# 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

```
1  Input: a set of GPS records and a time span threshold t_max
2  Output: a set of GPS records with each record assigned a trip ID
3
4  curr_tripid = 1
5  last_occup = curr_occup = records[0].OCCUPIED
6  last_unixepoch = curr_unixepoch = records[0].UNIX_EPOCH
7  for recd in records:
8      curr_occup = recd.OCCUPIED
9      curr_unixepoch = recd.UNIX_EPOCH
10     if curr_occup != last_occup:
11         ++curr_tripid
12     elif (curr_occup == 0) && (curr_unixepoch - last_unixepoch > t_max):
13         ++curr_tripid
14     recd.TRIP_ID = curr_tripid
15     last_occup = curr_occup
16     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.

Listing 3.2: Pseudocode for counting road frequency

```
1 frequencies = dict{}
2 for trip in trips:
3         // get unique streets
4         streets = unique(trip.streets)
5         for street in streets:
6                 if street in frequencies:
7                         ++frequencies[street]
8                 else:
9                         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.

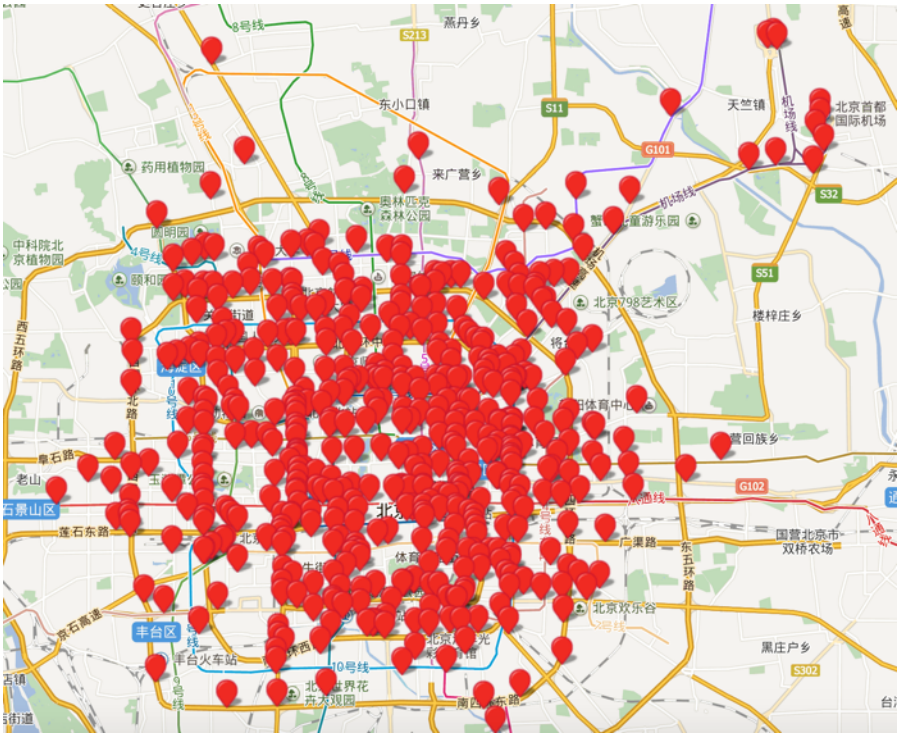


Figure 3-1: A plot of landmarks when $k = 500$

## 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 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 3.3.

Listing 3.3: Pseudocode for constructing landmark graph

```
1  for trip in trips:
2     //get unique, chronologically ordered streets
3     streets = unique_ordered(trip.streets)
4     while j < len(streets):
5          //loop until a landmark is found
6          while j < len(streets) && !is_landmark(streets[j]):
7                  j = j + 1
8
9          intermediaries = []
10         //find another landmark
11         k = j + 1
12         while k < len(streets) && !is_landmark(streets[k]):
13                 intermediaries.append(streets[k])
14                 k = k + 1
15
16         //insert edge (streets[j], intermediaries, streets[k])
17         insert(E, streets[j], intermediaries, streets[k])
18
19         //next search starts from the second landmark
20         j = k
```