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:

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

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 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

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

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) and (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

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

```
1  Input: a set of trips
2  Output: a landmark graph G = (V, E)
3  for trip in trips:
4     //get unique, chronologically ordered streets
5     streets = unique_ordered(trip.streets)
6     while j < len(streets):
7           //loop until a landmark is found
8           while j < len(streets) and not is_landmark(streets[j]):
9               j = j + 1
10
11          //find another landmark
12          intermediaries = []
13          k = j + 1
14          while k < len(streets) and not is_landmark(streets[k]):
15              intermediaries.append(streets[k])
16              k = k + 1
17          //insert edge
18          E.insert(streets[j], intermediaries, streets[k])
19          //next search starts from the second landmark
20          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 $w : E, t \rightarrow \mathbb{R}$. 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

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.

Chapter 5

Time-Dependent Shortest Path Calculation

Definition 12 (*FIFO Graph*)**.** A time-dependent graph $G = (V, E)$ with a dynamic weight function $w : E, t \rightarrow \mathbb{R}$ 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.

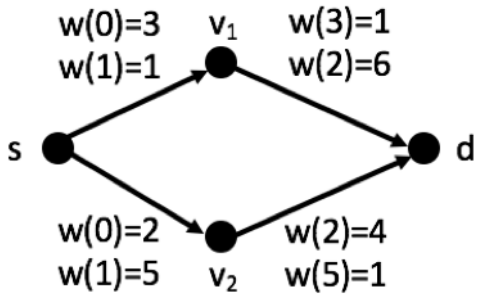

Figure 5-1: An example of updating edge costs dynamically

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

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

```
7  pq.put(s)
8  for vertex in G.V − {s}:
9      vertex.EAT = ∞
10     pq.put(vertex)
11 while not pq.empty():
12     hd = pq.get() // get the vertex at the queue head
13     for v in hd.neighbours:
14         if hd.EAT + w(hd, v, hd.EAT) < v.EAT:
15             v.EAT = hd.EAT + w(hd, v, hd.EAT)
16             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

```
1  Input: a predecessor graph G_p = (V_p, E_p) and a destination d
2  Ouput: a shortest path s ⇝ d
3
4  def print_path(predecessor, dst):
5      pred = predecessor[dst]
6      if pred is None:
7          print(dst.name)
8      else:
9          print_path(predecessor, pred)
10         print('→' + dst.name)
11
12 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. With more hand-hold GPS devices embedded in mobile phones available, it is possible in the future to incorporate personal driving trajectories into the travel time estimation, so that more personalised, accurate driving route suggestions can be provided.

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

```
1  <?php
2  include "../inc/jingodbinfo.inc";
3
4  function connect_db(){
5      $conn = new mysqli(DB_SERVER, DB_USERNAME, DB_PASSWORD);
6      if($conn->connect_error){
7          die("Connection failed: " . $conn->connect_error);
8      }
9      $conn->select_db(DB_DATABASE);
10     $conn->set_charset("utf8");
11     return $conn;
12 }
13
```

```
14 function disconnect_db($conn){ $conn->close(); }
15
16 function db_select($conn, $table, $cols, $cond = "true", $distinct = "")
17 {
18     $sql_select = "SELECT {$distinct} ";
19     if(count($cols) == 0){
20         $sql_select .= "*,";
21     }
22     foreach ($cols as $col) {
23         $sql_select .= "{$col},";
24     }
25     $sql_select = rtrim($sql_select, ",");
26     $sql_select .= " FROM {$table} WHERE {$cond};";
27     $ret = $conn->query($sql_select);
28     $res = array();
29     if($ret->num_rows > 0){
30         while($row = $ret->fetch_assoc()){
31             $curr = array();
32             foreach ($cols as $col) {
33                 $curr[$col] = $row[$col];
34             }
35             $res[] = $curr;
36         }
37     }
38     return $res;
39 }
40
41 function db_update($conn, $table, $values, $cond){
42     $sql_update = "UPDATE {$table} SET ";
43     foreach ($values as $key => $value) {
44         if(is_numeric($value)){
45             $sql_update .= "{$key} = {$value},";
```

```
46          }else{
47              $sql_update .= "{$key} = '{$value}',";
48          }
49      }
50      $sql_update = rtrim($sql_update, ",");
51      $sql_update .= " WHERE {$cond};";
52      $succ = $conn->query($sql_update);
53      return $succ;
54  }
55
56  function db_insert($conn, $table, $values, $cond = "", $ignore = ""){
57      $sql_insert = "INSERT {$ignore} INTO {$table} ";
58      $cols = "";
59      $vals = "";
60      foreach ($values as $key => $value) {
61          $cols .= "{$key},";
62          $vals .= (is_numeric($value) ? "{$value}," : "'{$value}',");
63      }
64      $cols = rtrim($cols, ",");
65      $vals = rtrim($vals, ",");
66      $sql_insert .= "({$cols}) VALUES ({$vals}) {$cond};";
67      echo "{$sql_insert}\n";
68      $conn->query($sql_insert);
69  }
70
71  function db_delete($conn, $table, $cond){
72      $sql_delete = "DELETE FROM {$table} WHERE {$cond};";
73  }
74  ?>
```

A.2 Data Pre-processing

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

Listing A.2: Outlier Removal

```
1  <?php
2  include "db_utilities.php";
3
4  class Filter{
5      private $ldmk_start;
6      private $ldmk_end;
7      private $ldmk_table;
8      private $data_table;
9      private $update_table;
10     private $path;
11     private $bchmk;
12
13     private $conn;
14     private $num_dele;
15
16     public function __construct($ldmk_start, $ldmk_end, $ldmk_table,
           $data_table, $path, $bchmk, $update_table){
17         $this->ldmk_start = $ldmk_start;
18         $this->ldmk_end = $ldmk_end;
19         $this->ldmk_table = $ldmk_table;
20         $this->data_table = $data_table;
21         $this->update_table = $update_table;
22         $this->path = $path;
23         $this->bchmk = $bchmk;
24         $this->conn = connect_db();
25         $this->num_dele = 0;
26     }
```

```
27
28      public function __destruct(){ disconnect_db($this->conn); }
29
30      private function getCentres($ldmk){
31          $filename = "{$this->path}{$ldmk}.csv";
32          $centres = array();
33          if(($handle = fopen($filename, 'r')) !== FALSE){
34              while(($line = fgetcsv($handle, 0, ',')) !== FALSE){
35                  $centres[] = array($line[0], $line[1]);
36              }
37          }
38          return $centres;
39      }
40
41      private function getRecords($ldmk){
42          $ldmk_cols = array('LandmarkName');
43          $ldmk_cond = "LandmarkID = {$ldmk}";
44          $ret = db_select($this->conn, $this->ldmk_table, $ldmk_cols,
                $ldmk_cond);
45          $data_cols = array('DataUnitID', 'BD09_LONG', 'BD09_LAT');
46          $data_cond = "Street = '{$ret[0][$ldmk_cols[0]]}'";
47          $ret = db_select($this->conn, $this->data_table, $data_cols,
                $data_cond);
48          $records = array();
49          foreach ($ret as $item) {
50              $records[] = array($item[$data_cols[0]], $item[$data_cols
                    [1]], $item[$data_cols[2]]);
51          }
52          return $records;
53      }
54
55
```

```
56      private function removeRecord($centres, $records){
57          foreach ($records as $recd) {
58              $min_dist = INF;
59              foreach ($centres as $centre) {
60                  $min_dist = min($min_dist, $this->getHaversineDist
61                  ($centre[0], $centre[1], $recd[1], $recd[2]));
62              }
63              if($min_dist > $this->bchmk){
64                  $cond = "DataUnitID = {$recd[0]}";
65                  db_delete($this->conn, $this->data_table, $cond);
66                  ++$this->num_dele;
67              }
68          }
69      }
70
71      private function toRadian($degree){ return $degree * M_PI / 180; }
72
73      private function getHaversineDist($long1, $lat1, $long2, $lat2){
74          $BJ_LAT = $this->toRadian(39);
75          $EQ_R = 6378137; // equatorial radius in metres
76          $POL_R = 6356752; // polar radius in metres
77
78          $BJ_R = sqrt((pow($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)));
79          $long1 = $this->toRadian($long1);
80          $lat1 = $this->toRadian($lat1);
81          $long2 = $this->toRadian($long2);
82          $lat2 = $this->toRadian($lat2);
83          $d = 2 * $BJ_R * asin(sqrt(pow(sin(($lat1 - $lat2) / 2), 2) +
84          cos($lat1) * cos($lat2) * pow(sin(($long1 - $long2) / 2), 2)));
85          return $d; }
```

```
86      private function getWithin($ldmk, $centres, $records){
87          foreach ($records as $recd) {
88              $min_dist = INF;
89              foreach ($centres as $centre) {
90                  $min_dist = min($min_dist,
91                  $this->getHaversineDist($centre[0], $centre[1], $recd
                        [1], $recd[2]));
92              }
93              if($min_dist <= $this->bchmk){
94                  db_insert($this->conn, $this->update_table,
95                          array('LandmarkID' => $ldmk, 'BD09_LONG' =>
                              $recd[1], 'BD09_LAT' => $recd[2]));
96              }
97          }
98      }
99
100     public function doFiltering(){
101         for($ldmk = $this->ldmk_start; $ldmk != $this->ldmk_end;
102                 ++$ldmk){
103             $centres = $this->getCentres($ldmk);
104             $records = $this->getRecords($ldmk);
105             $this->getWithin($ldmk, $centres, $records);
106         }
107     }
108 }
109 set_time_limit(0);
110 ini_set('memory_limit','2048M');
111 $filter = new Filter($_POST['ldmk_start'], $_POST['ldmk_end'],
112         $_POST['ldmk_table'], $_POST['data_table'], $_POST['path'],
113         $_POST['bchmk'], $_POST['update_table']);
114 $filter->doFiltering();
115 ?>
```

A.3 Landmark Graph Construction

This section presents algorithms related to landmark graph construction.

Listing A.3: Trip Identification

```
1  <?php
2  include "db_utilities.php";
3
4  class IdentifyTrip{
5      private $cuid_start;
6      private $cuid_end;
7      private $table;
8      private $tripid;
9      private $threshold;
10     private $occup;
11     private $conn;
12
13     public function __construct($cuid_start, $cuid_end, $table, $tripid,
           $threshold, $occup){
14         $this->cuid_start = $cuid_start;
15         $this->cuid_end = $cuid_end;
16         $this->table = $table;
17         $this->tripid = $tripid;
18         $this->threshold = $threshold;
19         $this->occup = $occup;
20         $this->conn = connect_db();
21     }
22
23     public function __destruct(){ disconnect_db($this->conn); }
24
25
26
```

```
27      public function startIdentifyTrip(){
28          $cols = array('DataUnitID', 'UnixEpoch');
29          for($cuid = $this->cuid_start; $cuid != $this->cuid_end;
30                  ++$cuid){
31              $cond = "CUID = {$cuid} and Occupied = {$this->occup}";
32              $res = db_select($this->conn, $this->table, $cols, $cond);
33              if(count($res) > 0){
34                  $this->splitTrip($res, $cols);
35              }
36          }
37      }
38
39      public function splitTrip($res, $cols){
40          $last = $curr = $res[0][$cols[1]];
41          foreach ($res as $item) {
42              $curr = $item[$cols[1]];
43              if($curr - $last > $this->threshold){
44                  ++$this->tripid;
45              }
46              $values = array('TripID' => $this->tripid);
47              $cond = "{$cols[0]} = {$item[$cols[0]]}";
48              $succ = db_update($this->conn, $this->table, $values,
49                  $cond);
50              $last = $curr;
51          }
52          ++$this->tripid;
53      }
54  }
55  set_time_limit(0); ini_set('memory_limit','2048M');
56  $identifyTrip = new IdentifyTrip($_POST['start'], $_POST['end'], $_POST[
        'table'], $_POST['tripid'], $_POST['threshold'], $_POST['occup']);
57  $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

```
1  <?php
2  include "db_utilities.php";
3
4  class GraphBuilder{
5      private $start;
6      private $end;
7      private $ldmktable;
8      private $triptable;
9      private $ldmklimit;
10     private $holi_table;
11     private $wrkd_table;
12     private $conn;
13     private $landmarks;
14
15     public function __construct($start, $end, $ldmktable, $triptable,
           $ldmklimit, $holi_table, $wrkd_table){
16         $this->start = $start;
17         $this->end = $end;
18         $this->ldmktable = $ldmktable;
19         $this->triptable = $triptable;
20         $this->ldmklimit = $ldmklimit;
21         $this->holi_table = $holi_table;
22         $this->wrkd_table = $wrkd_table;
23         $this->conn = connect_db();
24         $this->landmarks = $this->fetchldmk();
25     }
26
27     public function __destruct(){ disconnect_db($this->conn); }
28
29
30
```

```
31      private function fetchldmk(){
32          $cols = array('LandmarkName', 'LandmarkID');
33          $cond = "{$cols[1]} <= {$this->ldmklimit}";
34          $res = db_select($this->conn, $this->ldmktable, $cols, $cond);
35          $ldmks = array();
36          foreach ($res as $item) {
37              $ldmks[$item[$cols[0]]] = $item[$cols[1]];
38          }
39          return $ldmks;
40      }
41
42      private function isLandmark($street){
43          return array_key_exists($street, $this->landmarks);
44      }
45
46      private function isHoliday($atime){
47          $date = date('d', $atime);
48          $day = date('w', $atime);
49          if($date == '01' || $date == '28' || $date == '29'){
50              return true;
51          }else if($date == '31'){
52              return false;
53          }else if($day == 0 || $day == 6){
54              return true;
55          }
56          return false;
57      }
58
59
60
61
62
```

```
63      public function buildGraph(){
64          $street_cols = array("Street");
65          $utc_cols = array("UnixEpoch");
66
67          for($tripid = $this->start; $tripid != $this->end; ++$tripid){
68              $street_utc = array();
69              $street_cond = "TripID = {$tripid}";
70              $streets = db_select($this->conn, $this->triptable,
                    $street_cols, $street_cond, "DISTINCT");
71              foreach ($streets as $item) {
72                  $street = $item[$street_cols[0]];
73                  $utc_cond = "{$street_cols[0]} = '{$street}' AND {
                        $street_cond} LIMIT 1";
74                  $ret = db_select($this->conn, $this->triptable,
                        $utc_cols, $utc_cond);
75                  if(count($ret) > 0){
76                      $street_utc[$street] = $ret[0][$utc_cols[0]];
77                  }
78              }
79              $this->addEdge($street_utc, $tripid);
80           }
81       }
82
83      private function addEdge($street_utc, $tripid){
84          $cols = array("LandmarkU", "Intermediate", "LandmarkV", "
                ArrivalTime", "LeavingTime", "Duration", "TripID");
85          $streets = array_keys($street_utc);
86          $size = count($streets);
87          $low = 0;
88          while($low < $size && !$this->isLandmark($streets[$low])){
89              ++$low;
90          }
```

```
91          $landmarkU = $landmarkV = $low < $size ? $streets[$low] : NULL;
92          $atime = $ltime = $low < $size ? $street_utc[$landmarkV] : NULL;
93          ++$low;
94          while($low < $size){
95              $inbetween = "";
96              while($low < $size && !$this->isLandmark($streets[$low])){
97                  $inbetween .= "{$streets[$low]}-";
98                  ++$low;
99              }
100             $inbetween = rtrim($inbetween, "-");
101             $landmarkV = $low < $size ? $streets[$low] : NULL;
102             $ltime = $low < $size ? $street_utc[$landmarkV] : NULL;
103             if(!is_null($landmarkU) && !is_null($landmarkV)){
104                 $vals = array($landmarkU, $inbetween, $landmarkV, $atime
                        , $ltime, $ltime - $atime, $tripid);
105                 if($this->isHoliday($atime)){
106                     db_insert($this->conn, $this->holi_table,
                            array_combine($cols, $vals));
107                 }else{
108                     db_insert($this->conn, $this->wrkd_table,
                            array_combine($cols, $vals));
109                 }
110             }
111             $landmarkU = $landmarkV;
112             $atime = $ltime;
113             ++$low;
114         }
115     }
116 }
117
118 set_time_limit(0);
119 ini_set('memory_limit','2048M');
```

```
120 date_default_timezone_set("Asia/Singapore");
121
122 $graphBuilder = new GraphBuilder($_POST['tripid_start'], $_POST['
       tripid_end'], $_POST['ldmktable'], $_POST['triptable'], $_POST['
       ldmklimit'], $_POST['holi_table'], $_POST['wrkd_table']);
123
124 $graphBuilder->buildGraph();
125 ?>
```

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.