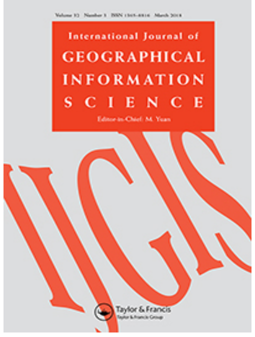


Taylor & Francis
Taylor & Francis Group

International Journal of Geographical Information Science

ISSN: 1365-8816 (Print) 1362-3087 (Online) Journal homepage: www.tandfonline.com/journals/tgis20

Fast map matching, an algorithm integrating hidden Markov model with precomputation

Can Yang & Győző Gidófalvi

To cite this article: Can Yang & Győző Gidófalvi (2018) Fast map matching, an algorithm integrating hidden Markov model with precomputation, International Journal of Geographical Information Science, 32:3, 547-570, DOI: 10.1080/13658816.2017.1400548

To link to this article: https://doi.org/10.1080/13658816.2017.1400548

Published online: 13 Nov 2017.

Submit your article to this journal

Article views: 3297

View related articles

View Crossmark data

Citing articles: 107 View citing articles

INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SCIENCE, 2018
VOL. 32, NO. 3, 547–570
https://doi.org/10.1080/13658816.2017.1400548

Taylor & Francis
Taylor & Francis Group

ORIGINAL ARTICLE

Check for updates

Fast map matching, an algorithm integrating hidden Markov model with precomputation

Can Yang and Győző Gidófalvi

Division of Geoinformatics, Department of Urban Planning and Environment, KTH, Royal Institute of Technology in Sweden, Stockholm, Sweden

ABSTRACT

Wide deployment of global positioning system (GPS) sensors has generated a large amount of data with numerous applications in transportation research. Due to the observation error, a map matching (MM) process is commonly performed to infer a path on a road network from a noisy GPS trajectory. The increasing data volume calls for the design of efficient and scalable MM algorithms. This article presents fast map matching (FMM), an algorithm integrating hidden Markov model with precomputation, and provides an open-source implementation. An upper bounded origin-destination table is precomputed to store all pairs of shortest paths within a certain length in the road network. As a benefit, repeated routing queries known as the bottleneck of MM are replaced with hash table search. Additionally, several degenerate cases and a problem of reverse movement are identified and addressed in FMM. Experiments on a large collection of real-world taxi trip trajectories demonstrate that FMM has achieved a considerable single-processor MM speed of 25,000–45,000 points/second varying with the output mode. Investigation on the running time of different steps in FMM reveals that after precomputation is employed, the new bottleneck is located in candidate search, and more specifically, the projection of a GPS point to the polyline of a road edge. Reverse movement in the result is also effectively reduced by applying a penalty.

ARTICLE HISTORY
Received 21 April 2017
Accepted 31 October 2017

KEYWORDS
Map matching;
precomputation;
performance improvement

1. Introduction

Benefiting from the rapid development of wireless communication and positioning technologies, global positioning system (GPS) sensors have been widely deployed on vehicles, smart phones, and bicycles to collect movement information. Compared with traditional stationary sensors used in transportation research such as loop detector mounted at fixed locations in a road network to monitor velocity and traffic volume, GPS sensor is recognized as a promising data collection approach which is cost-effective with a broad spatial coverage (Li *et al*. 2014). It typically reports ID, latitude, longitude of the tracked object, and timestamp of record while some advanced sensors also report velocity, heading, and acceleration (White *et al*. 2000). Due to the uncertainties caused by inevitable measurement error and low sampling

CONTACT Can Yang cyang@kth.se

rate (Li *et al*. 2013), continuous movement on a road network is observed as a sequence of periodically reported GPS observations with an offset from the true locations. In order to infer the path traversed by the object, a *map matching* (MM) process is frequently performed, which is an important preprocessing step in many GPS data relevant studies such as route navigation (Yuan *et al*. 2010a), travel time estimation (Rahmani *et al*. 2015), traffic modeling (Castro *et al*. 2012), and activity recognition (Liao *et al*. 2007).

In the literature, many algorithms have been proposed to solve the MM problem. In Quddus *et al*. (2007), early MM algorithms are classified into four categories including geometric, topological, probabilistic, and other advanced. Most of them are sensitive to GPS error, sampling rate, and initial MM results (Newson and Krumm 2009, Zeng *et al*. 2015). Since hidden Markov model (HMM) was first employed to solve MM problem in Newson and Krumm (2009) and Lou *et al*. (2009), it has been widely adopted in later studies because of the elegant and extensible integration of GPS error, geometry, topology, and other factors in the path inference. Along that direction, most of the efforts are devoted to improving the accuracy of MM by introducing new information such as speed limit (Yuan *et al*. 2010b), turning angle (Li *et al*. 2013), and curvedness (Zeng *et al*. 2015) or designing more robust and realistic objective functions for path inference (Yuan *et al*. 2010b, Wei *et al*. 2012, Chen *et al*. 2014). However, performance improvement in MM is not fully exploited in those studies (Marchal *et al*. 2005, Huang *et al*. 2013).

Considering the increasing size of GPS data, it is necessary to develop MM algorithms capable of handling a large amount of data in a short time. Concentrated primarily on performance improvement, this article proposes *Fast Map Matching* (FMM), an algorithm integrating HMM with precomputation. The algorithm contains two stages where in the first precomputing stage, an upper bounded origin-destination table is created from the road network to store information about all pairs of shortest paths whose lengths are under a certain threshold. In the second stage, MM is performed where computationally expensive routing queries are replaced with hash table search. Contributions of this article are highlighted below:

(1) An efficient and robust MM algorithm is developed with open-source implementation provided. Several degenerate cases in MM are discovered and settled in this article. A problem of reverse movement is identified in previous algorithms and addressed by applying a penalty.
(2) Various aspects of the algorithm are evaluated with a large GPS trajectory dataset. The results demonstrate that FMM has achieved a considerable single processor MM speed of 25,000–45,000 points/second varying with the output mode. Investigation on the running time of the steps in FMM reveals that after precomputation is employed, the bottleneck has shifted from previously known repeated routing queries to candidate search (CS), and more specifically, the projection of a GPS point to the polyline of a road edge. Reverse movement in the result is effectively reduced by applying a penalty.

The remainder of this article is organized as follows. Section 2 provides a systematic literature review, and Section 3 gives the preliminaries. The algorithm is presented in

Section 4. In Section 5, a case study on a real-world GPS dataset is reported. Finally, conclusions are drawn and future work is planned in Section 6.

2. Related work

In the literature, several approaches have been proposed to improve the performance of MM, which can be classified into four categories: spatial indexing, trajectory simplification, parallel computing, and advanced routing algorithms.

Spatial indexing technologies such as R-tree guttman 1984 and Quad-tree (Guttman 1984) and Quad-tree (Finkel and Bentley 1974) have been widely employed in MM algorithms to accelerate the search of neighboring edges around GPS point (Marchal *et al*. 2005, Li *et al*. 2011, Wei *et al*. 2012, Chen *et al*. 2014). However, the contribution of spatial indexing is limited to candidate search, and it is commonly used in collaboration with other methods.

Trajectory simplification approaches improve the performance by reducing the number of points to be matched, which are effective especially for dense and duplicated observations. In Li *et al*. (2014), three simplification algorithms incremental, slicing, and global window are proposed by assigning weights to GPS points based on the geometric information of the trajectory and filtering those with small weights. In the authors' recent work (Li *et al*. 2015), an oriented bounding rectangle algorithm is proposed to cluster points in a trajectory based on moving direction, which is also capable of handling outliers. However, these algorithms are restricted to dense GPS data.

Parallel computing is also a promising technology for high-performance MM because not only matching individual trajectory but also searching candidate edges for each GPS point within a trajectory can be parallelized. With the rapid development of distributed and parallel computing technologies, considerable improvement has been reported in previous studies. In Li *et al*. (2011), an online MM framework is designed and tested on a 16 nodes cluster with an overall performance of around 20,000 points/second. Similarly, in Huang *et al*. (2013), the highest speed is reported as 120,000 points/second on a 20 nodes cluster. However, the potential improvement of single processor's performance is not fully exploited in those studies.

Compared with above three categories, advanced routing algorithms attract more attention because the bottleneck of HMM-based MM algorithm has been widely recognized as repeated routing queries (Lou *et al*. 2009, Wei *et al*. 2012, Rahmani and Koutsopoulos 2013, Zhe and Zhang 2015). Apart from classical Dijkstra algorithm (Dijkstra 1959), various advanced routing algorithms have been proposed for accelerating shortest path queries such as A* (Hart *et al*. 1968) and ALT (Goldberg and Harrelson 2005). However, their contribution to MM is limited because they primarily focus on the query of a large graph where origin and destination are distant while in MM context, the distance between consecutive GPS observations is likely to be small. In Rahmani and Koutsopoulos (2013), traversing the whole network graph is avoided by constructing a local search tree according to the distance estimated from the time interval between GPS observations. A more significant improvement is achieved in Chen *et al*. (2014) where the number of repeated queries is reduced by adding dummy nodes and links connecting GPS point with neighboring edges in the network graph. Consequently, multiple sources to multiple targets routing queries are converted to a single query passing a sequence of dummy nodes. However, it frequently updates the network graph for each trajectory, which degrades the performance. Another approach to address the problem is to precompute some routing results. In Zhe and Zhang (2015), an upper bounded origin and destination table is created from the road network and queried to obtain the shortest path distance between candidates. FMM developed in this article improves the performance based on the same principle but differs from their work in several aspects. First, the bottleneck of repeated routing queries is fully addressed with a much higher performance achieved. Second, various aspects of integrating precomputation into MM are reported including sensitivities to configurations, running time constitution, and handling of long-distance routing queries. Third, several degenerate cases and a problem of reverse movement are identified and addressed.

3. Preliminaries

A road network is represented as a directed graph denoted by $G = (V, E)$ where V is the set of nodes and E is the set of edges. Each node in V is stored as a point representing intersections and dead ends of road segments. Each edge $e \in E$ represents a directed road segment with ID of *eid* starting from a source node with ID of s to a target node with ID of t. The geometry of e is a polyline denoted by *geom* with a length of L. Therefore, edge e is stored as a tuple denoted by

$$e = (eid, s, t, geom, L) \tag{1}$$

Movement on e is constrained to the direction from node s to t. For brevity, edge e is also written as (s, t) in subsequent content.

Based on the format of GPS data collected in this article, a *GPS observation p* records the latitude *lat* and longitude *long* of a tracked object at a specific time-stamp *ts*, denoted as a tuple $p = (lat, long, ts)$, which is also referred as a *GPS point*. A *GPS trajectory tr* is a sequence of temporally ordered GPS points denoted by $tr = \langle p_1, p_2, \ldots, p_N \rangle$, which is stored as a polyline. For simple explanation, the tracked object is assumed to be a vehicle in later content.

4. FMM algorithm

Similar with most previous work, FMM takes a GPS trajectory set TR and a road network G as input. For each trajectory, it exports a path stored as a sequence of edges traversed by the vehicle together with its geometry. Figure 1 displays the whole process of FMM, which consists of two stages precomputing and MM.

4.1. *Precomputing stage*

In the precomputing stage, all pairs of shortest paths (SP) in G whose lengths are under a threshold Δ are calculated and stored in an Upper Bounded Origin Destination Table (UBODT) in a compressive manner. Each row in UBODT corresponds to the SP from an origin node n_o to a destination node n_d and is denoted by

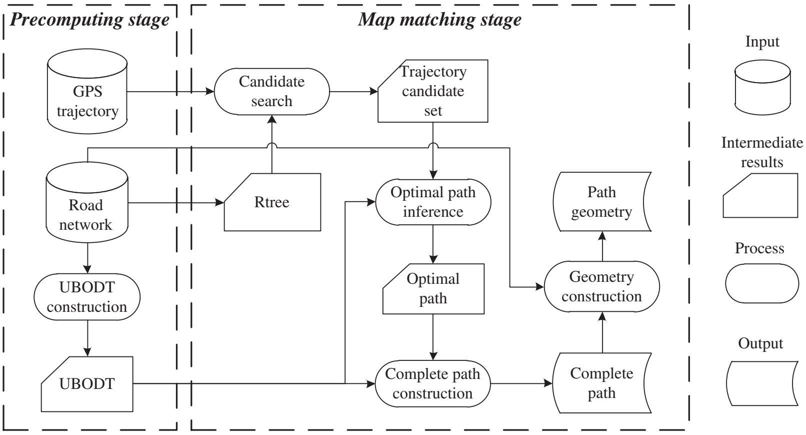


Figure 1. Process of fast map matching algorithm.

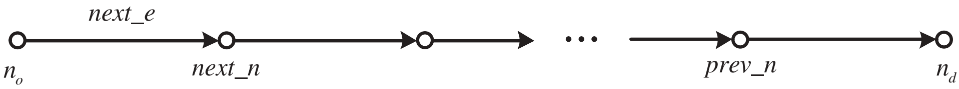


Figure 2. Information stored in a row of UBODT for the shortest path from origin node n_o to destination node n_d.

$$R(n_o, n_d) = (n_o, n_d, next\_n, next\_e, prev\_n, dist) \tag{2}$$

where *next_n* and *next_e* represent the next node and next edge visited after n_o, respectively. The previous node visited before n_d is stored in *prev_n* and *dist* is the SP distance. Figure 2 gives an illustration for these information stored in UBODT.

UBODT is constructed by iterating through every node s in G and calling a classical single-source Dijkstra algorithm starting from s with upper bound of Δ, as shown in Algorithm 1. In each iteration, a shortest path tree is generated, from which the fields in $R(n_o, n_d)$ are derived, as indicated by line 4–10 in Algorithm 1.

After construction, UBODT is stored as a hash table with a composite key of (n_o,n_d), which is implemented as a linear chaining structure (Cormen *et al.* 2001) configured by two parameters: the number of buckets H, which is commonly selected as a prime value, and a multiplier M to compute a unique key as $n_o \times M + n_d$, which can be selected as the number of nodes in G. Given a row $R(n_o, n_d)$, it is inserted into the bucket with index h of

$$h = (n_o \times M + n_d) \bmod H \tag{3}$$

where $0 \le h \le H$. The performance of query in UBODT is closely related to a *load factor* defined as $lf = |UBODT|/H$, which indicates the average number of elements stored in a bucket of the hash table.

From UBODT, SP information can be queried efficiently. The SP distance from node n_o to n_d, denoted by $l^N(n_o, n_d)$, is retrieved directly as follows:

$$l^N(n_o, n_d) = R(n_o, n_d).dist \tag{4}$$

Algorithm 1. UBODT construction algorithm

```
Input:
    G = (V, E): graph of a road network RN
    Δ: Upper bound of distance for all pairs of shortest paths
Output:
    UBODT: a table with columns of (no, nd, next_n, next_e, prev_n, dist)
for s in G.V do
    /* A shortest path tree is returned as a set of nodes visited within Δ
       (NodesVisited), corresponding predecessors (Pred) and distances (Dist) */
    NodesV isited, Pred, Dist = Single_Source_UpperBounded_Dijkstra(s,Δ)
    /* Iterating shortest path tree to get next_n and next_e */
    for u in NodesV isited do
        prev_n = Pred[u]
        /* Find the next node v visited from s to u */
        v = u
        while Pred[v] ≠ s do
            v = Pred[v]
        end
        next_n = v
        next_e = G[s, u]
        write(s, u, next_n, next_e, prev_n, Dist[u])//Write output
    end
end
return
```

Based on the column *next_n* stored in UBODT, the edges in the SP from n_o to n_d, denoted by $Path(n_o, n_d)$, can be efficiently constructed as follows:

$$Path(n_o, n_d) = \langle R(n_o, n_d).next\_e, R(R(n_o, n_d).next\_n, n_d).next\_e, \cdots \rangle \tag{5}$$

which recursively queries the next edge visited from n_o to n_d until the destination is reached.

Although it is enough to construct the path from *next_n*, *prev_n* is also stored in UBODT and it is primarily used for the correction of reverse movement explained later in Section 4.3. In case that the node pair (n_o, n_d) is not found in UBODT, implying that $l^N(n_o, n_d) > \Delta$, a conventional Dijkstra algorithm could be invoked, which is described in detail in Section 4.2.4.

4.2. *MM stage*

In the MM stage, HMM is integrated with precomputation to infer the path traversed by the vehicle considering both GPS error and topology constraint. The stage is divided into four steps: CS, optimal path inference (OPI), complete path construction (CPC), and geometry construction (GC) as illustrated in Figure 1.

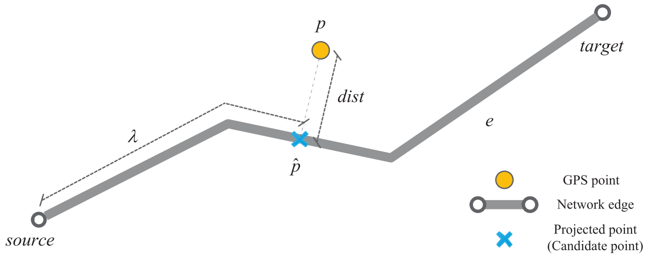


Figure 3. Example of a candidate selected for a GPS point.

4.2.1. *Candidate search*

This step searches for candidate edges for each point in a trajectory. An example of candidate edge is displayed in Figure 3. When e is selected as a *candidate edge* of GPS point p, the location of the vehicle is estimated as the projected point of p on $e.geom$, denoted by $\hat{p}$. The point $\hat{p}$ is also called a *candidate point* of p. A candidate C is stored as a tuple

$$C = (\hat{p}, e, dist, \lambda) \tag{6}$$

where $dist$ is the Euclidean distance from p to $\hat{p}$ and λ is the offset distance along $e.geom$ from $e.s$ to $\hat{p}$ with $0 \leq \lambda \leq e.L$. The projection of a point on a polyline is solved with the linear referencing algorithm (Rahmani and Koutsopoulos 2013) by iterating through each segment in the polyline and returning the one with minimum distance.

Given a GPS point p, the set of candidates is constructed as the k-nearest neighbors (KNN) of road edges around p within a *search radius* of r, denoted by

$$CS(p) = KNN(p)_{k,r} \tag{7}$$

In this article, R-tree (Guttman 1984) is employed to accelerate CS. Although KNN search using R-tree has been studied in Roussopoulos *et al.* (1995), it is not used in FMM because of the additional constraint imposed by r. In practice, it is likely that a small r will be specified and the intersection query will return very few edges, from which KNN can be retrieved efficiently. In FMM, KNN query on R-tree is divided into the following three steps:

(1) R-tree intersection: Create a square with center of p and side length of $2r$ and perform an intersection query on R-tree to export all edges whose bounding box is intersected with the square.
(2) Candidate construction: For each edge e obtained in previous step, project p to $\hat{p}$ on $e.geom$. If $dist(p, \hat{p}) \leq r$, create a candidate for e.
(3) KNN sort: Sort all the candidates found in previous step according to $dist(p, \hat{p})$ and return the k smallest ones.

Based on the above definition, given a trajectory $tr = \langle p_1, p_2, \cdots, p_N \rangle$, a trajectory candidate set (TRCS) is constructed as

$$TRCS(tr) = \langle CS(p_1), CS(p_2), \ldots, CS(p_N) \rangle \tag{8}$$

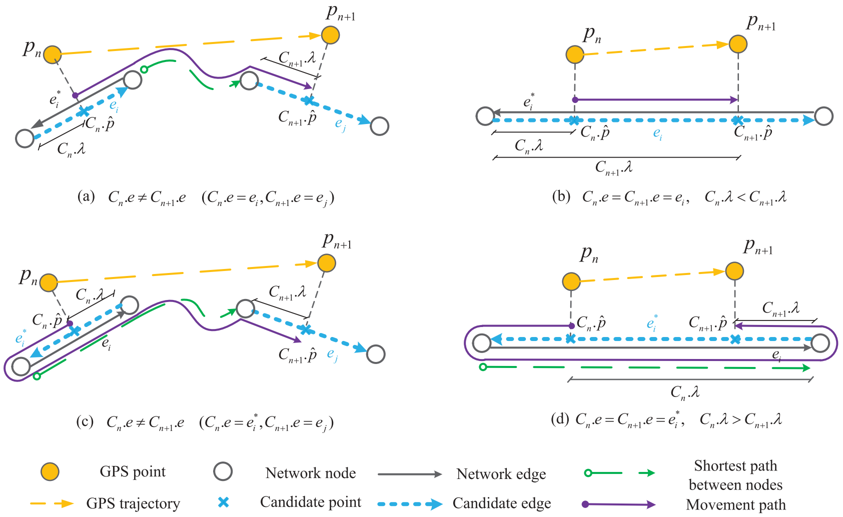
In some areas with a low density of road network, e.g. suburb, there may be less than k edges around p_n within r. Therefore, $TRCS(tr)$ is stored as a variable-length two-dimensional array of candidates. In $TRCS(tr)$, each candidate is indexed by C_n^i representing the i-th candidate matched to p_n where $1 \leq i \leq k$ and $1 \leq n \leq N$. For brevity, i is ignored sometimes thus C_n represents an arbitrary candidate matched to p_n in tr.

4.2.2. *OPI integrated with UBODT*

In this step, a transition graph is firstly built from the trajectory candidates based on HMM model, where the SP distances between candidates are queried. After that, an optimal path is inferred for the trajectory. In this section, acceleration of SP distance query utilizing UBODT is introduced first. After that, the definition of HMM and details of OPI are presented.

It has been demonstrated in Section 4.1 that SP distance between nodes of a graph can be retrieved directly from UBODT. However, additional processing is required for computing the SP distance from C_n to C_{n+1}, which is actually the distance between points on edges rather than nodes of a graph. Assume that a vehicle will always move along the direction of an edge, all the possible movements from p_n and p_{n+1} are illustrated in Figure 4 and listed below:

(1) p_n and p_{n+1} are matched to different edges, namely $C_n.e \neq C_{n+1}.e$. It implies that the vehicle first leaves the edge $C_n.e$ following the path $C_n.\hat{p} \rightarrow C_n.e.t$. After that, it enters the edge $C_{n+1}.e$ along the SP from $C_n.e.t$ to $C_{n+1}.e.s$. Finally, on edge $C_{n+1}.e$, it moves to position $C_{n+1}.\hat{p}$. Therefore, the complete path traversed is $C_n.\hat{p} \rightarrow C_n.e.t \rightarrow C_{n+1}.e.s \rightarrow C_{n+1}.\hat{p}$. Two examples are illustrated in Figure 4(a,c).



(a) $C_n.e \neq C_{n+1}.e \quad (C_n.e = e_i, C_{n+1}.e = e_j)$

(b) $C_n.e = C_{n+1}.e = e_i, \quad C_n.\lambda < C_{n+1}.\lambda$

(c) $C_n.e \neq C_{n+1}.e \quad (C_n.e = e_i^*, C_{n+1}.e = e_j)$

(d) $C_n.e = C_{n+1}.e = e_i^*, \quad C_n.\lambda > C_{n+1}.\lambda$

Figure 4. Illustration of querying SP distance from candidate C_n to C_{n+1} using UBODT. Edge e_i and e_i^* are two edges with reverse direction representing a bi-directional road segment.

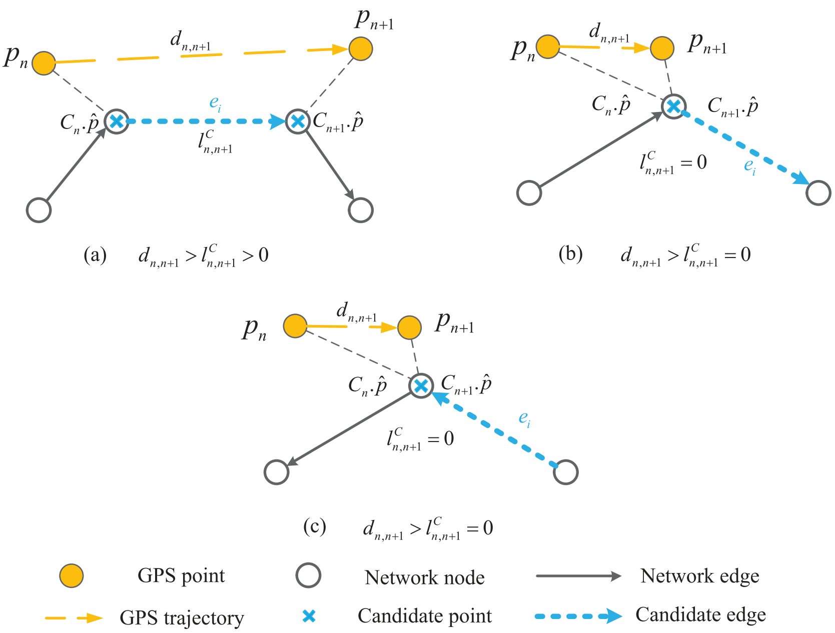


Figure 5. Degenerate cases in transition probability calculation where the shortest path distance ($l^C_{n,n+1}$) is smaller than the Euclidean distance ($d_{n,n+1}$).

(2) p_n and p_{n+1} are matched to the same edge, namely $C_n.e = C_{n+1}.e = e$. It can be further divided into two conditions below:
   (a) $C_n.\lambda \leq C_{n+1}.\lambda$. It represents that the vehicle stays on edge e when moving from p_n to p_{n+1}. The path is $C_n.\hat{p} \rightarrow C_{n+1}.\hat{p}$ with length of $C_{n+1}.\lambda - C_n.\lambda$. An example is shown in Figure 5(b).
   (b) $C_n.\lambda > C_{n+1}.\lambda$. It indicates that the vehicle leaves edge e then re-enters e following the path $C_n.\hat{p} \rightarrow e.t \rightarrow e.s \rightarrow C_{n+1}.\hat{p}$. An example is demonstrated in Figure 5(d).

Based on the above observations, the SP distance from C_n to C_{n+1} is calculated as

$$l^C_{n,n+1} = \begin{cases} C_{n+1}.\lambda - C_n.\lambda, & \text{if } C_n.e = C_{n+1}.e, C_n.\lambda \leq C_{n+1}.\lambda \\ C_n.e.L - C_n.\lambda + l^N_{n,n+1} + C_{n+1}.\lambda, & \textit{else} \end{cases} \tag{9}$$

where $l^N_{n,n+1} = l^N(C_n.e.t, C_{n+1}.e.s)$ is the SP distance from the node $C_n.e.t$ to $C_{n+1}.e.s$.

In the literature, the definition of HMM for MM differs primarily in three parts: transition probability, emission probability, and the objective function to be maximized or minimized. A commonly used definition for transition probability is the Euclidean distance divided by SP distance (Lou *et al.* 2009, Yuan *et al.* 2010b), which can be problematic in cases of dense GPS data and large search radius. Figure 5 provides 3 examples where the Euclidean distance between p_n and p_{n+1}, denoted by $d_{n,n+1}$, is greater than $l^C_{n,n+1}$. In Figure 5(a), after projection, the candidate points get closer than the GPS observations, resulting in a transition probability larger than 1. A more serious problem occurs in Figure 5(b) where a large search radius is set and the two points are matched to the same place of an edge with $l^C_{n,n+1} = 0$, which would cause a division by 0 error. The same problem exists in Figure 5(c) where they are matched to the same edge with a reverse direction. Since it is always preferred over transitions with an SP whose length is closer to the Euclidean distance, in FMM the transition probability is defined as

$$tp(C_n, C_{n+1}) = \frac{min(d_{n,n+1}, l^C_{n,n+1})}{max(d_{n,n+1}, l^C_{n,n+1})} \tag{10}$$

Therefore, in Figure 5(a) $tp < 1$, whereas in (b) and (c) $tp = 0$. These paths are likely to be eliminated in later OPI.

Regarding the emission probability, the same definition used by Lou *et al.* (2009) is adopted with the assumption that the GPS point follows a zero mean Gaussian distribution around the vehicle's true location. Formally, the emission probability of candidate C_n is defined as follows:

$$C_n.ep = \frac{1}{\sqrt{2\pi}\sigma} e^{-(C_n.dist)^2/2\sigma^2} \tag{11}$$

where σ is the standard deviation indicating GPS error.

Based on the above definitions of transition and emission probability, $TRCS(tr)$ can be treated as a transition graph where a node represents a candidate C_n and an edge represents the SP from C_n to C_{n+1}. A *candidate path* in $TRCS(tr)$ is defined as a sequence of candidates matched for each point in *tr* denoted by $\langle C_1, C_2, \ldots, C_N \rangle$. Similar with Lou *et al.* (2009), each candidate path is assigned with a score defined as follows:

$$score = \sum_{n=1}^{N-1} tp(C_n, C_{n+1}) \times C_{n+1}.ep \tag{12}$$

By employing Viterbi algorithm (Forney 1973), an optimal path *O_path* can be inferred efficiently as the path with the maximum score, which is stored as a sequence of *N* candidates matched for each point in *tr*, denoted by

$$O\_path = \langle \hat{C}_1, \hat{C}_2, \cdots, \hat{C}_N \rangle \tag{13}$$

4.2.3. *CPC and GC*

Once *O_path* is obtained, the last two steps are CPC and GC. In the CPC step, a complete path *C_path*, which is a sequence of edges traversed by *tr*, is constructed by concatenating the SPs connecting consecutive candidates in *O_path*, denoted by

$$C\_path = \langle \hat{C}_1.e, \cdots, \hat{C}_n.e, Path(\hat{C}_n.e.t, \hat{C}_{n+1}.e.s), \hat{C}_{n+1}.e, \cdots, \hat{C}_N.e \rangle \tag{14}$$

The query of $Path(\hat{C}_n.e.t, \hat{C}_{n+1}.e.s)$ is also performed in UBODT as illustrated in Section 4.1. In some previous work (Chen *et al.* 2014, Zhe and Zhang 2015), both the SP distance and path are queried and stored for each transition of candidates, which wastes computational resources because only one of them is finally exported. FMM addresses that issue by only querying the SP path after *O_path* is obtained. Finally, the corresponding geometry is constructed by concatenating the polyline for each edge in *C_path*.

4.2.4. *Handling of long-distance routing queries*

In FMM, UBODT is queried in the OPI step for SP distance and CPC step for the detailed path. In practice, it could happen that the SP distance between consecutive candidates is longer than Δ. Consequently, no SP information is returned from UBODT. To address that issue and evaluate the improvement gained by precomputation, fours modes are of FMM are designed as follows:

(M1) UBODT+Dijkstra (brute force): The first mode directly calls a Dijkstra algorithm whenever the SP distance is not found in UBODT in the OPI step or the SP path is not found in CPC step.

In M1, there exists a problem of *false distant candidates* where consecutive GPS observations p_n and p_{n+1} are close but some of their candidates are distant. As illustrated in Figure 6, p_1 and p_2 are close but the SP distances from n_1, n_3 to n_6, n_8, namely $l^N_{\{1,3\},\{6,8\}}$ are longer than Δ. The corresponding SP queries are unnecessary because $l^N_{4,6}$ is much smaller so that the paths from $\{n_1, n_3\} \rightarrow \{n_6, n_8\}$ are likely to be pruned. These unnecessary queries can degrade the performance of OPI significantly.

(M2) UBODT+Dijkstra (optimized): To avoid the aforementioned unnecessary queries, the second mode firstly returns Δ as an estimation of the SP distance if it is not found in UBODT. When p_n and p_{n+1} are close, the correct candidates should still have a much shorter SP distance and will still be inferred as part of the optimal path. If the *O_path* returned in OPI still contains the distant candidates, which implies that p_n and p_{n+1} are really distant, it switches to M1 to reprocess the trajectory candidates.

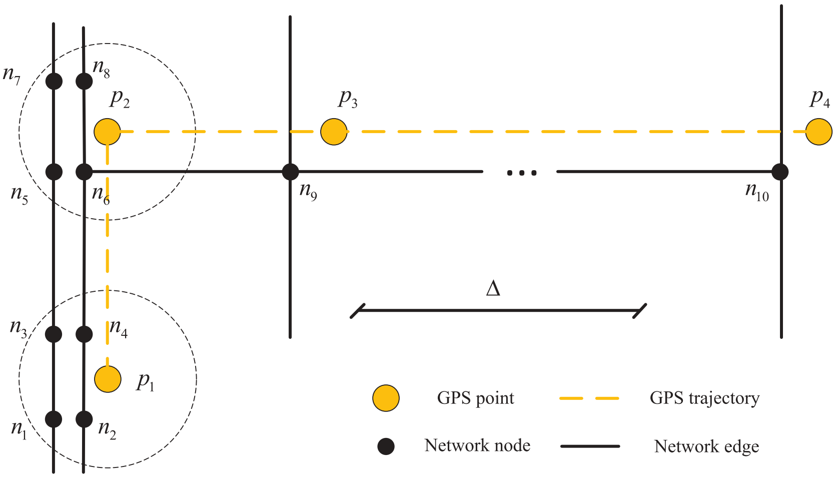


Figure 6. Illustration of unnecessary long-distance routing queries. When processing candidates of p_1 and p_2 in the OPI step, $l^N_{\{1,3\},\{6,8\}}$ and $l^N_{\{2,4\},\{5,7\}}$ are much larger than Δ, which needs to call Dijkstra routing queries. However, these queries are unnecessary because finally $n_4 \rightarrow n_6$ is likely to be inferred as the optimal path.

Taking the same case in Figure 6 as an example, when the SP distances $l^N_{(1,3),(6,8)}$ are estimated as Δ, they are still longer than $l^N_{4,6}$ and likely to be pruned in the OPI. When p_n and p_{n+1} are really distant as shown by p_3 and p_4 in Figure 6, the candidates with estimated distance of Δ can be inferred as the optimal path, which is $n_9 \rightarrow n_{10}$. Consequently, no *C_path* is found in UBODT in the CPC step. According to the definition of M2, it will switch to M1 mode, where the correct distance and path can be returned.

(M3) UBODT: The third mode also returns Δ as the distance if it is not found in UBODT in the OPI step. When a large gap still exists in *O_path*, the trajectory is exported as unmatched. Without invoking Dijkstra algorithm, M3 has a higher performance than M1 and M2. In practice, a trajectory containing a large gap provides less reliable information about the movement and might be useless. The large gap is likely to be generated from sparse GPS data, which would be less common in the future.

(M4) Dijkstra: Without querying UBODT, the last mode invokes Dijkstra algorithm all the time. The performance gained by precomputation can be evaluated by comparing with M3.

4.3. *Penalty for reverse movement in MM*

When matching real-world GPS data using the HMM model defined by Equations (10)–(12), it is frequently observed in the result that the vehicle moves back and forth on the two directed edges corresponding to a single bidirectional road segment, which is called *reverse movement*. Two such examples are displayed in Figure 7. Although it could happen that the vehicle is actually moving back and forth on a road segment, a more common case is that there exists a bidirectional road segment and the error in GPS observation is large. As a consequence, the path containing reverse movement can be returned in the OPI step because it has a higher score than the correct path, as demonstrated in Figure 7.

Detailed analysis shows that reverse movement occurs between an SP and a candidate edge rather than inside an SP. When there are two reverse edges within the search region, p_n may be matched to either of them and reverse movement occurs in both

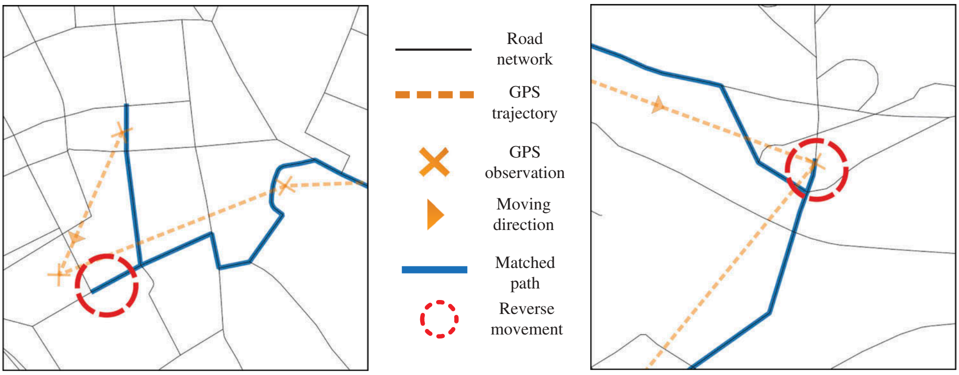
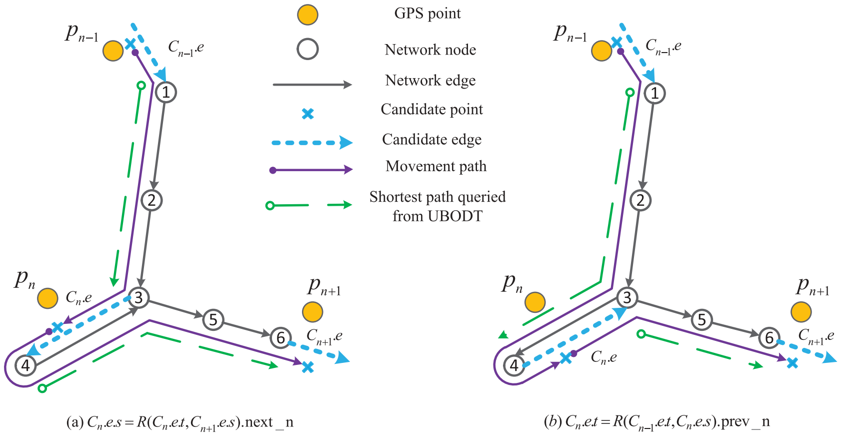


Figure 7. Two examples of reverse movement observed in matching real-world GPS data without applying penalty.



(a) $C_n.e.s = R(C_n.e.t, C_{n+1}.e.s)$.next_n

(b) $C_n.e.t = R(C_{n-1}.e.t, C_n.e.s)$.prev_n

Figure 8. Detection of reverse movement by comparing the candidate edge with the SP queried from UBODT, namely checking $C_n.e.s = R(C_n.e.t, C_{n+1}.e.s).next\_n$ in (a) and $C_n.e.t = R(C_{n-1}.e.t, C_n.e.s).prev\_n$ in (b).

cases. An example is illustrated in Figure 8 with three consecutive observations p_{n-1}, p_n and p_{n+1}. The two cases shown in Figure 8(a,b) actually represent the same movement with the nodes visited in the order of $\langle 1,2,3,4,3,5,6 \rangle$. According to Equation (9), the row in UBODT queried by $l^C_{n-1,n}$ is $R(C_{n-1}.e.t, C_n.e.s)$ and similarly $R(C_n.e.t, C_{n+1}.e.s)$ is queried for $l^C_{n,n+1}$. In Figure 8(a), p_n is matched to the edge $(3,4)$, i.e. $C_n.e = (3,4)$. Reverse movement occurs between $C_n.e$ and the SP from C_n to C_{n+1}, which can detected by $C_n.e.s = R(4,6).next\_n = 3$. In Figure 8(b), p_n is matched to the reverse edge $C_n.e = (4,3)$. Reverse movement occurs between $C_n.e$ and the SP from C_{n-1} to C_n, which is detected by $C_n.t = R(1,4).prev\_n = 3$.

Based on the two observations, reverse movements can be detected efficiently by comparing candidate edge with the SP queried from UBODT. Once reverse movement is detected from C_n to C_{n+1}, a penalized SP distance $\tilde{l}^C_{n,n+1}$ is calculated by increasing the actual SP distance controlled by a penalty factor pf, as shown below:

$$\tilde{l}^C_{n,n+1} = \begin{cases} l^C_{n,n+1} + pf * C_n.L & \text{if } record.next\_n = C_n.e.s; \\ l^C_{n,n+1} + pf * C_{n+1}.L & \text{else if } record.prev\_n = C_{n+1}.e.t; \\ l^C_{n,n+1} & \text{else.} \end{cases} \tag{15}$$

where $record = R(C_n.e.t, C_{n+1}.e.s)$. The first and second condition correspond to Figure 8 (a,b), respectively. The penalized SP distance $\tilde{l}^C_{n,n+1}$ is used in replace of $l^C_{n,n+1}$ in Equation (10). After applying penalty, the path containing reverse movement is likely to be assigned with a lower transition probability and gets eliminated in the OPI step.

4.4. *Implementation details*

An open-source implementation of the algorithm in C++ is provided at https://github.com/cyang-kth/fmm. Depending on the unit of processing, the algorithm can be performed in either *stream mode* or *batch mode*. The former is selected by default where a single trajectory is matched and followed by the next. In the latter, a batch of trajectories are matched step by step with intermediate results stored. The batch mode consumes more memory and is primarily designed for measuring the running time of different steps in FMM.

5. Case study

In this section, a case study is performed to evaluate FMM using a real-world GPS dataset. The input data and experiment set-up are firstly introduced, followed with performance and accuracy assessment. After that, the penalty for reverse movement is evaluated.

5.1 *Data description and experiment set-up*

The taxi GPS data used in this case study were originally collected in the Mobile Millennium Project (Allström *et al*. 2011). The dataset covers 1 month period from 2013-03-01 to 2013-03-31 and contains about 13 million GPS observations reported by around 1500 vehicles in Stockholm, Sweden. Each record is stored in the format (*record ID, vehicle ID, time-stamp, latitude, longitude, hired*). The last attribute *hired* is a Boolean value indicating whether the taxi is hired by consumers. Statistics of time interval and distance between consecutive GPS records are demonstrated in Figure 9. The GPS data are sparse with most time intervals between 1 min and 4 min and a majority of distances under 2 km. From the raw GPS data, a taxi trip is extracted as a consecutive sequence of records with true hired states and stored as a polyline. Totally 673,849 trip trajectories are extracted, and they are preprocessed to remove outliers according to the following criteria: (1) the trajectory contains only one point; (2) its average speed is above 150 km/h; (3) its geometry exceeds the boundary of the road network. Finally 644,695 trips (95.7%) containing 6,812,720 points are kept. The GPS error used for MM is empirically set as 100 m.

The road network used in this study is provided by Swedish National Road Database, which is displayed in Figure 10. It covers a region of 75 $km \times 89\ km$ with an area of 6675 km^2 and consists of 23,921 nodes and 57,928 directed edges where each bidirectional road segment is stored as two opposite directed edges.

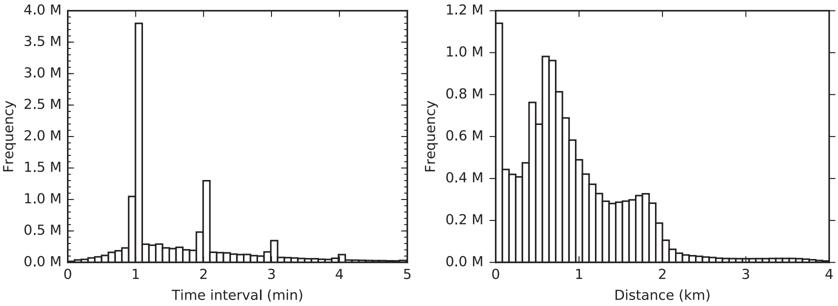


Figure 9. Histogram of time interval (left) and distance (right) between consecutive GPS observations.

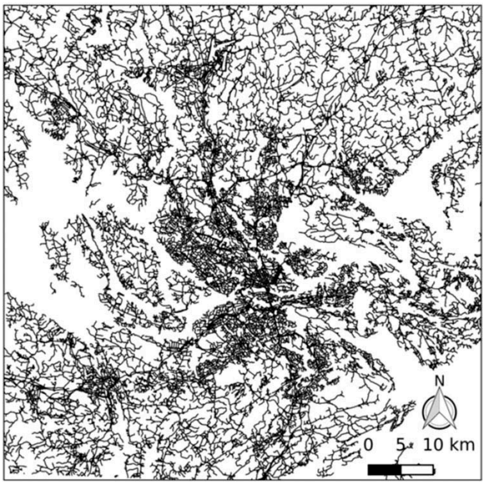


Figure 10. Study area (6675 km^2) of Stockholm and road network with 23,921 nodes and 57,928 directed edges.

All the experiments in this case study are conducted on a desktop computer running Ubuntu operating system with Intel Quad Core CPU 3.00 GHz and 4GB RAM where only a single processor is used.

5.2. *Performance assessment*

5.2.1. *Precomputing stage*

Statistics of precomputing UBODT are presented in Table 1. As the SP distance threshold Δ increases, a larger UBODT is generated in a longer but acceptable time. When $\Delta = 5km$, it takes less than 10 s to generate a UBODT with about 4.3 million rows. In previous similar work (Zhe and Zhang 2015), it took 500–600 s to generate the same kind of table on a smaller road network with $\Delta = 4km$. The corresponding running time of UBODT precomputation here is 7.471 s, about 70–80 times faster than their work. The reason for this considerable improvement is that their algorithm creates a minimum priority queue to store all pairs of origin and destination (OD) in G then iteratively pulls the OD pair with minimum distance from the queue until all SPs shorter than Δ have been retrieved. The maintenance of the priority queue adopts a strategy similar with Floyd Warshall algorithm (Floyd 1962), which is applicable for dense graph, whereas a road network graph is generally sparse. In the meanwhile, it consumes more memory to

Table 1. Statistics of precomputing UBODT from a road network with 23,921 nodes and 57,928 directed edges.

$\Delta(km)$	Number of rows	Neighbors per node	Average Running time(second)
2	811,014	33	4.265
3	1,689,430	70	5.678
4	2,861,572	119	7.471
5	4,305,012	179	9.656

store all pairs of nodes, which is less efficient than iterating through all nodes with a single source Dijkstra algorithm adopted in FMM.

5.2.2. *MM stage*

This section evaluates various aspects of MM performance including sensitivities to configurations, different modes of routing integration and running time of all steps in FMM. No penalty for reverse movement is applied in these experiments. The performance is measured by MM speed defined as the number of GPS points matched per second.

The first experiment explores the impact of the upper bound of UBODT and routing integration on the performance of MM. Three UBODTs are generated with Δ of 3, 4, and 5km, and the four integration modes described in Section 4.2.4 are tested. In the experiment, a 1-day dataset containing 27,343 trajectories and 310,137 points is map matched with configurations of $k = 8$, $r = 300m$, $H = 5,178,049$, $M = 30,000$ and no geometry output. The results are reported in Table 1. As can be observed, M3 obtains the highest speed around 30,000 points/second among the four modes, which is 167 times of the slowest mode M4 with a speed of 179 points/second. The matched percentage of M3 is slightly lower than the rest three modes. The reason is that no Dijkstra routing is invoked so that a high performance is achieved and trajectories containing a large gap are exported as unmatched. Compared with the brute force integration of Dijkstra in M1, the optimization in M2 is effective with the number of routing queries reduced substantially. When $\Delta = 5km$, M1 calls Dijkstra queries 319,945 times while M2 calls only 13,457 times. In both M1 and M2, the MM speed increases with Δ, which is resulted from fewer Dijkstra queries. However, the speed of M3 drops from 32,215 points/second to 29,102 points/second when Δ rises from 4 km to 5 km. It can be explained by the fact that a larger UBODT table is read and searched but the number of points matched is slightly increased by 1%.

Based on the above findings, the second experiment takes the 1-month dataset as input using the same configurations where only M2 and M3 are tested because M1 and M4 are too slow. The results are reported in Table 3. When matching a large number of trajectories, M3 is much faster than M2 with a sacrifice of smaller number of points matched. When $\Delta = 3km$, M3 achieves a speed of 43,816 points/second, which is 14 times of M2. However, only 87% of points are matched in M3 while in M2 it is 99.69%. By setting a larger Δ, the match percentage of M3 get increased while no significant change is observed in M2. In M3, the highest speed is reported with Δ of 4 km, which is similar with Table 2. On average, the speeds of M3 reported in Table 3 are higher than those in Table 2. The reason is that when tested with a small dataset, the input and output account for a larger proportion of the total running time. It can also be observed that matching a large number of trajectories does not consume a huge memory. With $\Delta = 5km$ and UBODT size of 4.3 million rows, it consumes only 261 megabytes (MB) memory, which can be explained by the fact that trajectories are matched in a stream way.

As a summary, M3 with only UBODT queried is the most suitable mode in practice. With appropriate setting of Δ, M3 achieves a high MM speed and matches a large proportion of trajectories at the same time. Therefore, it is set as the default integration mode in subsequent experiments.

Table 2. MM performance tested on a 1-day dataset containing 27,343 trajectories and 310,137 points with configurations of candidate set size $k = 8$, search radius $r = 300m$, hash table size $H = 5,178,049$, node multiplier $M = 30,000$ and no geometry output. The modes tested are (M1) UBODT+Dijkstra (brute force), (M2) UBODT+Dijkstra (optimized), (M3) UBODT, and M4 (Dijkstra).

Mode	$\Delta(km)$	Running time(second)	Match percentage(%)	Speed (points/second)	Dijkstra queries count
M1	3	225.35	98.86	1360.5	1,642,888
M1	4	115.95	98.86	2644.2	692,301
M1	5	69.15	98.86	4433.2	319,945
M2	3	83.15	98.83	3686.2	582,474
M2	4	18.11	98.83	16,923.4	50,349
M2	5	13.91	98.83	22,027.5	13,457
M3	3	8.99	88.15	30,390.5	0
M3	4	9.31	96.80	32,215.1	0
M3	5	10.42	97.87	29,102.8	0
M4	–	1711.07	98.86	179.2	16,644,031

Table 3. MM performance tested on the 1-month dataset containing 644,695 trajectories and 6,812,720 points with the same configurations as Table 2 (No geometry output). The modes tested are (M2) UBODT+Dijkstra (optimized) and (M3) UBODT.

Mode	$\Delta(km)$	Running time(second)	Match percentage(%)	Speed (points/second)	Memory (Megabyte)
M2	3	2075.07	99.69	3272.9	146.3
M2	4	475.72	99.69	14,276.4	201.3
M2	5	315.10	99.69	21,554.2	266.2
M3	3	135.55	87.18	43,816.6	141.9
M3	4	140.49	96.77	46,929.6	195.6
M3	5	147.20	98.24	45,471.9	261.7

The third experiment investigates the running time constitutions of FMM. The trajectories are matched in batch mode where a small buffer set is read iteratively from the input trajectories and matched step by step. The running time of each step then gets accumulated as the final result. In the experiment, the 1-month dataset is matched with $k = 8$ and different combinations of Δ, r and output mode are explored. The results are reported in Table 4. It shows that without geometry output, the most time-consuming step in FMM is CS, which occupies more than 50% of the total running time. Following CS, OPI accounts for about 20%. When $r = 300m$ and $\Delta = 5km$, writing geometry output degrades the performance substantially from 43,043 points/second to 25,105 points/second where about 40% of the time is spent in the output step. By setting a smaller r of 200 m, both CS and OPI take a shorter time because a smaller candidate set is constructed and processed. However, a smaller r may result in a lower accuracy. The

Table 4. Running time of all steps in FMM (M3) tested with the 1-month dataset containing 6,812,720 points with configurations of $k = 8$. The steps are input, candidate search (CS), optimal path inference (OPI), complete path construction (CPC), geometry construction (GC), and output.

			Running time (RT) of steps (second)								
Output mode	Δ (km)	r (m)	Input	CS	OPI	CPC	GC	Output	RT (second)	Match percentage (%)	Speed (points/ second)
No GO	5	300	17.81	90.16	31.40	5.68	–	8.04	155.43	98.24	43,043.5
No GO	4	300	7.23	89.62	28.39	5.34	–	6.49	139.38	96.77	47,186.1
No GO	5	200	11.01	61.41	26.17	5.46	–	6.61	113.17	92.56	55,717.7
GO	5	300	18.45	90.03	30.85	5.45	8.22	107.18	266.45	98.24	25,105.7

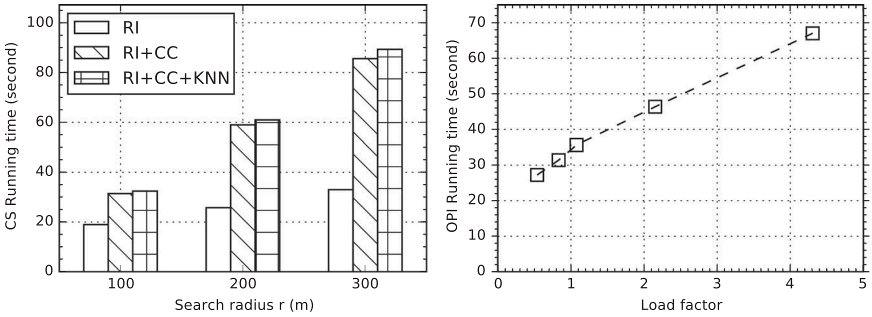
running time constitution of $\Delta = 4km$ also explains the speed improvement in Table 3 with shorter input and OPI step.

The fourth experiment investigates the bottleneck of MM by dividing CS into three steps: R-tree intersection (RI), candidate construction (CC), and KNN sort, as described in Section 4.2.1. Since the three steps are coupled with each other, CS is tested in three modes: RI, RI+CC, and RI+CC+KNN. The running time is displayed in Figure 11(a). It shows that CC is the most time-consuming step in CS, in which the linear referencing algorithm is the most complex task as the projection needs to iterate through all segments in the polyline of a road edge. Regarding the running time of OPI, it is evaluated against the load factor of UBODT *lf*, which is displayed in Figure 11(b). When a smaller *lf* is set, there are fewer elements in each bucket of UBODT on average. Consequently, OPI takes a shorter time.

Finally, the performance of FMM is compared with previous studies as summarized in Table 5. The road network used here is on a similar scale in terms of graph size while the GPS data is sparser than previous work. With $k = 8$ and $r = 300m$, FMM reports a much higher speed of around 45,000 points/second without geometry output and 25,000 points/second with geometry output. Besides, a large r of 300 m is used here as opposed to 50 m commonly used in previous studies due to the low accuracy of GPS data collected. When matching a high-quality dataset with smaller r and k, the speed has a potential to be higher according to the findings in Table 4.

5.3. *Accuracy assessment*

To evaluate the accuracy of the matched result, a ground truth dataset is collected in a similar way as Lou *et al.* (2009) by randomly selecting 30 trips that are diversely distributed in space, time, and length, which are labeled with a true path based on human judgment. Let *GT* and *MR* denote the set of complete paths in the ground truth and matched result, respectively. The accuracy of MM, denoted by *A*, is defined as the average of the overlapping ratio between *GT* and *MR* calculated below



(a) CS running time

(b) OPI running time

Figure 11. Investigation of running time for candidate search (CS) (a) and optimal path inference (OPI) (b) tested with the 1-month data with $\Delta = 5km$. In (a), the three steps are R-tree intersection (RI), candidate construction (CC), and $k = 8$ nearest neighbor sort (KNN).

Table 5. Performance comparison with the literature by single processor's MM speed.

Study	Implementation and hardware	GPS interval (second)	Network		Configurations	Performance (points/second)
			$\lvert V\rvert$	$\lvert E\rvert$		
Marchal *et al.* (2005)	Java, 4 processor 2.5 GHz CPU	1	45k	15k	N[a] $= 30$	2000
Li *et al.* (2011)	Java, Amazon Elastic Compute Cloud	20–60	11k	15k	$k = 5$ $r = 50m$	1250*[b]
Miwa *et al.* (2012)	Fortan, Dual Core 2 GHz CPU, 2 GB RAM	5–90	45k	121k	–[c]	287*
Li *et al.* (2013)	Quadcore 3.2GHz CPU	120	–	–	–	50*
Huang *et al.* (2013)	Hadoop, Single core 3.0 GHz CPU,4G RAM	20	106k	141k	–	900–6,000*
Chen *et al.* (2014)	C#, Dual-core 3.1 GHz CPU, 8GB RAM	10–120	19k	46k	$k = 10$ $r = 40m$	5582(No IO) 228(IO)
Zhe and Zhang (2015)	C++, Single-Core 3.4 GHZ CPU, 32 GB RAM	20	20k	47k	–	1000–2000
this article	C++, Quad-core 3 GHz CPU, 4GB RAM	60–240	23k	58k	$k = 8$ $r = 300m$	25,000 (GO)[d] 45,000 (No GO)

[a] *N* denotes the size of a candidate path set maintained in the algorithm.

[b] The speed suffixed with * is not directly reported in original article but calculated with information provided by the authors. In some studies where multiple cores of CPU or nodes were used, the speed reported here is rescaled to reflect single-processor's performance.

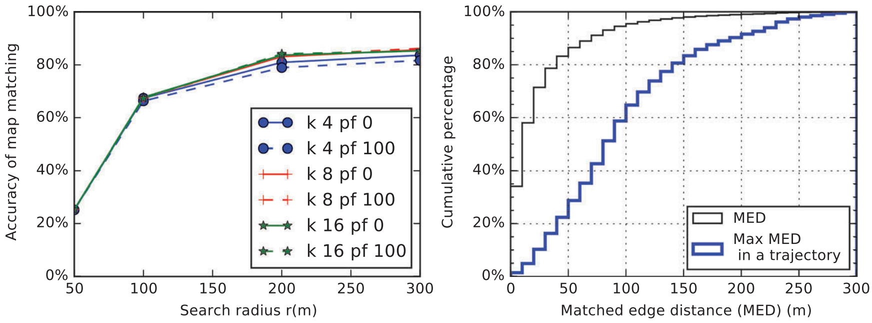
[c] The sign of – means that the information was not provided in that article.

[d] GO stands for geometry output. The speed is based on Table 3.

$$A = \frac{1}{|GT|}\sum_{i}^{|GT|}\frac{|GT[i] \cap^{M} R[i]|}{|GT[i] \cup^{M} R[i]|} \tag{16}$$

The influence of various configurations on the accuracy is displayed in Figure 12(a). When r increases from 50 m to 300 m, the accuracy grows from 25% to 80%. Table 5 shows that r is set as 50 m in most previous work. However, the corresponding accuracy is only 25% in this experiment, which results from the large error in GPS observations.

To prove the above hypothesis, let the distance from a GPS point p_i to its matched edge $\hat{C}_i.e$ be defined as *matched edge distance* (MED), namely $\hat{C}_i.dist$. Figure 12(b) demonstrates the distribution of MED of a point and the maximum of MED within a



(a) Accuracy

(b) Matched edge distance (MED)

Figure 12. Accuracy of map matching for various configurations (a) and the cumulative distribution of matched edge distance obtained from matching 1-month data with $k = 8, r = 300m$ and $\Delta = 5km$.

trajectory obtained from matching the 1-month dataset with $k = 8, r = 300m$ and $\Delta = 5km$. It can be observed that although there are 82% of points whose MED is smaller than 50 m, only 22% of the trajectories have a maximum MED under 50 m. It implies that when r is set as 50 m, 78% of the trajectories are either incorrectly matched or unmatched.

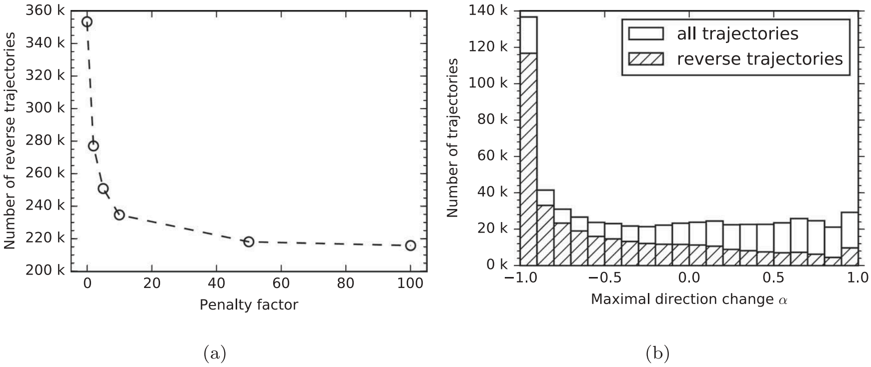
It is important to emphasize that the HMM model defined by Equations (10)–(12) is designed the same as Lou *et al*. (2009) except that several degenerate cases are settled in FMM. Because of the large error in GPS data, the accuracy of FMM is lower than previous work. In practice, it is flexible to define a different set of emission and transition probabilities in FMM with the same performance achieved. Additionally, varying *pf* and *k* exhibits no significant impact on the accuracy. It may be explained by the selection bias that the trips affected by penalty are not included in *GT*. More detailed evaluation of *pf* is provided in the next section.

5.4. *Evaluation of penalty for reverse movement*

When evaluating the penalty for reverse movement, experiments are performed on the 1-month dataset with $k = 8$, $r = 300m$, $\Delta = 5km$, and *pf* varied from 0 to 100. Reverse movement in the matched result can be detected by comparing consecutive edges in the *C_path*. A trajectory containing reverse movement is also referred as *reverse trajectory*. The distribution of reverse trajectories in the matched result with respect to *pf* is shown in Figure 13(a). As *pf* increases from 0 to 100, the number of reverse trajectories reduces rapidly from 355,000 (55%) to a lower bound of 215,000 (33%). The high percentage of reverse trajectories can be explained by the fact that a taxi is likely to change moving direction at the start of a trip after fetching a passenger. To prove that, the *maximal direction change* of a trajectory *tr* is introduced and defined as follows:

$$a = \min_{1 \leq n \leq N-2} (cos(\overrightarrow{p_n, p_{n+1}}, \overrightarrow{p_{n+1}, p_{n+2}})) \tag{17}$$

where N is the number of points in *tr* and $-1 \leq a \leq 1$. When $a = -1$, it implies a completely reverse movement in *tr* while $a = 1$ corresponds to a straight line.



(a)

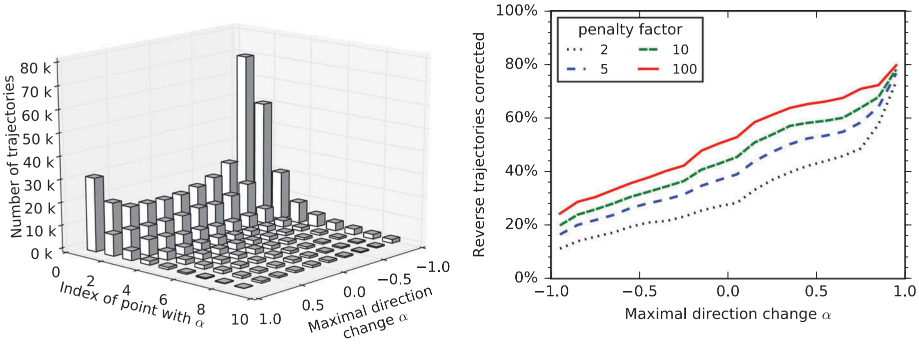
(b)

Figure 13. Distribution of reverse trajectories with respect to *pf* (a) and distribution of maximal distribution change a in the original and reverse trajectories with $pf = 0$ (b).

The distribution of α in the original and reverse trajectories in the result of $pf = 0$ is displayed in Figure 13(b). A large proportion of the original and reverse trajectories are concentrated in the region with α close to -1. The joint distribution of α and the index of point where α occurs in the original trajectories is displayed in Figure 14(a). It shows that a large number of maximal direction changes occur close to the start of a trajectory, which fits with the taxi driving behavior of changing direction after fetching a passenger. Figure 14(b) demonstrates the percentage of reverse trajectories being corrected with respect to α. As can be observed, with a larger α indicating a smaller direction change, a reverse trajectory has a higher probability of being corrected. The reason is that a trajectory with a small direction change is unlikely to contain reverse movement. Therefore, when reverse movement is generated in the matched result, there should exist another candidate path representing the true movement with a high score. Consequently, applying a penalty has a great potential to correct the result. Some examples tested with $pf = 100$ are displayed in Figure 15, which shows that the algorithm can correct reverse movement effectively.

6. Conclusions and future work

The article proposed FMM, an algorithm integrating HMM with precomputation and provided an open-source implementation. By precomputing all pairs of shortest paths in the road network under a specific distance threshold, repeated routing queries in MM are replaced with hash table search. Additionally, several degenerate cases and a problem of reverse movement were identified in previous MM algorithms and corrected in this article. Experiments on a real-world GPS dataset demonstrate that precomputation can effectively improve the performance of MM and FMM has achieved a considerable performance in both precomputing and MM stage. It takes less than 10 s to precompute a UBODT containing more than 4 million rows. Single-processor MM speeds are reported as 25,000 points/second with geometry output and 45,000 points/second without geometry output, which are much higher than previous work. Further investigation on the running time of all steps in FMM reveals that when precomputation is employed, the new bottleneck of MM is located in CS



(a) (b)

Figure 14. Distribution of maximal distribution change α and the index of point where α occurs in the original trajectories (a) and percentage of reverse trajectories being corrected with respect to α.

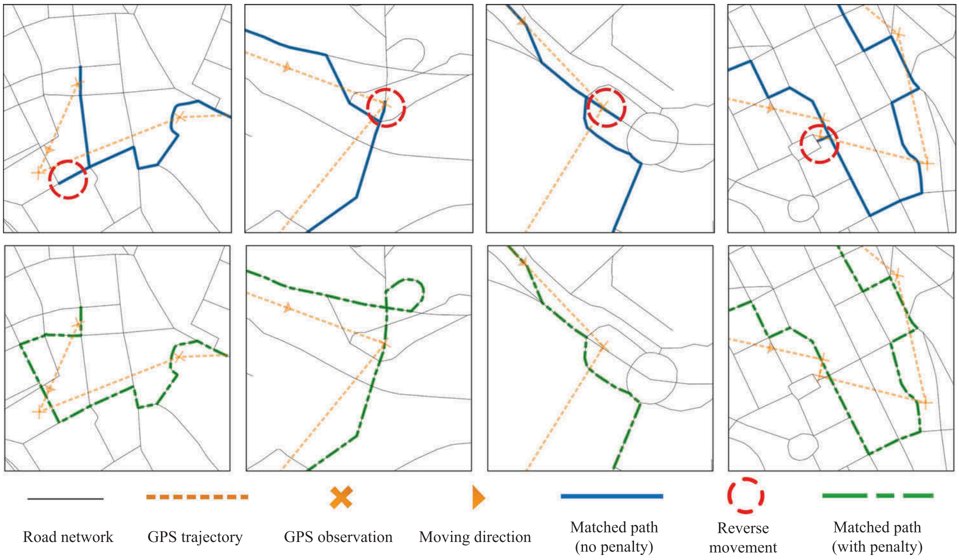


Figure 15. Examples of reverse movement corrected by applying a penalty with $pf = 100$.

and more specifically, the linear referencing function which projects a GPS point to the polyline of a road edge. Comparison of different modes of routing integration shows that exporting the large-gap trajectories as unmatched without invoking Dijkstra is the most suitable mode in practice, which achieves a high speed and matches a large proportion of trajectories. In addition, by applying a penalty, reverse movement in the result is corrected effectively.

The future work would be planned in the following directions. First, the performance of FMM can be further improved by introducing parallel computing. The new bottleneck is identified as CS and how to alleviate this problem will be investigated. Besides, useful information can also be extracted and analyzed from the matched result, e.g. travel time and movement patterns.

Acknowledgments

The authors would like to thank the editor and the reviewers for their valuable and constructive comments, which greatly improved the quality of this article. The first author is supported by Chinese Scholarship Council. We are also grateful to Clas Rydegren for providing access to the taxi GPS dataset.

Disclosure statement

No potential conflict of interest was reported by the authors.

ORCID

Can Yang http://orcid.org/0000-0001-5361-6034
Győző Gidófalvi http://orcid.org/0000-0003-1164-8403

References

Allström, A., *et al*., 2011. Mobile Millennium Stockholm. In: *2nd international conference on models and technologies for intelligent transportation systems*. Leuven, Belgium.

Castro, P.S., Zhang, D., and Li, S., 2012. 10. Urban mobility and computing. *In*: J. Kay, P. Lukowicz, H. Tokuda, P. Olivier, A. Krüger, eds. *Urban traffic modelling and prediction using large scale taxi GPS traces*. Berlin, Heidelberg: Springer Berlin Heidelberg, 57–72.

Chen, B.Y., *et al*. 2014. Map-matching algorithm for large-scale low-frequency floating car data. *International Journal of Geographical Information Science*, 28 (1), 22–38. doi:10.1080/13658816.2013.816427

Cormen, T.H., *et al*., 2001. *Introduction to algorithms*. 2nd. New York: McGraw-Hill Higher Education.

Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 (1), 269–271. doi:10.1007/BF01386390

Finkel, R.A. and Bentley, J.L., 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4 (1), 1–9. doi:10.1007/BF00288933

Floyd, R.W., 1962. Algorithm 97: shortest path. *Commun. ACM*, 5 (6), 345. doi:10.1145/367766.368168

Forney, G.D., 1973. The viterbi algorithm. *Proceedings of the IEEE*, 61 (3), 268–278. doi:10.1109/PROC.1973.9030

Goldberg, A.V. and Harrelson, C., 2005. Computing the shortest path: a search meets graph theory. In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA: SODA '05 Society for Industrial and Applied Mathematics. 156–165.

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. *SIGMOD Record*, 14 (2), 47–57. doi:10.1145/971697

Hart, P., Nilsson, N., and Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4 (2), 100–107. doi:10.1109/TSSC.1968.300136

Huang, J., *et al*., 2013. Parallel map matching on massive vehicle GPS data using mapreduce. *In*: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov. IEEE, 1498–1503.

Li, H., Kulik, L., and Ramamohanarao, K., 2014. Spatio-temporal trajectory simplification for inferring travel paths. *In*: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL '14* New York, NY, USA: ACM Press, 63–72.

Li, H., Kulik, L., and Ramamohanarao, K., 2015. Robust inferences of travel paths from GPS trajectories. *International Journal of Geographical Information Science*, 29 (12), 2194–2222. doi:10.1080/13658816.2015.1072202

Li, Q., Zhang, T., and Yu, Y., 2011. Using cloud computing to process intensive floating car data for urban traffic surveillance. *International Journal of Geographical Information Science*, 25 (8), 1303–1322. doi:10.1080/13658816.2011.577746

Li, Y., *et al*., 2013. Large-scale joint map matching of GPS traces. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 214–223.

Liao, L., Fox, D., and Kautz, H., 2007. 10. Learning and adaptive behavior. *In*: S. Thrun, R. Brooks, H. Durrant-Whyte, eds. *Hierarchical conditional random fields for GPS-based activity recognition*. Berlin, Heidelberg: Springer Berlin Heidelberg. 487–506.

Lou, Y., *et al*., 2009. Map-matching for low-sampling-rate GPS trajectories. *In*: *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, Seattle, Washington New York, NY, USA: ACM, 352–361.

Marchal, F., Hackney, J., and Axhausen, K., 2005. Efficient map matching of large global positioning system data sets: tests on speed-monitoring experiment in Zürich. *Transportation Research Record*, 1935 (1), 93–100. doi:10.3141/1935-11

Miwa, T., *et al*., 2012. Development of map matching algorithm for low frequency probe data. *Transportation Research Part C: Emerging Technologies*, 22, 132–145. doi:10.1016/j.trc.2012.01.005

Newson, P. and Krumm, J., 2009. Hidden Markov map matching through noise and sparseness. *In*: *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '09*, Nov. New York, New York, USA: ACM Press, p. 336.

Quddus, M.A., Ochieng, W.Y., and Noland, R.B., 2007. Current map-matching algorithms for transport applications: state-of-the art and future research directions. *Transportation Research Part C: Emerging Technologies*, 15 (5), 312–328. doi:10.1016/j.trc.2007.05.002

Rahmani, M., Jenelius, E., and Koutsopoulos, H.N., 2015. Non-parametric estimation of route travel time distributions from low-frequency floating car data. *Transportation Research Part C: Emerging Technologies*, 58 (Part B), 343–362. Big Data in Transportation and Traffic Engineering. doi:10.1016/j.trc.2015.01.015

Rahmani, M. and Koutsopoulos, H.N., 2013. Path inference from sparse floating car data for urban networks. *Transportation Research Part C: Emerging Technologies*, 30, 41–54. doi:10.1016/j.trc.2013.02.002

Roussopoulos, N., Kelley, S., and Vincent, F., 1995. Nearest neighbor queries. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD '95*. San Jose, California, USA New York, NY, USA: ACM, 71–79.

Wei, H., *et al*. 2012. Fast Viterbi map matching with tunable weight functions. In: *Proceedings of the 20th international conference on advances in geographic information systems - SIGSPATIAL '12 New York*, New York, USA: ACM Press, 613.

White, C.E., Bernstein, D., and Kornhauser, A.L., 2000. Some map matching algorithms for personal navigation assistants. *Transportation Research Part C: Emerging Technologies*, 8 (1), 91–108. doi:10.1016/S0968-090X(00)00026-7

Yuan, J., *et al*., 2010a. T-Drive: driving directions based on taxi trajectories. *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '10)*, 99–108.

Yuan, J., *et al*., 2010b. An interactive-voting based map matching algorithm. *In*: *2010 Eleventh International Conference on Mobile Data Management* IEEE, 43–52.

Zeng, Z., *et al*., 2015. Curvedness feature constrained map matching for low-frequency probe vehicle data. *International Journal of Geographical Information Science*, 8816 (January 2016), 1–31.

Zhe, Z. and Zhang, T., 2015. Acceleration of map matching for floating car data by exploiting travelling velocity. *In*: *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, Sept. IEEE, 2895–2899.