NANYANG TECHNOLOGICAL UNIVERSITY

SCE16-0446
Time-Dependent Shortest Path Queries on Mobile Devices

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
2017

SCE16-0446

Time-Dependent Shortest Path Queries on Mobile Devices

by

Wei Yumou

Submitted to the School of Computer Science and Engineering
on 27 March 2017, in partial fulfillment of the
requirements for the degree of
Bachelor of Computer Science

Abstract

In this thesis, I designed and implemented a compiler which performs optimizations that reduce the number of low-level floating point operations necessary for a specific task; this involves the optimization of chains of floating point operations as well as the implementation of a “fixed” point data type that allows some floating point operations to simulated with integer arithmetic. The source language of the compiler is a subset of C, and the destination language is assembly language for a micro-floating point CPU. An instruction-level simulator of the CPU was written to allow testing of the code. A series of test pieces of codes was compiled, both with and without optimization, to determine how effective these optimizations were.

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

Acknowledgments

I would like to express my special thanks of gratitude to my FYP supervisor (Assoc Prof. Xiao Xiaokui) who gave me the golden opportunity to do this wonderful project on the topic (GPS Trajectory Mining), which also helped me in doing a lot of Research and i came to know about so many new things I am really thankful to them. Secondly i would also like to thank my parents and friends who helped me a lot in finalising this project within the limited time frame.

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 practically shortest route 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 shortest routes can be challenging, especially when the road traffic is known to be *time-dependent* or *dynamic*, namely, when the road conditions change 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, 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 of these road segments. It has a weight function $w : E, t \to \mathbb{R}$, where t represents a time instant.

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:

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

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 the abstract 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 Apple Maps are able to recommend shortest routes by incorporating real-time traffic information. This project seeks to investigate an alternative approach of 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 two describes the preliminary data processing. Chapter three introduces the approach for building a landmark graph. Chapter four explains how to estimate the travel time for each landmark graph edge. Chapter five presents methods for evaluating travel time estimations.

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

[1]Global Positioning System

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 during 6 p.m. to 7 p.m. on a particular street and hence avoid travelling on that street during that period whenever possible. But observations of this kind, albeit valuable, are just too subtle to be captured by any general algorithms and oftentimes, even the drivers themselves may not be aware of that.

However, mining their GPS trajectories can reveal 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. Such information typically includes latitudes, longitudes, instantaneous speeds and heading directions. Therefore, the 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 of 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 sorted 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 in the GPS-based approach, since the taxi GPS trajectories do not necessarily cover every part of a city's map. It is likely that there are no trajectories passing through the source and the destination.

Low sampling rate

Taxis report their locations to the central reporting agency at a relatively low rate 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 of the *exact* trajectory that the taxi has moved along. The picture[11] below demonstrates a problem caused by low sampling rate and long inter-sample-point distance.

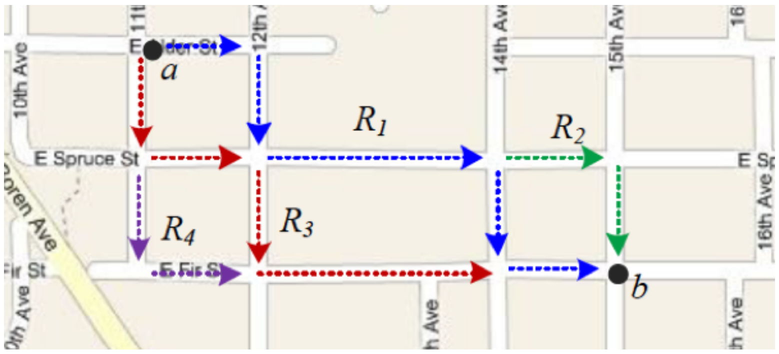

Figure 1-1: Example of low sampling rate problem

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

Limited GPS accuracy

After decades of development, the GPS service has achieved great accuracy, but it is still not completely error-free. A report[7] 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 than this value 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 that of today's.

The limited accuracy in GPS devices makes the exact mapping from a GPS data point to a street impossible. In Beijing, there is usually a side road running in parallel with a main road. Due to that limited accuracy, the taxi might be *actually* on the side road but the GPS data point is shown on the main road, or vice versa.

1.3 Related work

The incentive for carrying out this project comes from a similar project[11], and similar procedures are followed in this project but with some modifications.

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[12] 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. The original 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 the data set 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 difference between the last *non-stationary* data point and the next *non-*

[1]World Geodetic System

Field	**Explanation**
CUID	ID for each taxi
UNIX_EPOCH	Unix timestamp in milliseconds 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: Fields in the original data set

stationary data point.

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 road segment, which is also known as *reverse geocoding*. A number of algorithms[4] 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 APIs[3] 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”[9] where coordinates encoded in WGS-84 format are required by regulations to be shifted by a large and variable amount 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, with the correct location displayed on the right.

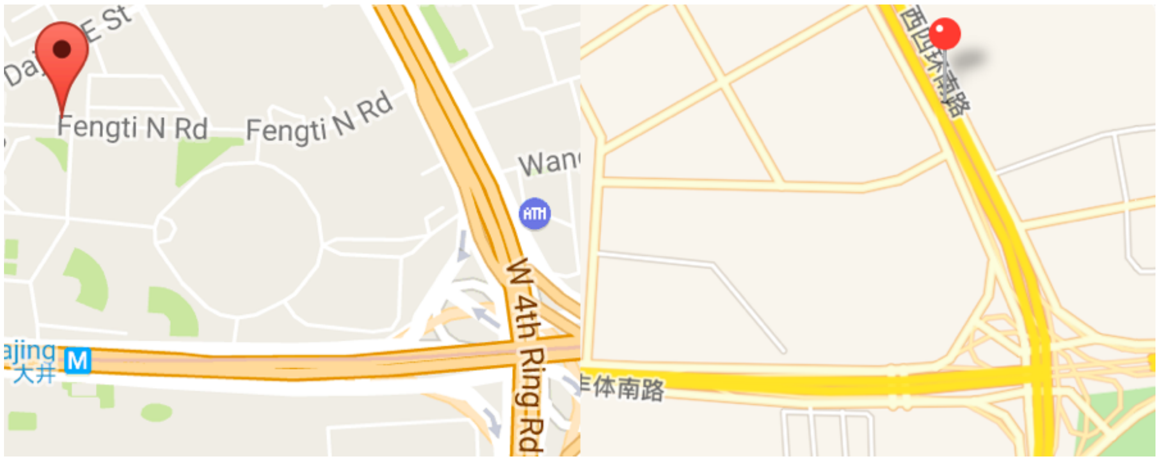

Figure 2-1: China GPS shift problem

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 provides a set of APIs to convert WGS-84 coordinates into BD-09 ones. Therefore, to reverse-geocode the data points, the coordinates must be converted to BD-09 format. To store the converted coordinates 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 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 request to Baidu Maps Web API to get

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: Additional fields added to data set

converted coordinates. Then it updates the database through PHP *mysqli* utilitity.

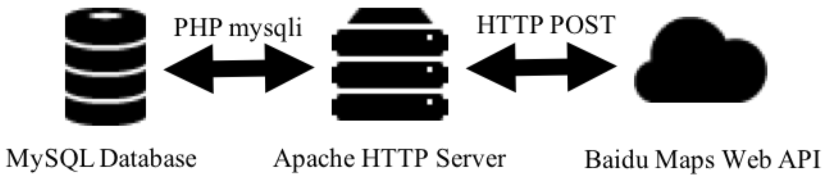

Figure 2-2: Basic system architecture

After the coordinates 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 subsequently sent to the Baidu Maps Web API server *asynchronously* via HTTP GET requests. Once the server responded with the name of the road segment, AJAX

[4] Asynchronous JavaScript and XML

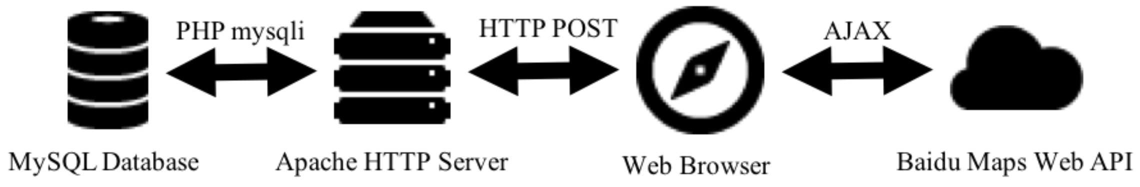

Figure 2-3: Augmented system architecture

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 Motivation 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, the 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 the *entire* road network in Beijing. The actual, correct street is represented in the figure as the *thickest* line on the right half of the figure with a longitude ranging from

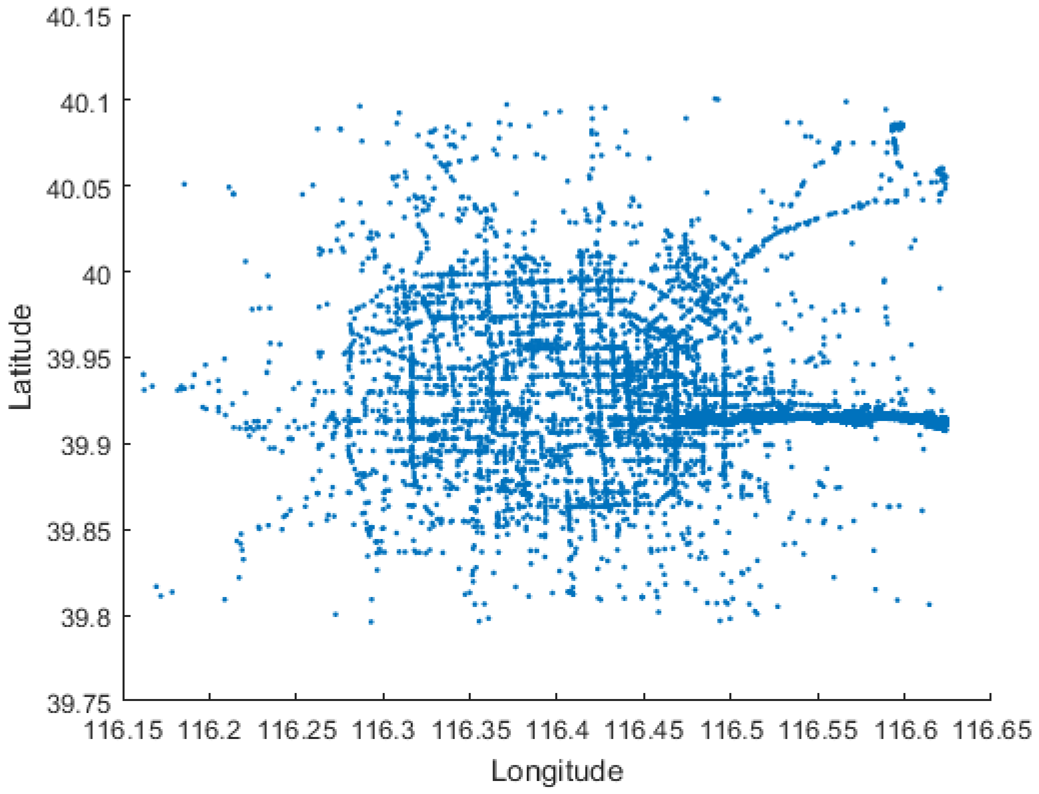

Figure 2-4: Example of outliers

116.45° to 116.65°. 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 a 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 area. Thus, the reasonable reverse geocoder has only achieved an accuracy less 50%, which contradicts the 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 outliers.

Many clustering techniques are available[6]. Since each record can be represented graphically by a point on a 2-D plane with longitute as the x axis and latitude as the y axis, a **self-organising feature map**[3](SOFM) seems to be an appropriate technique to use.

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

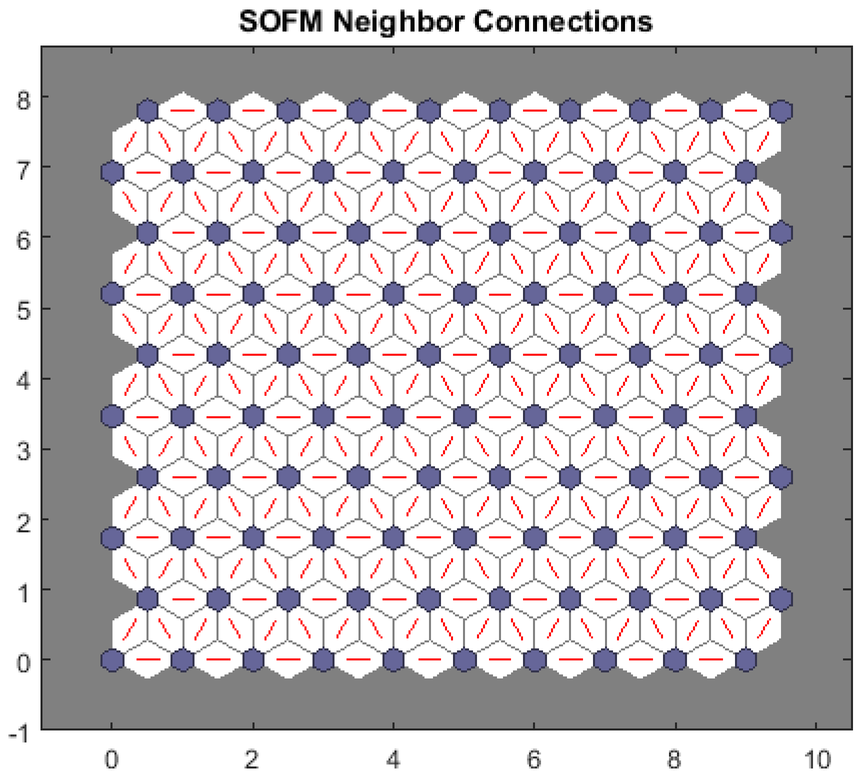

Figure 2-5: An example of SOFM

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, majority of the neurons are clustered to *cover* the correct street they should represent.

[5]Other distance measures are also possible.

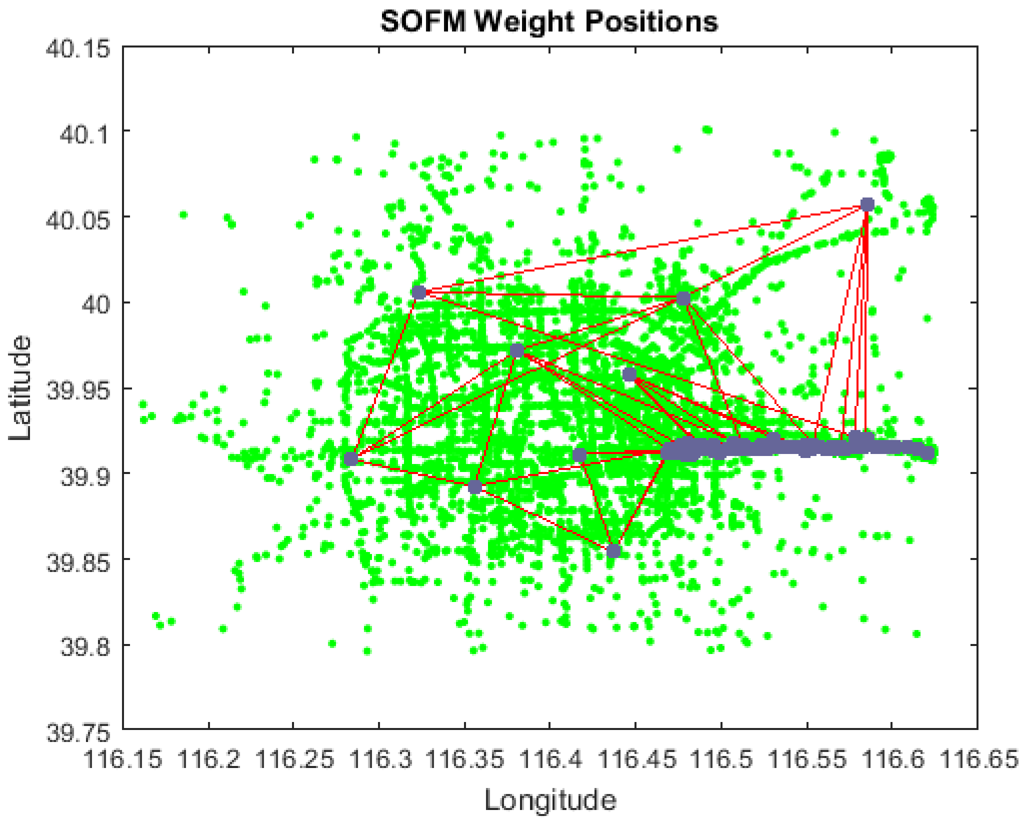

Figure 2-6: Neuron positions after training

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**

centroids 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.

Listing 2.1: Pseudocode for outlier detection

```
for record in records:
        min_distance = math.inf // infinity
        for centroid in centroids:
                min_distance = min(min_distance, \
                        get_distance(record, centroid))
        if min_distance > threshold:
                remove(records, record)
```

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 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[5]

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

where R is the radius of the sphere.

Since the Earth is not a perfect sphere, 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[8]

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

It is known that $R_e = 6,378,137$ metres and $R_p = 6,356,752$ metres on the Earth[10] 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 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 were as least affected as possible by outliers. If the threshold were set to a too small value, the remaining data could not have been sufficient; on the other hand, however, if the threshold were set to a too big value, the accuracy of the final results would have been subject to outliers.

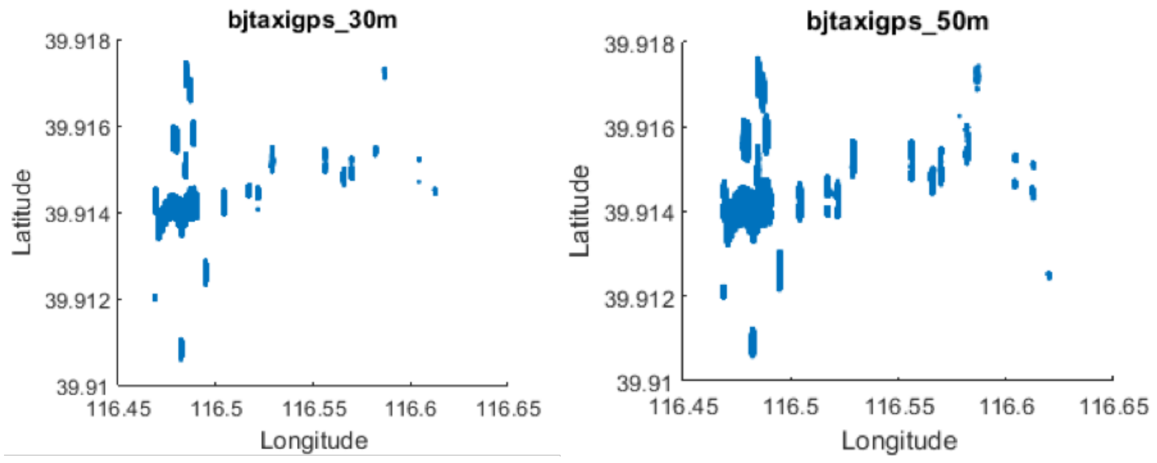

Figure 2-7: Plot of data points after outlier removal

After the outliers were removed, two data sets remained. They are hereinafter referred to as *bjtaxigps_30m*, where outliers were filtered by a threshold of 30 metres and *bjtaxigps_50m*, where outliers were filtered by a threshold of 50 metres, respectively. bjtaxigps_30m contains approximately 51 million records while bjtaxigps_50m has 59 million. All algorithms described hereinafter are applicable to both data sets. Figure 2-7 gives a plot of the data points from both data sets. Clearly, the data points are now contained in a much smaller area and roughly form a shape similar to that of the street they are 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. [11]

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 determining the exact route on which a taxi traversed difficult. Therefore, it is not possible to estimate the time cost for each road segment. However, ignoring the length of a road segment and considering it as an abstract "landmark" make estimating the travel time between two *landmarks* viable.

Moreover, the notion of landmarks closely follows the way how drivers remember the driving routes in daily life[11]. 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 a landmark graph in order to estimate the time 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 that satisfies either condition:

1. A passenger is on the taxi, namely, all records in the set have an OCCUPIED field of value 1 or,
2. No passenger is on board, but the time span between *any* two consecutive records is no larger than 3 minutes.

Table 4.1 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 4.1 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 larger than 3 minute. The next five records constitute another trip, even though the last two records have a time difference 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 4.1 gives the pseudocode for trip identification.

Listing 3.1: Pseudocode for trip identification

```
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) && \
        (curr_unixepoch - last_unixepoch > threshold):
                ++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 10 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 10 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 11.

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 4.2 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 4.2.

Listing 3.2: Pseudocode for counting road frequency

```
frequencies = dict{}
for trip in trips:
        // get unique streets
        streets = unique(trip.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 4-1.

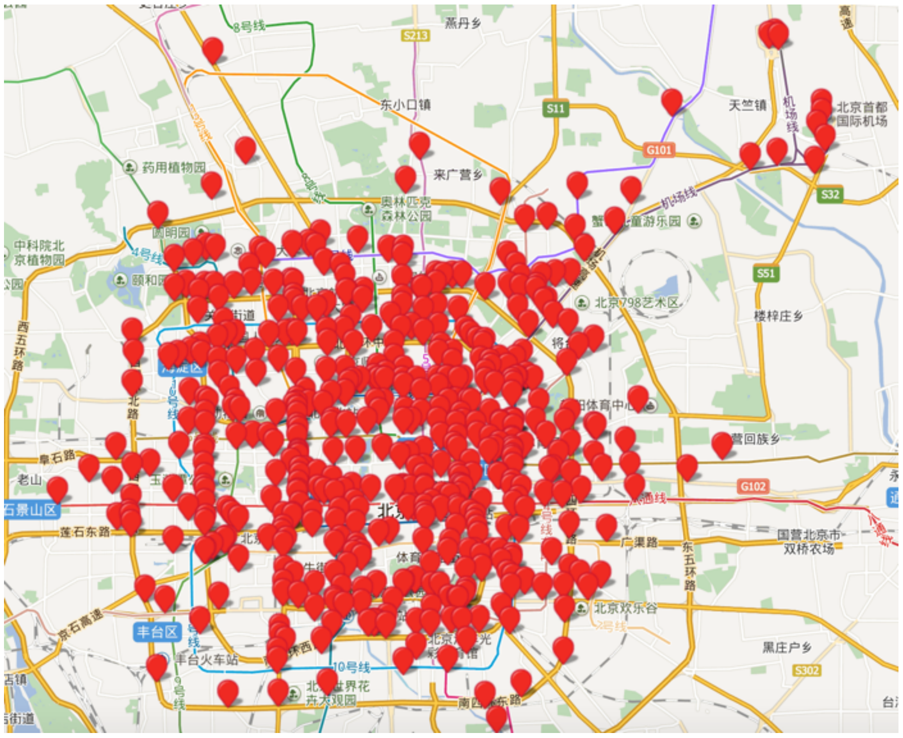

Figure 3-1: A plot of landmarks when $k = 500$

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[11]. Sometimes, a taxi may traverse several landmarks but only the first and the last landmarks are recorded, due to the low sampling rate, which cause the duration 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 A.1.

Listing 3.3: Pseudocode for constructing landmark graph

```
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) && !is_landmark(streets[j]):
                j = j + 1

        intermediaries = []
        //find another landmark
        k = j + 1
        while k < len(streets) && !is_landmark(streets[k]):
                intermediaries.append(streets[k])
                k = k + 1

        //insert edge (streets[j], intermediaries, streets[k])
        insert(E, streets[j], intermediaries, streets[k])

        //next search starts from the second landmark
        j = k
```

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Travel Time Estimation

Chapter 3 describes the general procedures for constructing a landmark graph. This chapter deals with how to estimate the travel time of each edge of a landmark graph at a particular moment in time.

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 road has one *peak* at 8 a.m. when people go to work and the other peak at 6 p.m. when people go 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 weekend shopping with families and the other peak at only 8 p.m. when the whole day's celebrations are over.

Based on this observation, 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 **??**.

Definition 9 (*Landmark*)**.** A landmark is a road segment that is frequently traversed by taxi drivers according to the taxi GPS trajectory database. [11]

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 determining the exact route on which a taxi traversed difficult. Therefore, it is not possible to estimate the time cost for each road segment. However, ignoring the length of a road segment and considering it as an abstract "landmark" make estimating the travel time between two *landmarks* viable.

Moreover, the notion of landmarks closely follows the way how drivers remember the driving routes in daily life[11]. 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 a landmark graph in order to estimate the time cost between two landmarks. But before that, the taxi GPS trajectories must be separated into a set of *trips.*

4.1 Trip Identification

Definition 10 (*Trip*)**.** A trip is a taxi trajectory that satisfies either condition:

1. A passenger is on the taxi, namely, all records in the set have an OCCUPIED field of value 1 or,

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

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 4.1: An example of trip identification

Table 4.1 gives a tiny example of trip identification.

Clearly, all records in Table 4.1 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 larger than 3 minute. The next five records constitute another trip, even though the last two records have a time difference 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 4.1 gives the pseudocode for trip identification.

Listing 4.1: Pseudocode for trip identification

```
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) && \
        (curr_unixepoch - last_unixepoch > threshold):
                ++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 10 is more of a heuristic rather than a theorem. It is meant to provide sufficient data for subsequent machine learning tasks with reasonable accuracy.

4.2 Landmark Frequency

Definition 10 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

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 4.2: An illustration of frequency counting

drivers frequently enough is it recognised as a landmark. In this project, the notion of frequency is given by Definition 11.

Definition 11 (*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 4.2 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 4.2.

Listing 4.2: Pseudocode for counting road frequency

```
frequencies = dict{}
for trip in trips:
        // get unique streets
        streets = unique(trip.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 4-1.

4.3 Landmark Graph

Definition 12 (*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}.

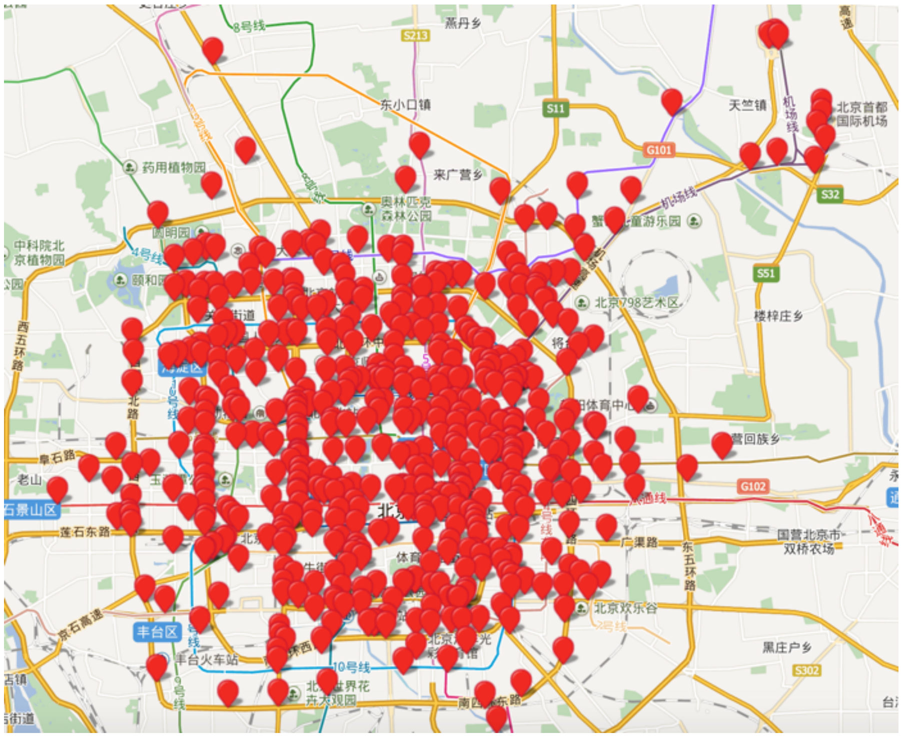

Figure 4-1: A plot of landmarks when $k = 500$

The threshold t_{max} is used to eliminate trajectories with unreasonably long durations[11]. Sometimes, a taxi may traverse several landmarks but only the first and the last landmarks are recorded, due to the low sampling rate, which cause the duration 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 A.1.

Listing 4.3: Pseudocode for constructing landmark graph

```
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) && !is_landmark(streets[j]):
                j = j + 1

        intermediaries = []
        //find another landmark
        k = j + 1
        while k < len(streets) && !is_landmark(streets[k]):
                intermediaries.append(streets[k])
                k = k + 1

        //insert edge (streets[j], intermediaries, streets[k])
        insert(E, streets[j], intermediaries, streets[k])

        //next search starts from the second landmark
        j = k
```

Appendix A

Source Code

Listing A.1: Pseudocode for constructing landmark graph

```
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) && !is_landmark(streets[j]):
                    j = j + 1

            intermediaries = []
            //find another landmark
            k = j + 1
            while k < len(streets) && !is_landmark(streets[k]):
                    intermediaries.append(streets[k])
                    k = k + 1

```

```
        //insert edge (streets[j], intermediaries, streets[k])
        insert(E, streets[j], intermediaries, streets[k])

        //next search starts from the second landmark
        j = k
```

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

[4] 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.

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

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

[7] 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.

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

[9] ———, *Restrictions on geographic data in china*, March 2017.

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

[11] 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.

[12] 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.