use. A modified Dijkstra's algorithm is proposed for calculating shortest paths based on the estimates.

This data-driven approach harvests the collective intelligence from thousands of taxi drivers who possess implicit knowledge about the time-dependent shortest paths in a road network.

Appendix A

Source Code

A.1 Database Utilities

The source code listed in this section provides utility functions for general database operations, such as **select** and **update**. It is to be used in other modules.

Listing A.1: Database Utilities

```
<?php
include "../inc/jingodbinfo.inc";

function connect_db(){
    $conn = new mysqli(DB_SERVER, DB_USERNAME, DB_PASSWORD);
    if($conn->connect_error){
        die("Connection failed: " . $conn->connect_error);
    }
    $conn->select_db(DB_DATABASE);
    $conn->set_charset("utf8");
    return $conn;
}

```

```
function disconnect_db($conn){ $conn->close(); }

function db_select($conn, $table, $cols, $cond = "true", $distinct = "")
{
    $sql_select = "SELECT {$distinct} ";
    if(count($cols) == 0){
        $sql_select .= "*,";
    }
    foreach ($cols as $col) {
        $sql_select .= "{$col},";
    }
    $sql_select = rtrim($sql_select, ",");
    $sql_select .= " FROM {$table} WHERE {$cond};";
    $ret = $conn->query($sql_select);
    $res = array();
    if($ret->num_rows > 0){
        while($row = $ret->fetch_assoc()){
            $curr = array();
            foreach ($cols as $col) {
                $curr[$col] = $row[$col];
            }
            $res[] = $curr;
        }
    }
    return $res;
}

function db_update($conn, $table, $values, $cond){
    $sql_update = "UPDATE {$table} SET ";
    foreach ($values as $key => $value) {
        if(is_numeric($value)){
            $sql_update .= "{$key} = {$value},";
```

```
        }else{
            $sql_update .= "{$key} = '{$value}',";
        }
    }
    $sql_update = rtrim($sql_update, ",");
    $sql_update .= " WHERE {$cond};";
    $succ = $conn->query($sql_update);
    return $succ;
}

function db_insert($conn, $table, $values, $cond = "", $ignore = ""){
    $sql_insert = "INSERT {$ignore} INTO {$table} ";
    $cols = "";
    $vals = "";
    foreach ($values as $key => $value) {
        $cols .= "{$key},";
        $vals .= (is_numeric($value) ? "{$value}," : "'{$value}',");
    }
    $cols = rtrim($cols, ",");
    $vals = rtrim($vals, ",");
    $sql_insert .= "({$cols}) VALUES ({$vals}) {$cond};";
    echo "{$sql_insert}\n";
    $conn->query($sql_insert);
}

function db_delete($conn, $table, $cond){
    $sql_delete = "DELETE FROM {$table} WHERE {$cond};";
}
?>
```

A.2 Data Pre-processing

This section lists down the source code used in data pre-processing.

Listing A.2: Outlier Removal

```
<?php
include "db_utilities.php";

class Filter{
    private $ldmk_start;
    private $ldmk_end;
    private $ldmk_table;
    private $data_table;
    private $update_table;
    private $path;
    private $bchmk;

    private $conn;
    private $num_dele;

    public function __construct($ldmk_start, $ldmk_end, $ldmk_table,
        $data_table, $path, $bchmk, $update_table){
        $this->ldmk_start = $ldmk_start;
        $this->ldmk_end = $ldmk_end;
        $this->ldmk_table = $ldmk_table;
        $this->data_table = $data_table;
        $this->update_table = $update_table;
        $this->path = $path;
        $this->bchmk = $bchmk;
        $this->conn = connect_db();
        $this->num_dele = 0;
    }
```

```

    public function __destruct(){ disconnect_db($this->conn); }

    private function getCentres($ldmk){
        $filename = "{$this->path}{$ldmk}.csv";
        $centres = array();
        if(($handle = fopen($filename, 'r')) !== FALSE){
            while(($line = fgetcsv($handle, 0, ',')) !== FALSE){
                $centres[] = array($line[0], $line[1]);
            }
        }
        return $centres;
    }

    private function getRecords($ldmk){
        $ldmk_cols = array('LandmarkName');
        $ldmk_cond = "LandmarkID = {$ldmk}";
        $ret = db_select($this->conn, $this->ldmk_table, $ldmk_cols,
            $ldmk_cond);
        $data_cols = array('DataUnitID', 'BD09_LONG', 'BD09_LAT');
        $data_cond = "Street = '{$ret[0][$ldmk_cols[0]]}'";
        $ret = db_select($this->conn, $this->data_table, $data_cols,
            $data_cond);
        $records = array();
        foreach ($ret as $item) {
            $records[] = array($item[$data_cols[0]], $item[$data_cols
                [1]], $item[$data_cols[2]]);
        }
        return $records;
    }


```

```
    private function removeRecord($centres, $records){
        foreach ($records as $recd) {
            $min_dist = INF;
            foreach ($centres as $centre) {
                $min_dist = min($min_dist, $this->getHaversineDist
                ($centre[0], $centre[1], $recd[1], $recd[2]));
            }
            if($min_dist > $this->bchmk){
                $cond = "DataUnitID = {$recd[0]}";
                db_delete($this->conn, $this->data_table, $cond);
                ++$this->num_dele;
            }
        }
    }

    private function toRadian($degree){ return $degree * M_PI / 180; }

    private function getHaversineDist($long1, $lat1, $long2, $lat2){
        $BJ_LAT = $this->toRadian(39);
        $EQ_R = 6378137; // equatorial radius in metres
        $POL_R = 6356752; // polar radius in metres

        $BJ_R = sqrt((pow($EQ_R * $EQ_R * cos($BJ_LAT), 2) + pow($POL_R
             * $POL_R * sin($BJ_LAT), 2)) / (pow($EQ_R * cos($BJ_LAT), 2)
             + pow($EQ_R * sin($BJ_LAT), 2)));
        $long1 = $this->toRadian($long1);
        $lat1 = $this->toRadian($lat1);
        $long2 = $this->toRadian($long2);
        $lat2 = $this->toRadian($lat2);
        $d = 2 * $BJ_R * asin(sqrt(pow(sin(($lat1 - $lat2) / 2), 2) +
        cos($lat1) * cos($lat2) * pow(sin(($long1 - $long2) / 2), 2)));
        return $d; }
```

```
    private function getWithin($ldmk, $centres, $records){
        foreach ($records as $recd) {
            $min_dist = INF;
            foreach ($centres as $centre) {
                $min_dist = min($min_dist,
                $this->getHaversineDist($centre[0], $centre[1], $recd
                    [1], $recd[2]));
            }
            if($min_dist <= $this->bchmk){
                db_insert($this->conn, $this->update_table,
                        array('LandmarkID' => $ldmk, 'BD09_LONG' =>
                            $recd[1], 'BD09_LAT' => $recd[2]));
            }
        }
    }

    public function doFiltering(){
        for($ldmk = $this->ldmk_start; $ldmk != $this->ldmk_end;
                ++$ldmk){
            $centres = $this->getCentres($ldmk);
            $records = $this->getRecords($ldmk);
            $this->getWithin($ldmk, $centres, $records);
        }
    }
}
set_time_limit(0);
ini_set('memory_limit','2048M');
$filter = new Filter($_POST['ldmk_start'], $_POST['ldmk_end'],
        $_POST['ldmk_table'], $_POST['data_table'], $_POST['path'],
        $_POST['bchmk'], $_POST['update_table']);
$filter->doFiltering();
?>
```

A.3 Landmark Graph Construction

This section presents algorithms related to landmark graph construction.

Listing A.3: Trip Identification

```
<?php
include "db_utilities.php";

class IdentifyTrip{
    private $cuid_start;
    private $cuid_end;
    private $table;
    private $tripid;
    private $threshold;
    private $occup;
    private $conn;

    public function __construct($cuid_start, $cuid_end, $table, $tripid,
         $threshold, $occup){
        $this->cuid_start = $cuid_start;
        $this->cuid_end = $cuid_end;
        $this->table = $table;
        $this->tripid = $tripid;
        $this->threshold = $threshold;
        $this->occup = $occup;
        $this->conn = connect_db();
    }

    public function __destruct(){ disconnect_db($this->conn); }



```

```
    public function startIdentifyTrip(){
        $cols = array('DataUnitID', 'UnixEpoch');
        for($cuid = $this->cuid_start; $cuid != $this->cuid_end;
                ++$cuid){
            $cond = "CUID = {$cuid} and Occupied = {$this->occup}";
            $res = db_select($this->conn, $this->table, $cols, $cond);
            if(count($res) > 0){
                $this->splitTrip($res, $cols);
            }
        }
    }

    public function splitTrip($res, $cols){
        $last = $curr = $res[0][$cols[1]];
        foreach ($res as $item) {
            $curr = $item[$cols[1]];
            if($curr - $last > $this->threshold){
                ++$this->tripid;
            }
            $values = array('TripID' => $this->tripid);
            $cond = "{$cols[0]} = {$item[$cols[0]]}";
            $succ = db_update($this->conn, $this->table, $values,
                $cond);
            $last = $curr;
        }
        ++$this->tripid;
    }
}
set_time_limit(0); ini_set('memory_limit','2048M');
$identifyTrip = new IdentifyTrip($_POST['start'], $_POST['end'], $_POST['table'], $_POST['tripid'], $_POST['threshold'], $_POST['occup']);
$identifyTrip->startIdentifyTrip();?>
```

Listing A.4: Landmark Frequency

```
<?php
include "db_utilities.php";
function insertLandmark($conn, $table, $landmark){
    $values = array("LandmarkName" => $landmark, "LandmarkCount" => 1);
    $cond = "ON DUPLICATE KEY UPDATE LandmarkCount = LandmarkCount + 1";
    $succ = db_insert($conn, $table, $values, $cond);
}
function fetchLandmark($conn, $table, $tripid){
    $cols = array("Street");
    $cond = "TripID = {$tripid} group by Street";
    $res = db_select($conn, $table, $cols, $cond);
    $streets = array();
    foreach ($res as $item) {
        foreach ($item as $key => $value) {
            $streets[] = $value;
        }
    }
    return $streets;
}
set_time_limit(0); ini_set('memory_limit','2048M');
$conn = connect_db();
$start = $_POST['tripid_start']; $end = $_POST['tripid_end'];
$data_table = $_POST['dtable']; $landmark_table = $_POST['ltable'];
for($tripid = $start; $tripid != $end; ++$tripid){
    $landmarks = fetchLandmark($conn, $data_table, $tripid);
    foreach ($landmarks as $ldmk) {
        $succ = insertLandmark($conn, $landmark_table, $ldmk);
    }
}
disconnect_db($conn);?>
```

Listing A.5: Landmark Construction

```
<?php
include "db_utilities.php";

class GraphBuilder{
    private $start;
    private $end;
    private $ldmktable;
    private $triptable;
    private $ldmklimit;
    private $holi_table;
    private $wrkd_table;
    private $conn;
    private $landmarks;

    public function __construct($start, $end, $ldmktable, $triptable,
        $ldmklimit, $holi_table, $wrkd_table){
        $this->start = $start;
        $this->end = $end;
        $this->ldmktable = $ldmktable;
        $this->triptable = $triptable;
        $this->ldmklimit = $ldmklimit;
        $this->holi_table = $holi_table;
        $this->wrkd_table = $wrkd_table;
        $this->conn = connect_db();
        $this->landmarks = $this->fetchldmk();
    }

    public function __destruct(){ disconnect_db($this->conn); }



```

```
    private function fetchldmk(){
        $cols = array('LandmarkName', 'LandmarkID');
        $cond = "{$cols[1]} <= {$this->ldmklimit}";
        $res = db_select($this->conn, $this->ldmktable, $cols, $cond);
        $ldmks = array();
        foreach ($res as $item) {
            $ldmks[$item[$cols[0]]] = $item[$cols[1]];
        }
        return $ldmks;
    }

    private function isLandmark($street){
        return array_key_exists($street, $this->landmarks);
    }

    private function isHoliday($atime){
        $date = date('d', $atime);
        $day = date('w', $atime);
        if($date == '01' || $date == '28' || $date == '29'){
            return true;
        }else if($date == '31'){
            return false;
        }else if($day == 0 || $day == 6){
            return true;
        }
        return false;
    }
```

```
    public function buildGraph(){
        $street_cols = array("Street");
        $utc_cols = array("UnixEpoch");

        for($tripid = $this->start; $tripid != $this->end; ++$tripid){
            $street_utc = array();
            $street_cond = "TripID = {$tripid}";
            $streets = db_select($this->conn, $this->triptable, $street_cols, $street_cond, "DISTINCT");
            foreach ($streets as $item) {
                $street = $item[$street_cols[0]];
                $utc_cond = "{$street_cols[0]} = '{$street}' AND {$street_cond} LIMIT 1";
                $ret = db_select($this->conn, $this->triptable, $utc_cols, $utc_cond);
                if(count($ret) > 0){
                    $street_utc[$street] = $ret[0][$utc_cols[0]];
                }
            }
            $this->addEdge($street_utc, $tripid);
        }
    }

    private function addEdge($street_utc, $tripid){
        $cols = array("LandmarkU", "Intermediate", "LandmarkV", "ArrivalTime", "LeavingTime", "Duration", "TripID");
        $streets = array_keys($street_utc);
        $size = count($streets);
        $low = 0;
        while($low < $size && !$this->isLandmark($streets[$low])){
            ++$low;
        }
```

```
        $landmarkU = $landmarkV = $low < $size ? $streets[$low] : NULL;
        $atime = $ltime = $low < $size ? $street_utc[$landmarkV] : NULL;
        ++$low;
        while($low < $size){
            $inbetween = "";
            while($low < $size && !$this->isLandmark($streets[$low])){
                $inbetween .= "{$streets[$low]}-";
                ++$low;
            }
            $inbetween = rtrim($inbetween, "-");
            $landmarkV = $low < $size ? $streets[$low] : NULL;
            $ltime = $low < $size ? $street_utc[$landmarkV] : NULL;
            if(!is_null($landmarkU) && !is_null($landmarkV)){
                $vals = array($landmarkU, $inbetween, $landmarkV, $atime
                    , $ltime, $ltime - $atime, $tripid);
                if($this->isHoliday($atime)){
                    db_insert($this->conn, $this->holi_table,
                        array_combine($cols, $vals));
                }else{
                    db_insert($this->conn, $this->wrkd_table,
                        array_combine($cols, $vals));
                }
            }
            $landmarkU = $landmarkV;
            $atime = $ltime;
            ++$low;
        }
    }
}

set_time_limit(0);
ini_set('memory_limit','2048M');
```

```
date_default_timezone_set("Asia/Singapore");

$graphBuilder = new GraphBuilder($_POST['tripid_start'], $_POST['
    tripid_end'], $_POST['ldmktable'], $_POST['triptable'], $_POST['
    ldmklimit'], $_POST['holi_table'], $_POST['wrkd_table']);

$graphBuilder->buildGraph();
?>
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, third edition ed., MIT Press, Cambridge, MA, 2009.

[2] E. W. Dijkstra, *A note on two problems in connections with graphs*, Numerische Mathematik **1** (1959), 269–271.

[3] Bolin Ding, Jeffrey Xu Yu, and Lu Qin, *Finding time-dependent shortest paths over large graphs*, Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008), ACM, January 2008, pp. 205–216.

[4] The MathWorks Inc., *Weibull cumulative distribution function.*

[5] ______, *Weibull distribution.*

[6] Teuvo Kohonen, *Self-organized formation of topologically correct feature maps*, Biological Cybernetics (1982), 43, 59–69.

[7] Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang, *Map-matching for low-sampling-rate gps trajectories*, ACM SIGSPATIAL GIS 2009, ACM SIGSPATIAL GIS 2009, November 2009.

[8] Iain A. McCormick, Frank H. Walkey, and Dianne E. Green, *Comparative perceptions of driver ability—a confirmation and expansion*, Accident Analysis and Prevention **18** (1986), 205–208.

[9] NorthEast SAS Users Group, *Calculating geographic distance: Concepts and methods*, 2006.

[10] Lior Rokach and Oded Maimon, *Data mining and knowledge discovery handbook*, ch. 15, pp. 321–352, Springer US, 2005.

[11] Frank van Diggelen and Per Enge, *The world's first gps mooc and worldwide laboratory using smartphones*, Proceedings of the 28th International Technical

Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2015) (Tampa, Florida), September 2015, pp. 361–369.

[12] Wikipedia, *Earth radius*, March 2017.

[13] ______, *Restrictions on geographic data in china*, March 2017.

[14] David R. Williams, *Earth fact sheet*, December 2016.

[15] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang, *T-drive: Driving directions based on taxi trajectories*, ACM SIGSPATIAL GIS 2010, November 2010.

[16] Bing Zhu, Peter Huang, Leo Guibas, and Lin Zhang, *Urban population migration pattern mining based on taxi trajectories*, Mobile Sensing Workshop at CPSWeek 2013 (Philadelphia), April 2013.