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 4.1.

bjtaxigps_30m		bjtaxigps_50m	
wrkd_ldmkgraph_30m	holi_ldmkgraph_30m	wrkd_ldmkgraph_50m	holi_ldmkgraph_50m

Table 4.1: An summary of landmark graphs

4.1 Travel Time Distribution

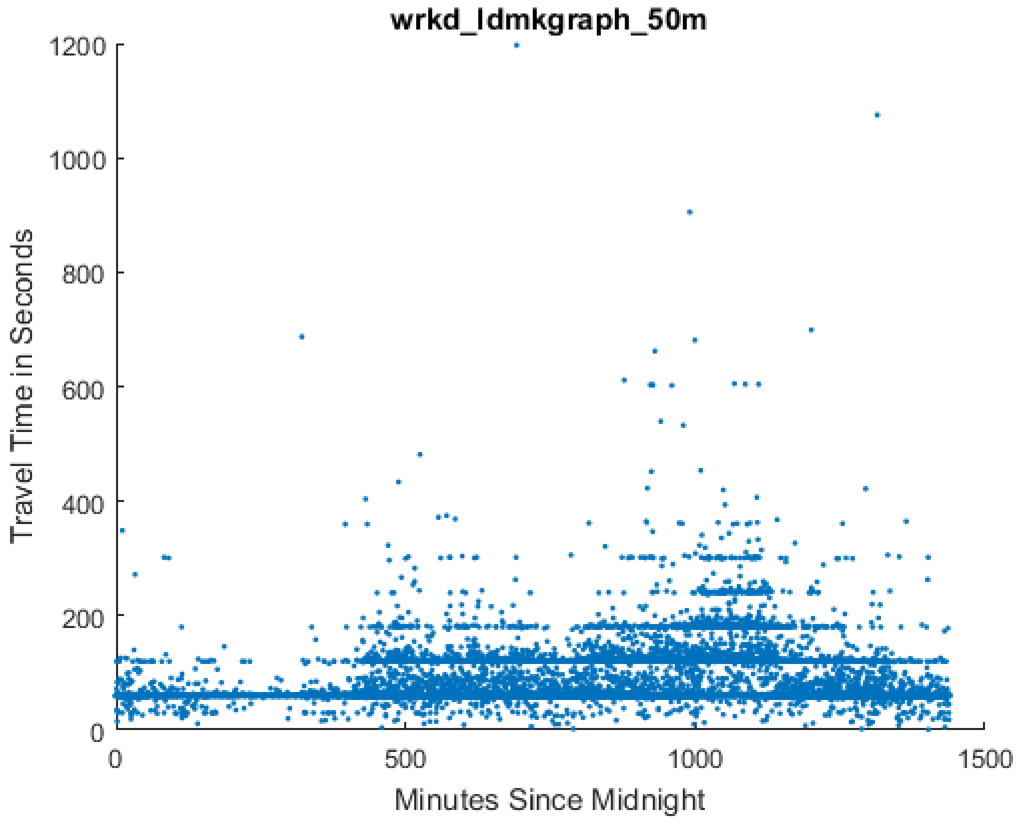

Figure 4-1: An example of travel time patterns

Figure 4-1 illustrates a scatter plot of the travel time of a particular landmark graph edge during the course of a weekday. 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 choose different routes to travel between the two landmarks. Different routes have different traffic conditions, 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, 120 seconds and 180 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.

Therefore, it is not possible to fit the scatter points with a single-valued function. Rather, the clustering technique should be used to identify travel time clusters. Like in Section 2.3.2, a **self-organising feature map** is used to cluster the travel time. Figure 4-2 shows the final positions of the neurons at the end of the training.

One of the merits of clustering is *dimension reduction* or *feature extraction*. In this case, after the training is completed, the SOFM has learned some features about the travel time and expressed those features by moving its neurons to the centroids of the clusters. Now, **the data points can be represented by their respective centroids**, thus the dimension is reduced. Figure 4-3 shows the effect of replacing each data point's value with their centroids'.

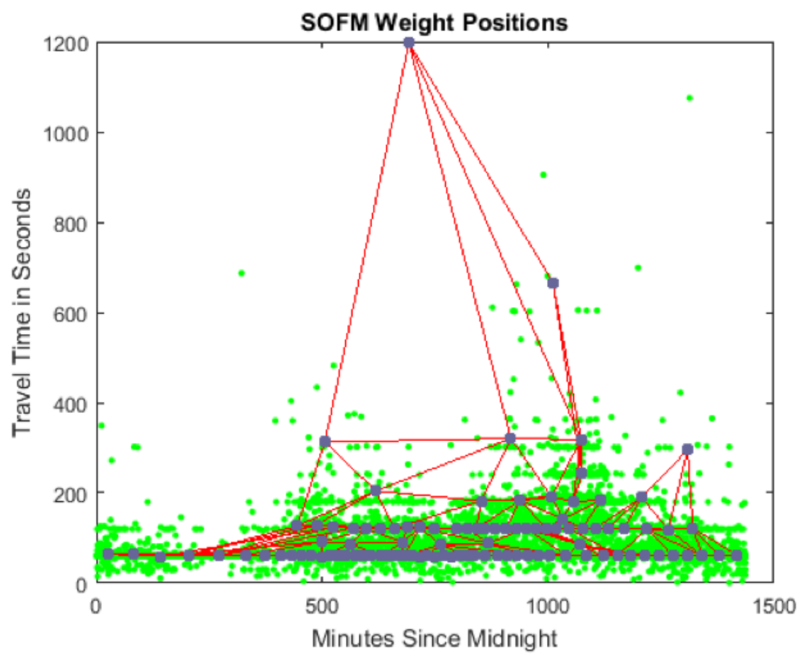

Figure 4-2: An example of travel time patterns

4.2 Trip Identification

Definition 9 (*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 3.1 gives a tiny 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 larger than 3 minute. The next five records constitute another trip, even

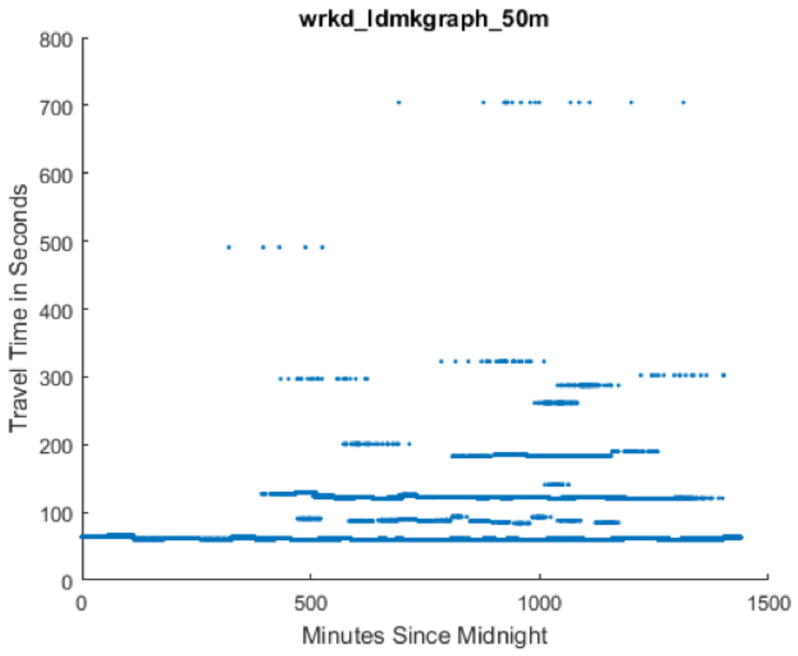

Figure 4-3: An example of travel time patterns

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 9 is more of a heuristic rather than a theorem. It is meant to provide sufficient data for subsequent machine learning tasks with reasonable accuracy.

4.3 Landmark Frequency

Definition 9 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 10.

Definition 10 (*Frequency of a Road Segment*)**.** The frequency of a road segment is the sum of *unique* occurrences of that road segment in each trip.

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

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 4-4.

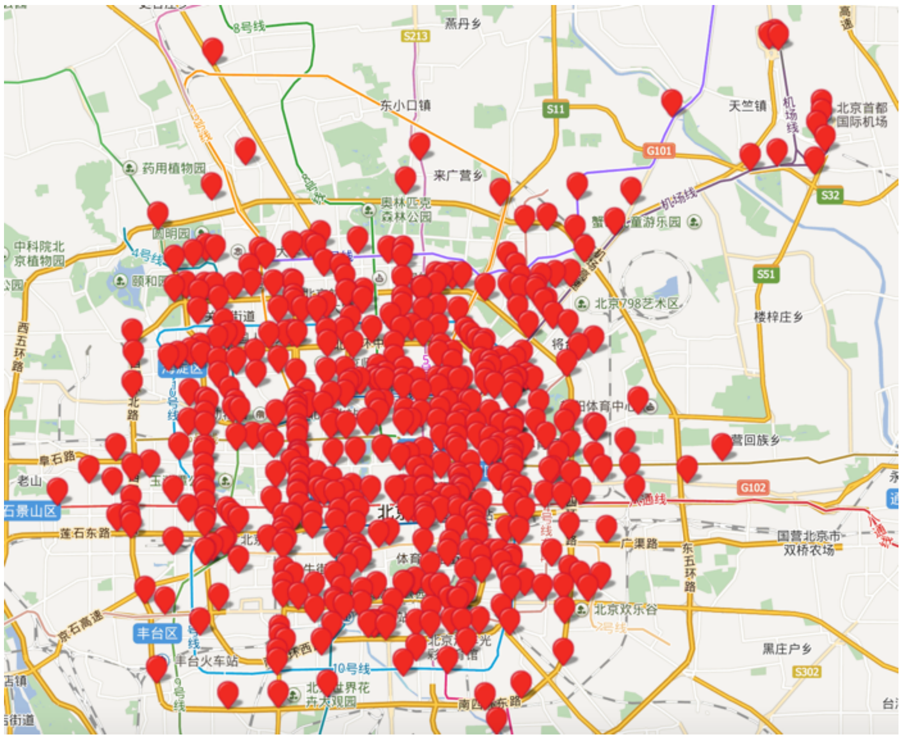

Figure 4-4: A plot of landmarks when $k = 500$

4.4 Landmark Graph

Definition 11 (*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 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
```