Noname manuscript No.
(will be inserted by the editor)

Solving the Data Sparsity Problem in Destination Prediction

Andy Yuan Xue · Jianzhong Qi · Xing Xie · Rui Zhang · Jin Huang · Yuan Li

Received: date / Accepted: date

Abstract Destination prediction is an essential task for many emerging location-based applications such as recommending sightseeing places and targeted advertising according to destinations. A common approach to destination prediction is to derive the probability of a location being the destination based on historical trajectories. However, almost all the existing techniques use various kinds of extra information such as road network, proprietary travel planner, statistics requested from government, and personal driving habits. Such extra information, in most circumstances, is unavailable or very costly to obtain. Thereby we approach the task of destination prediction by using only historical trajectory dataset. However, this approach encounters the "data sparsity problem", i.e., the available historical trajectories are far from enough to cover all possible query trajectories, which considerably limits the number of query trajectories that can obtain predicted destinations. We propose a novel method named *Sub-Trajectory Synthesis* (SubSyn) to address the data sparsity problem. SubSyn first decomposes historical trajectories into sub-trajectories comprising two adjacent locations, and then connects the sub-trajectories into "synthesised" trajectories. This process effectively expands the historical trajectory dataset to contain much more trajectories. Experiments based on real datasets show that SubSyn can predict destinations for up to ten times more query trajectories than a baseline prediction algorithm. Furthermore, the running time of the SubSyn training algorithm is almost negligible for a large set of 1.9 million trajectories, and the SubSyn prediction algorithm runs over two orders of magnitude faster than the baseline prediction algorithm constantly.

Keywords Trajectory Mining · Destination Prediction · Markov Model · Bayes' Rule

A.Y. Xue (✉)
University of Melbourne, Victoria, Australia
E-mail: andy.xue@unimelb.edu.au

J. Qi
University of Melbourne, Victoria, Australia
E-mail: jianzhong.qi@unimelb.edu.au

X. Xie
Microsoft Research Asia, Beijing, P.R.China
E-mail: xingx@microsoft.com

R. Zhang
University of Melbourne, Victoria, Australia
E-mail: rui.zhang@unimelb.edu.au

J. Huang
University of Melbourne, Victoria, Australia
E-mail: jin.huang@unimelb.edu.au

Y. Li
University of Melbourne, Victoria, Australia
E-mail: yuanl4@student.unimelb.edu.au

1 Introduction

As the usage of smart phones and in-car navigation systems becomes part of our daily lives, we benefit increasingly from various types of location-based services (LBSs) such as route finding and location-based social networking. A number of new location-based applications require *destination prediction*, for example, to recommend sightseeing places, to send targeted advertisements based on destination, and to automatically set destination in navigation systems. Fig. 1 provides a schematic with the lines representing roads and the circles representing locations of interests, which may be road intersections, sightseeing places, shopping centres, etc. If one drives from l_1 to l_4, an LBS provider may predict the most probable destinations to be l_7, l_8 and l_9 based on past popular routes taken by other drivers. As a result, the LBS provider can push advertisements of products currently on sale at those locations.

A common approach to destination prediction is to make use of historical spatial trajectories [33] of the public, avail-

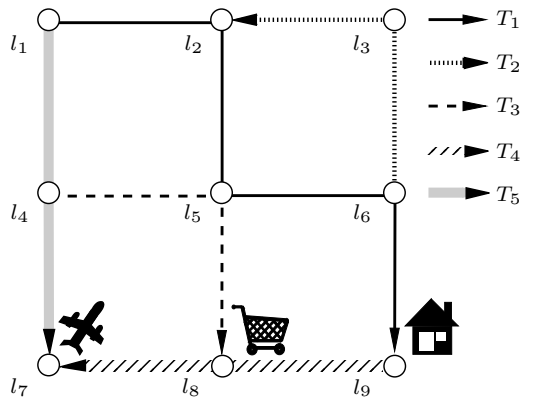


Fig. 1 An example of destination prediction

able from trajectory sharing websites [9, 23] or large sets of taxi trajectories [19]. If an ongoing trip partially matches a popular route derived from historical trajectories, the destination of the popular route is very likely to be the destination of the ongoing trip (we refer to the ongoing trip as the *query trajectory*). Shown in Fig. 1 are five historical trajectories: $T_1 = \{l_1, l_2, l_5, l_6, l_9\}$, $T_2 = \{l_6, l_3, l_2\}$, $T_3 = \{l_4, l_5, l_8\}$, $T_4 = \{l_9, l_8, l_7\}$, and $T_5 = \{l_1, l_4, l_7\}$. Each trajectory is represented by a different type of line. For instance, a trip is taken from l_1 to l_4, and this query trajectory $\{l_1, l_4\}$ matches part of the historical trajectory T_5. Therefore, the destination of T_5 (i.e., l_7) is the predicted destination of the query trajectory. In practice, each trajectory here may be associated with a weight denoting the number of historical trajectories that exactly match this one, and the most popular trajectories are used for destination prediction. The detail of this method is presented in Section 2.3 as the baseline algorithm.

The above method has a significant drawback. A location l can be predicted as a destination only when there exists a historical trajectory that matches the query trajectory and the historical trajectory's destination is l. In practice, l_8 and l_9 are also very likely to be the destination of the query trajectory, but will not be recommended to the user due to the limitation of the historical dataset. Moreover, if the query trajectory continues to l_5, the above method will not be able to predict any destination since no historical trajectory contains the query trajectory $\{l_1, l_4, l_5\}$. We refer to this phenomenon as the **data sparsity problem**. One plausible solution is to incorporate extra information such as road network, proprietary travel planner, statistics requested from government, and personal driving habits (cf. Section 2.1). Such information, however, is unavailable or very costly to obtain in most circumstances. Thereby we approach the task of destination prediction by using only historical trajectory dataset. As a result, the data sparsity problem is inevitable in practice due to the following reasons. First, the number of possible routes between all pairs of origin-destination is very large (exponential to the number of road segments in a city), and currently the largest available real-life trajectory dataset covers only a tiny portion of it. Second, even trips with the same origin-destination pair may vary on their routes, making it unlikely to have identical trajectories.

In this paper, we propose a novel method to address the data sparsity problem. Following most studies, we partition the data space using a grid, and each grid cell represents a location. Our method first decomposes all the trajectories into sub-trajectories, each of which comprises only two adjacent locations, i.e., two grid cells passed through by the sub-trajectory. The sub-trajectories are connected together into "synthesised" trajectories. As long as a query trajectory matches part of any synthesised trajectory, the destination of the synthesised trajectory can be used for destination prediction. By this means, the coverage of query trajectories on which we can make destination predictions is exponentially increased. The underlying process is formulated by a Markov model quantifying the correlation between adjacent locations/cells with transition probabilities. We can then compute the probability of reaching all the reachable locations from a given origin, and the top ranked ones are returned as predicted destinations. We call the above method ***Sub**-Trajectory **Syn**thesis* (**SubSyn**). For the aforementioned query trajectory $\{l_1, l_4, l_5, l_6\}$, the SubSyn algorithm will be able to predict other destinations such as l_8 and l_9 since they can be synthesised using sub-trajectories of T_1, T_3, and T_5. The outcome of the destination prediction process will depend on the transition probabilities and the number of top destinations to be returned. We have developed a system [29] in the form of a web application to demonstrate the use of the above method[1].

In summary, we make the following contributions in this article:

- We identify the data sparsity problem in destination prediction and propose a novel method called *Sub-Trajectory Synthesis* (SubSyn) to address this problem. SubSyn decomposes historical trajectories into sub-trajectories and connect them into "synthesised" trajectories for destination prediction. This process is formulated based on a Markov model.
- SubSyn is highly efficient because it is designed to have two phases, training and prediction. In the prediction phase, most of the data required by the algorithm are directly fetched from pre-computed matrices in the training phase. This is much faster than the baseline prediction algorithm, which has to compare query trajectories against all the historical trajectories.
- We propose a highly efficient training algorithm to significantly reduce the running time of the SubSyn training stage. This is achieved by optimising the matrix multiplications in the training stage.

[1] The demonstration system can be accessed following this link: `http://spatialanalytics.cis.unimelb.edu.au/subsyndemo/`.

- We further improve the accuracy of our prediction algorithm by employing the second-order Markov model and investigating various space partitioning techniques.
- We provide a detailed cost analysis on all the proposed algorithms.
- We conduct extensive experiments using a large real-life taxi trajectory dataset to evaluate the prediction accuracy and runtime efficiency of the SubSyn algorithms. The results show that, compared with the baseline prediction algorithm, the SubSyn prediction algorithm can predict destinations for up to ten times more query trajectories while running over two orders of magnitude faster. The SubSyn training algorithm is also highly efficient. It trains a Markov model with multiple high-order transition matrices on a dataset of 1.9 million trajectories within only several minutes.

This article is an extension of our earlier conference paper [28]. There we proposed the SubSyn method with its theoretical foundations. In this article, we extend our work by making the following additional contributions. First, we propose an improved SubSyn training algorithm (Section 5) that reduces the training time by more than five orders of magnitude compared with the training algorithm proposed in the conference paper [28]. Second, in Section 6 we improve the prediction accuracy of SubSyn by up to 20% by integrating the second-order Markov model. Fig. 2 summarises the relationship of the improved algorithm with the original SubSyn algorithm. Third, we propose two additional grid partitioning strategies (Section 7), quantile-based grid partitioning and k-d tree based grid partitioning. The k-d tree based grid partitioning strategy yields better accuracy than the uniform grid and quantile-based grid as it results in more even number of GPS points in each cell. This conforms to our theoretical analysis. The experimental results show that a more balancing grid partitioning strategy is particularly effective when computing power or the available data is limited. Fourth, we provide a detailed cost analysis on all the proposed algorithms (Section 8). Finally, we conduct a more extensive experimental study on all the algorithms using a much larger real dataset (Section 9).

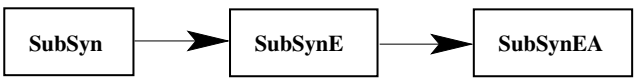


Fig. 2 Improvements on SubSyn algorithms: We improve the runtime efficiency of the training phrase of SubSyn to obtain *SubSynE* (*E* for *Efficiency*) and then enhance the prediction accuracy with *SubSynEA* (*A* for *Accuracy)*.

The remainder of the article is organised as follows: Section 2 presents related work and preliminaries. Our SubSyn method and its algorithms are described in Sections 3 and 4. We improve runtime efficiency of SubSyn in Section 5 and prediction accuracy in Section 6. Various grid partitioning techniques are compared in Section 7. Section 8 provides a detailed cost analysis of all the proposed algorithms. Experimental results are reported in Section 9. Finally Section 10 concludes the paper. Table 1 summarises the frequently used symbols.

2 Related Work and Preliminaries

We first discuss existing work in destination prediction, and compare their similarities and differences with our work in this article. Then we build a solution framework and propose a baseline algorithm based on existing work.

2.1 Related Work

Most existing destination prediction studies make use of historical trajectories, and their focuses have mainly followed two streams: (i) using external information in addition to historical trajectories to help improve the accuracy of predicted destinations; (ii) personalised destination prediction for individual users. We describe each stream in more details below.

Employing external information in addition to historical trajectories can often enhance the prediction accuracy. For example, distribution of different districts, trip time distribution, trajectory length, fastest route travel planner [12, 15, 16], accident reports, road condition, and driving habits [34] have been incorporated into Bayesian inference to compute the probabilities of predicted destinations. Similarly, context information such as time-of-day, day-of-week, and velocity has been incorporated as features in training the Bayesian network model for prediction [4, 8]. The intuition behind these studies is that certain travelling pattern which fits into the acknowledged external settings shall bring higher possibilities to locations corresponding to those external settings in the historical dataset. However, such extra information, in most circumstances, is unavailable or very costly to obtain. Thereby we approach the task of destination prediction by using only historical trajectory dataset. Our focus is to solve the data sparsity problem which cannot be solved by adding external information. Therefore, the above studies are not applicable to our problem.

Personalised destination prediction trains prediction models using historical trajectories from an individual and then predicts destinations for this same individual. Thus, these predictions for the same query trajectory from different users may vary. Natalia and Chris [18] and Patterson *et al.* [21] used a Bayesian method to predict destinations for specific individuals based on their historical transport modes. Markov model has been widely applied in predicting destinations for a specific individual as well [3, 5, 17, 22, 24]. Tiesyte

Table 1 Frequently Used Symbols

Symbol	Explanation
$\mathbb{D}$	The historical trajectory dataset
g	Granularity of a grid
m	Number of cells in a grid. $m = g^2$
$n_i(, n_s, n_c, n_d)$	i^{th} (, starting, current, destination) cell
p_{ij}	Transition probability from n_i to adjacent n_j
$p_{i \to k}$	Total transition probability from n_i to n_k
$T_{i(,j)}$	Trajectories in $\mathbb{D}$ that contain $\{n_i(, n_j)\}$
T^p	Partial or query trajectory from n_s to n_c
l_{ik}	ℓ_1 distance between n_i and n_k
α	Detour factor α
$l_{de,ik}$	Detour distance from n_i to n_k. $l_{de,ik} = \lceil \alpha l_{ik} \rceil$
M, M_{ik}	Transition matrix and its entry
M^T, M^T_{ik}	Total transition matrix and its entry

and Jensen [26] proposed a *Nearest-Neighbour Trajectory* (NNT) method that utilised distance measures to identify the historical trajectory which was the most similar to the current query trajectory. Chen *et al.* [7] used a tree structure to represent the historical movement patterns and then matched the current query trajectory by stepping down the tree. All these studies focused on predicting the repeated destinations of one or a group of specific individuals based on their own habits and historical travelling records. Our work considers a query trajectory from an unknown individual (without available personalised information). This is different from the personalised destination prediction studies. Therefore their solutions would be inapplicable to the data sparsity problem. Trajectory mining have also been studied in other contexts such as moving KNN query [10, 20], continuous moving queries on moving objects [1, 2, 13, 32], and group NN query [11]. Reference [33] contains a comprehensive survey on computation with spatial trajectories.

2.2 Preliminaries

The most popular approach to the destination prediction problem is to use a uniform grid to represent the map, and perform Bayesian inference to derive the probability of destinations based on historical trajectories [15, 16, 18, 21, 31, 34]. Our solution also follows this paradigm.

In previous studies on destination prediction [15, 16, 18, 21, 31, 34], a *uniform grid* is commonly used to help represent the dataset. It abstracts the map of a city as a two dimensional grid consisting of $m = g \times g$ congruent square cells. The granularity of this representation is a cell, i.e., all the locations within a single cell are considered to be the same object. Without loss of generality, we consider that each cell has the side length of 1 and adjacent cells have the distance of 1. This distance of 1 may correspond to a certain distance in the real world, e.g., 600 m. The whole grid is modelled as a graph where each cell corresponds to a node in the graph. Thereby, we will use the terms *node*, *cell*, and *location* interchangeably in the remainder of this paper. A trajectory can be represented as a sequence of cells according to the sequence of GPS points in the trajectory. An example of a 3×3 grid is given in Fig. 3, where the trajectory T_1 can be represented as $\{n_1, n_2, n_5, n_6, n_9\}$. By representing the trajectories using cells in a grid representation, similar trajectories are considered identical because a cell is the granularity of the graph. For example, in Fig. 3, T_0 and T_3 are identical, both of which are represented as $\{n_4, n_5, n_8\}$. It is easy to observe that when the area of each grid cell becomes smaller, the different trajectories become more distinguishable from each other in the grid model.

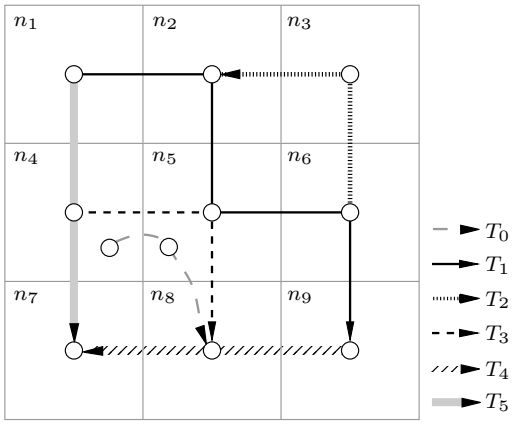


Fig. 3 A 3×3 grid on the example

Since query trajectories are incomplete trajectories whose destinations should be predicted by prediction algorithms, we denote them by *partial trajectories*, i.e., T^p. With a grid representation, two trajectories T_1 and T_2 are an *exact match* with each other if and only if their sequences of cells are identical, denoted by $T_1 = T_2$; a partial/query trajectory T^p *partially matches* a trajectory T if and only if their sequences start from the same cell and the cell sequence of T^p is fully contained by the cell sequence of T, denoted by $T^p \subset T$. In the example shown in Fig. 3, $T^p = \{n_1, n_4\}$ partially matches $T_5 = \{n_1, n_4, n_7\}$. A *non-matching query trajectory* is a query trajectory that has no partial match in the training dataset.

2.3 A Baseline Prediction Algorithm

As explained in Section 2.1, none of the methods described in existing work applies to our situation where the only available information is the GPS trajectory dataset. In what follows, we adapt the ideas of grid representation and Bayes'

rule from existing methods , and obtain a baseline prediction algorithm described as follows.

The probability of a cell n_k being the destination can be computed as the probability that n_k is the destination location n_d, conditioning on the query trajectory T^p. Formally, the probability is computed using Bayes' rule as

$$P(n_d = n_k|T^p) = \frac{P(T^p|n_d = n_k)P(n_d = n_k)}{P(T^p)} = \frac{P(T^p|n_d = n_k)P(n_d = n_k)}{\sum_{j=1}^{m} P(T^p|n_d = n_j)P(n_d = n_j)}. \quad (1)$$

The prior probability $P(n_d = n_k)$ can be easily computed as the number of trajectories terminating at n_k divided by the number of trajectories in the dataset. Formally,

$$P(n_d = n_k) = \frac{|T_{n_d=n_k}|}{|\mathbb{D}|}, \quad (2)$$

where $|\mathbb{D}|$ is the cardinality of the historical trajectory dataset, and $|T_{n_d=n_k}|$ is the number of trajectories in $\mathbb{D}$ that terminate at location n_k. Similarly, we can compute the prior probability $P(n_d = n_j)$ by replacing n_k with n_j in the above equation. As indicated by Equation (2), only locations that are the destinations of historical trajectories will have non-zero prior probabilities, reflecting the fact that only popular locations are of interest. Therefore, the crux of using Equation (1) is computing the *likelihood function* $P(T^p|n_d = n_k)$ (note that $P(T^p|n_d = n_j)$ can be computed by letting n_k be n_j). In order to solve this issue, we first count the number of trajectories satisfying two conditions: (i) it is partially matched by the query trajectory T^p; (ii) it terminates at location n_k. The count is then divided by the number of trajectories that terminate at location n_k to serve as the likelihood function. Formally,

$$P(T^p|n_d = n_k) = \frac{|\{T_{n_d=n_k}|T^p \subset T_{n_d=n_k}\}|}{|T_{n_d=n_k}|}, \quad (3)$$

where $|\{T_{n_d=n_k}|T^p \subset T_{n_d=n_k}\}|$ denotes the number of trajectories that satisfy both the aforementioned conditions and $|T_{n_d=n_k}|$ denotes the number of trajectories that terminate at a location in n_k.

The above method, which uses *trajectory matching* as the likelihood function, will be used as the *baseline prediction algorithm*. As discussed in Section 1, this method suffers from the data sparsity problem. If the query trajectory T^p cannot be partially matched by any trajectory in $|\mathbb{D}|$, then the numerator in Equation (3) $|\{T_{n_d=n_k}|T^p \subset T_{n_d=n_k}\}|$ equals 0, and the probability of any cell being the destination is 0 (i.e., $P(T^p|n_d = n_k) = 0$). Consequently, no predicted destination can be found for this query trajectory.

It should be made clear that the baseline algorithm is **not** directly borrowed from existing work, but rather an adapted version that utilises the same approach (e.g., grid representation and Bayes' rule). Taking Ziebart et al [34] as an example, the authors use Markov decision process as the likelihood function in Bayes' rule whereas our baseline algorithm use trajectory matching techniques as the likelihood function. The data used for the likelihood function in [34] are generated from trajectories, surveys, and road network.

3 Destination Prediction Based on Sub-Trajectory Synthesis

To overcome the data sparsity problem, we propose a novel method named ***Sub**-Trajectory **Syn**thesis* (SubSyn). The general idea is to first decompose each historical trajectory into segments of length 1 and then synthesise the segments in all possible combinations. This process effectively expands the historical trajectory dataset to cover a greater number of query trajectories. Thereby it overcomes the data sparsity problem. This section lays the theoretical foundation of the SubSyn method, which will be implemented in Section 4 as the SubSyn training and prediction algorithms.

Following previous studies based on Bayesian inference framework, the SubSyn method first represents the map with a grid. Then it uses a Markov model to model the trajectories, where each state corresponds to a grid cell and the transition from one state to another corresponds to travelling from one cell to another. This way any trajectory can be modelled as a series of state transitions in the Markov model.

We train the Markov model (i.e., learn the state transition probabilities) as follows. Given a set of historical trajectories, we first obtain the state transition series for each trajectory. Then the number of historic trajectories that contain a transition from state n_i to state n_j divided by the number of historic trajectories that contain a transition starting from state n_i yields the transition probability from n_i to n_j (Section 3.1).

We synthesise trajectories (i.e., generate state transition series) of different lengths and compute the *total transition probability* of each pair of states $\{n_i, n_k\}$, which is the sum of the transition probabilities of all the synthesised trajectories that starts at n_i and ends at n_k. The total transition probability will be used for computing the probabilities of the destinations (Section 3.2).

When a query trajectory arrives, we match the query trajectory with the synthesised trajectories, and derive the probability that the ending state of each matched trajectory being the predicted destination based on the precomputed total transition probabilities (Section 3.3).

We detail the theoretical foundation of the SubSyn method in the following subsections.

3.1 Modelling Trajectories using Markov Model

We model the trajectories using a Markov model by associating a state to each cell n_i in the grid. The transition probability of travelling from a location n_i to an adjacent location n_j is denoted by p_{ij}. We compute p_{ij} and p_{ji} for every pair of adjacent cells n_i and n_j. Fig. 4 shows the Markov model state transition diagram for the example in Fig. 3. These transition probabilities are conditional probabilities and can be computed as the number of trajectories that contain the sequence $\{n_i, n_j\}$ divided by the number of trajectories that contain the cell n_i. Formally,

$$p_{ij} = P(n_j|n_i) = \frac{|T_{i,j}|}{|T_i|}. \tag{4}$$

For each pair of adjacent cells in the grid, we precompute the transition probabilities using Equation (4). These probabilities are stored as entries of a two-dimensional $m \times m$ matrix where one dimension corresponds to the cell of current state and the other dimension corresponds to the next state. In the following sections, we denote the transition matrix and its entries by M and M_{ik}, respectively. Matrix (5) is the transition matrix of the example presented in Fig. 3 and Fig. 4.

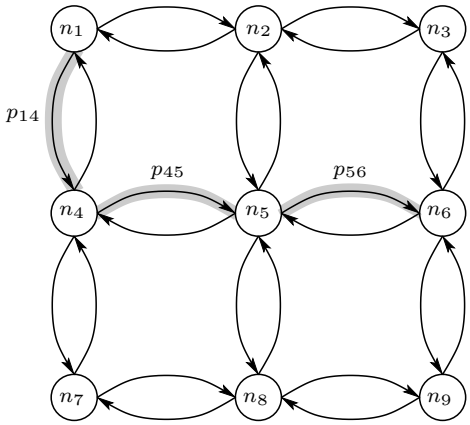


Fig. 4 A 3×3 Markov model state transition diagram

As indicated by Equation (4), since the first-order Markov model is used here, only the current state determines the probability of transiting to the next state. Higher order Markov models could be applied by involving previous states in addition to the current state in computing the probabilities. However, even the second-order Markov model will require a large number of trajectories as training data and a much higher memory occupation as shown by previous studies [5, 6]. Due to this reason, we first present our method using the first-order Markov model (which was published in the conference version [28]). As we have later substantially improved the efficiency and memory usage of the training algorithm, and we have obtained much more real data, we are able to further investigate the second-order Markov model and we will present how we tackle the difficulties in Section 6.

$$M = \begin{pmatrix} 0 & p_{12} & 0 & p_{14} & 0 & 0 & 0 & 0 & 0 \\ p_{21} & 0 & p_{23} & 0 & p_{25} & 0 & 0 & 0 & 0 \\ 0 & p_{32} & 0 & 0 & 0 & p_{36} & 0 & 0 & 0 \\ p_{41} & 0 & 0 & 0 & p_{45} & 0 & p_{47} & 0 & 0 \\ 0 & p_{52} & 0 & p_{54} & 0 & p_{56} & 0 & p_{58} & 0 \\ 0 & 0 & p_{63} & 0 & p_{65} & 0 & 0 & 0 & p_{69} \\ 0 & 0 & 0 & p_{74} & 0 & 0 & 0 & p_{78} & 0 \\ 0 & 0 & 0 & 0 & p_{85} & 0 & p_{87} & 0 & p_{89} \\ 0 & 0 & 0 & 0 & 0 & p_{96} & 0 & p_{98} & 0 \end{pmatrix} \tag{5}$$

3.2 Computing the Total Transition Probability and the Path Probability

In Section 3.1, a Markov model based transition matrix M is constructed and filled with probabilities of travelling from a cell to its adjacent cells. This process is effectively decomposing each trajectory in $\mathbb{D}$ into a set of sub-trajectories of length 1 (i.e, ordered pairs of adjacent cells). For instance, the trajectory T_1 in Fig. 1 is decomposed into $T^p_{1,2}$, $T^p_{2,5}$, $T^p_{5,6}$, and $T^p_{6,9}$ which in turn contribute to the transition probabilities p_{12}, p_{25}, p_{56}, and p_{69}, respectively. Using this set of sub-trajectories, we can synthesise other trajectories that also start at n_1 and end at n_9 but pass through other intermediate cells, e.g., $T'_1 = T^p_{1,4}$, $T^p_{4,5}$, $T^p_{5,8}$, and $T^p_{8,9}$. We synthesise all possible trajectories for every pair of cells n_i and n_k. Then we can obtain the *total transition probability* from cell n_i to cell n_k, which will be incorporated in Section 3.3 to formulate the posterior probability $P(n_d = n_k|T^p)$.

The Total Transition Probability: The following example demonstrates the concept of the total transition probability. By referring to Equation (5) and Fig. 4, the probability of travelling from n_1 to n_6 is found to be zero in M (i.e., $M_{16} = 0$) because M stores the probability of travelling from one cell to another in exactly one step, and there is no way of travelling between these two cells within one step. Furthermore, when M is multiplied by itself to form M^2, its entries are the probabilities of travelling from one cell to another in two steps. In general, M^r ($r \in [0, \infty)$) holds the probabilities of transition from one cell to another in exactly r steps (i.e., M^r holds r-step transition probabilities). Since the ℓ_1 distance between n_1 and n_6 (i.e., l_{16}) is 3, the probability of travelling from n_1 to n_6 via all the shortest paths can be found in matrix entry $M^3_{1,6}$. However, two problems also remain: (i) the ℓ_1 distance does not necessarily correspond to the actual travelling distance from n_1 to n_6 because sometimes a small detour is taken due to various reasons. Hence we wish to find the sum of r-step transition probabilities of various steps; (ii) the number of paths

from one cell to another is infinitely large without restrictions, i.e., $r \in [l_{16}, \infty)$. In order to solve these two problems, we introduce a detour factor α as an additional condition that the length of all possible paths between two cells, e.g., n_i, n_k, will not exceed $l_{ik} + \lceil \alpha l_{ik} \rceil$, where l_{ik} is the ℓ_1 distance between n_i and n_k. In a general case, we assume that $\alpha \in [0, 1)$. By examining the dataset used in our experiment, it is found that the median distances of all trips is 1.2 of the ℓ_1 distance between the starting and ending cells (cf. Section 6.1). Therefore, we use $\alpha = 0.2$ as a constant. We denote $\lceil \alpha l_{ik} \rceil$ by $l_{de,ik}$, and define the *total transition probability* as follows.

Definition 1 ***Total Transition Probability*** *The total transition probability of travelling from one cell n_i to another cell n_k, denoted by $p_{i \to k}$[2], is the sum of the r-step transition probabilities of all possible paths (with the detour distance restriction) between n_i and n_k. Formally:*

$$p_{i \to k} = \sum_{r=l_{ik}}^{l_{ik}+l_{de,ik}} M_{ik}^r = M_{ik}^{l_{ik}} + M_{ik}^{l_{ik}+1} + \cdots + M_{ik}^{l_{ik}+l_{de,ik}}. \quad (6)$$

In the equation above, the last term after expanding the summation equation, $M_{ik}^{l_{ik}+l_{de,ik}}$, gives the probability of travelling from n_i to n_k in exactly $l_{ik} + l_{de,ik}$ steps. For instance, suppose the detour factor $\alpha = 0.2$, then in Equation (5) and Fig. 4, $l_{16} = 3$ and $l_{de,16} = \lceil 0.2 \times 3 \rceil = 1$, so, $p_{1 \to 6} = M_{1,6}^3 + M_{1,6}^4$. The usage of Equation (6) will be revealed in the following subsection (Section 3.3) when formulating the posterior probability equation.

Synthesis of Paths for Query Trajectories We introduce the definition of *path probability* which will be used to compute the posterior probability $P(n_d = n_k | T^p)$. The path probability is the probability of a person travelling from one location to another given a path. The path is the query trajectory provided by a user. The value of the path probability can be obtained through multiplying the transition probabilities between all pairs of cells in this query trajectory T^p. For example, given the transition matrix M, the path probability of moving from a location in n_1 to another location in n_6 via the path $T^p_{1,4,5,6}$ can be obtained as follows: $P(T^p_{1,4,5,6}) = p_{14} \cdot p_{45} \cdot p_{56}$ where p_{14}, p_{45}, and p_{56} are the transition probabilities in the matrix M between consecutive and adjacent[3] cell pairs $\{n_1, n_4\}$, $\{n_4, n_5\}$, and $\{n_5, n_6\}$, respectively. In general, given any query trajectory $T^p_{1,2,\cdots,k}$, the definition of path probability is:

$$P(T^p) = P(T^p_{1,2,\cdots,k}) = \prod_{i=1}^{k} p_{i(i+1)}. \quad (7)$$

We discuss a special case of applying Equation (7), and use a 3×3 grid as an example. If a query trajectory provided by user contains non-adjacent consecutive cells (e.g., $T^p = \{n_1, n_5, n_6\}$ where $\{n_1, n_5\}$ are not adjacent cells), the transition probability p_{15} would be zero. This situation could occur when, for instance, the user's GPS device failed to report its position for several minutes. In such cases, we use *linear interpolation* to fill the gap between these two cells before calculating the path probability. More specifically, we first use a *linear polynomial* (i.e., straight line) to connect the two GPS points located in n_1 and n_5. Suppose that the linear polynomial passes n_4. Then we will insert n_4 to T^p to form $T^p = \{n_1, n_4, n_5, n_6\}$.

3.3 Computing the Destination Probability

To predict the destination is to compute the posterior probability $P(n_d = n_k | T^p)$ as defined by Equation (1) in Section 2.2. The method is described below.

We first transform the posterior probability such that it can be computed using the probabilities defined in the Markov model. We expand T^p to be the series of cells contained in T^p, denoted by $\{n_s, \cdots, n_c\}$, where n_s and n_c denote the starting and ending cells of T^p.

$$\begin{aligned} P(n_d | T^p) &= P(n_d | n_s, \cdots, n_c) \\ &= P(n_s, n_d | n_s, \cdots, n_c) \\ &= P(n_s, n_d | T^p) \\ &= \frac{P(T^p | n_s, n_d) \cdot P(n_s, n_d)}{P(T^p)} \\ &= P(T^p | n_s, n_d) \cdot \frac{P(n_d | n_s) \cdot P(n_s)}{P(T^p)}. \end{aligned}$$

Since when we compute $P(n_d | T^p)$ for different n_d the starting cell of the trajectory T^p does not change, $P(n_s)$ is a constant and it does not affect the comparative result of $P(n_d | T^p)$ for different n_d. Therefore, $P(n_d | T^p)$ is proportional to $P(T^p | n_s, n_d) \cdot \frac{P(n_d | n_s)}{P(T^p)}$. Formally,

$$P(n_d | T^p) \propto P(T^p | n_s, n_d) \cdot \frac{P(n_d | n_s)}{P(T^p)}, \quad (8)$$

where $P(T^p | n_s, n_d)$ and $P(n_d | n_s)$ will be explained below, and $P(T^p)$ was described in Equation (7).

[2] Note the difference between $p_{i \to j}$ and p_{ij}. The latter (without the arrow) is the transition probability defined in Markov model, and its definition was given in Equation (4).

[3] *consecutive* cells are two cells next to each other in a trajectory; *adjacent* cells are two cells next to each other in a grid.

We compute the conditional probability $P(T^p|n_s, n_d)$ by expanding T^p, and obtain

$$\begin{aligned} P(T^p|n_s, n_d) &= \frac{P(T^p, n_s, n_d)}{P(n_s, n_d)} \\ &= \frac{P(n_s, \cdots, n_c, n_d)}{P(n_s, n_d)} \\ &= \frac{P(n_s, \cdots, n_c) \cdot P(n_c, n_d)}{P(n_s, n_d)} \\ &= \frac{P(T^p) \cdot P(c \to d)}{P(s \to d)} \\ &= \frac{P(T^p) \cdot p_{c\to d}}{p_{s\to d}}, \end{aligned} \tag{9}$$

where $P(T^p)$ is the path probability of T^p, as given by Equation (7); $p_{c\to d}$ is the total transition probability of travelling from the current cell of T^p, n_c, to a predicted destination n_d; and $p_{s\to d}$ is the total transition probability of travelling from the starting cell of T^p, i.e., n_s, to a predicted destination n_d.

The other probability $P(n_d|n_s)$ that appeared in Equation (8) can be obtained by the following equation (for all trajectories starting at n_i, the proportion that finish at n_k):

$$P(n_d = n_k|n_s = n_i) = \frac{|T_{n_s=n_i, n_d=n_k}|}{|T_{n_s=n_i}|}. \tag{10}$$

Integrating Equation (9) into Equation (8), we obtain the posterior probability equation that computes the destination probabilities:

$$\begin{aligned} P(n_d|T^p) &\propto P(T^p|n_s, n_d) \cdot \frac{P(n_d|n_s)}{P(T^p)} \\ &= \frac{P(T^p) \cdot p_{c\to d}}{p_{s\to d}} \cdot \frac{P(n_d|n_s)}{P(T^p)} \\ &= \frac{p_{c\to d}}{p_{s\to d}} \cdot P(n_d|n_s) \end{aligned}$$

$$\therefore \boxed{P(n_d|T^p) \propto \frac{p_{c\to d}}{p_{s\to d}} \cdot P(n_d|n_s)}, \tag{11}$$

where the components are obtained by using Equations (6) and (10). Given a query trajectory, we will apply this equation to compute the destination probabilities for each cell, and rank them in order to retrieve the most probable destinations. Note that when computing destination probability, we do not need to obtain the exact value, but rather a relative probability for ranking purposes only. Therefore, a proportional relationship would suffice.

In this section, we have established the theoretical model for destination prediction. We still need efficient ways of applying this model because the model involves expensive computations such as repeated matrix multiplication. In the following section, we propose algorithms to tackle the runtime efficiency issues.

4 Algorithms

The algorithms comprise the training and prediction phases in order to efficiently process destination prediction queries. The train phase computes and stores the components in Equation (11) such that, in the prediction phase, one can simply retrieve these components and compute the destination probability $P(n_d|T^p)$ using Equation (11) directly. We present the SubSyn-Training algorithm in Section 4.1 and SubSyn-Prediction in Section 4.2. Improvements to SubSyn will be presented in Section 5 to achieve reduced running time and Section 6 for enhanced prediction accuracy.

4.1 SubSyn-Training

The SubSyn training stage is responsible for computing the components in Equation (11), i.e., $p_{i\to k}$ and $P(n_d|n_s)$. The computation of $P(n_d|n_s)$ will be done simply by counting satisfactory trajectories, so we will focus on the algorithm to compute $p_{i\to k}$.

The total transition probability $p_{i\to k}$ defined by *Definition 1* will be stored in a matrix M^T and used in the prediction phase. As the definition shows in Equation (6), the total transition probabilities are extremely expensive to compute. For example, for a 50×50 grid, M is a $50^2 \times 50^2 = 2500 \times 2500$ matrix. For a travel distance of ten cells, $p_{i\to k} = M^{10}_{ik} + M^{11}_{ik} + M^{12}_{ik}$ (since $\lceil 10 \times 1.2 \rceil = 12$) which means that the matrix multiplication operation needs to be performed on the large matrix M^r more than 30 times where each matrix multiplication requires $O(m^3)$ $(m = 2500)$ time (cf. Section 8.1 for a detailed discussion on complexities of matrix multiplication algorithms). While already inefficient, the same operation needs to be carried out for all pairs of cells $\{n_i, n_k\}$ ($2500^2 = 6.25 \times 10^6$ pairs). It is therefore infeasible in terms of running time.

In the conference paper [28], we presented an algorithm called *SubSyn-Training* to solve this runtime efficiency problem, as summarised in Algorithm 1. This algorithm works as follows. Firstly, Equation (6) is reformed so that few redundant computations are carried out.

$$p_{i\to k} = \sum_{r=l_{ik}}^{l_{ik}+l_{de,ik}} M^r_{ik}$$

$$\therefore \boxed{p_{i\to k} = \left(M^{l_{ik}} \sum_{r=0}^{l_{de,ik}} M^r \right)_{ik}} \tag{12}$$

For the purpose of explanation, we use a 50×50 grid as an example without any loss of generality. Since the longest distance $\max(l_{ik}) = l_{1,2500}$ is $2 \times (50 - 1) = 98$, the maximum possible value of $l_{de,ik} = \lceil 0.2 l_{ik} \rceil$ is therefore

$\lceil 0.2 \times 98 \rceil = \lceil 19.6 \rceil = 20$. By taking advantage of the concept of dynamic programming, an array of size 21 can be used to store all M^r, $r \in [0, 20]$ which are computed in ascending exponent order (Algorithm 1: lines 5 to 6) such that only 19 matrix multiplications are required since M^1 is already constructed and M^0 is the identity matrix I. Afterwards, we sequentially add each array element to the next element to form $\sum_{r=0}^{l_{de,ik}} M^r$ (the second factor in Equation (12)) where i is an array index (Algorithm 1: lines 7 to 8). The benefit is evident because all possible values of $\sum_{r=0}^{l_{de,ik}} M_{ik}^r$ can be directly retrieved from this array for further computations.

Algorithm 1: SubSyn-Training($\mathbb{D}, g$)

1 $M^T \leftarrow 0$; // total transition matrix
2 $l_{de,\max} \leftarrow \lceil 0.2 \times 2(g-1) \rceil$; // maximum detour distance
3 $A[0] \leftarrow I$; // an array to store $\sum_{r=0}^{\lceil \alpha l \rceil} M^r$
4 $A[1] \leftarrow M \leftarrow \mathbb{D}$; // construct transition matrix
5 **for** $i \leftarrow 2$ **to** $l_{\max}$ **do**
6 $A[i] \leftarrow M \cdot A[i-1]$; // $A[i]$ now holds M^i
7 **for** $i \leftarrow 1$ **to** $l_{\max}$ **do**
8 $A[i] \leftarrow A[i] + A[i-1]$; // $A[i]$ now holds $\sum_{r=0}^{i} M^r$
9 $list \leftarrow \emptyset$; // a list to store all cell pairs
10 **foreach** n_i *in grid* **do**
11 **if** M_{i*} *contains only zero entries* **then**
12 **continue**;
13 **foreach** n_k *in grid* **do**
14 **if** M_{*k} *contains only zero entries* **then**
15 **continue**;
16 add cell pair (n_i, n_k) to $list$;
17 **sort** $list$; // increasing order of ℓ_1 distance
18 $M_{power} \leftarrow M$; // matrix to store intermediate result
19 $M_{temp}^T \leftarrow M_{power} \cdot A[1]$; // matrix to store intermediate result
20 $l_{prev} \leftarrow 1$; // record distance of previous iteration
21 **foreach** $(n_i, n_k) \in list$ **do**
22 **while** $l_{ik} \geq l_{prev} + 1$ **do**
23 $M_{power} \leftarrow M \cdot M_{power}$;
24 $M_{temp}^T \leftarrow M_{power} \cdot A[l_{de,ik}]$;
25 l_{prev}++;
26 $M_{ik}^T \leftarrow M_{temp,ik}^T$; // i.e., $p_{i \to k}$
27 $P(n_d|n_s) \leftarrow \mathbb{D}$;
return: M, M^T, and $P(n_d|n_s)$

Regarding the first factor $M^{l_{ik}}$ in Equation (12), we could use the same strategy except that in order to store this term for all pairs of cells, too much memory is required, especially in a grid with high granularity. For example, in a 50×50 grid, each M requires $50^2 \times 50^2 \times 8$ Bytes ≈ 47.7 MB of storage space. Since the maximum ℓ_1 distance in such a grid is 98, the total amount of memory required will exceed 4.67 GB (i.e., 98×47.7 MB), which may not be accommodated by a regular computer. Therefore, we need to seek a scalable and robust solution. Fortunately, these matrices do not have to be stored. Instead, we enumerate all pairs of cells in the grid, sort these pairs in ascending order of their distance between each other, and compute $p_{i \to k}$ in this order (Algorithm 1: lines 9 to 17). Using Fig. 4 and Fig. 5 as an example, all pairs of cells and their distances are generated to be $\{n_1, n_2\}(1)$, $\{n_1, n_4\}(1)$, $\cdots$, $\{n_1, n_3\}(2)$, $\cdots$, $\{n_2, n_6\}(2)$, $\cdots$, $\{n_1, n_8\}(3)$, $\cdots$,$\{n_1, n_9\}(4)$, $\{n_9, n_1\}(4)$. In order to compute the total transition probability of each pair, $M^{l_{ik}}$ and $\sum_{r=0}^{l_{ik}} M^r$ are retrieved from memory, and they are multiplied together to form a matrix containing $p_{i \to k}$ of distance 1 (Algorithm 1: lines 18 to 19). The total transition probabilities of all pairs of distance 1 can be obtained directly from this matrix (M_{power} in Algorithm 1). After all pairs of distance 1 are obtained, M^2 is computed by multiplying M, and a matrix containing $p_{i \to k}$ of distance 2 is obtained (Algorithm 1: lines 22 to 25). Utilising this algorithm, only less than 150 matrix multiplications are carried out (in the case of a 50×50 grid) to compute all total transition probabilities, whereas the intuitive approach requires millions of matrix multiplications. During the process, each found $p_{i \to k}$ is stored in a separate matrix M^T (Algorithm 1: line 26). We call this matrix the *total transition matrix*, and it holds the same number of entries as the transition matrix M.

	n1	n2	n3	n4	n5	n6	n7	n8	n9
n1	–	1	2	1	2	3	2	3	4
n2	1	–	1	2	1	2	3	2	3
n3	2	1	–	3	2	1	4	3	2
n4	1	2	3	–	1	2	1	2	3
n5	2	1	2	1	–	1	2	1	2
n6	3	2	1	2	1	–	3	2	1
n7	2	3	4	1	2	3	–	1	2
n8	3	2	3	2	1	2	1	–	1
n9	4	3	2	3	2	1	2	1	–

Fig. 5 ℓ_1 Distance Matrix of all pairs of cells in a 3×3 grid

In the process of enumerating all pairs of cells, more computational steps could be eliminated by pruning unpromising pairs of cells. For two cells n_i and n_k, if either the entire row containing n_i, M_{i*} (Algorithm 1: lines 11 to 12), or the entire column containing n_k, M_{*k} comprises only zero entries, the pair is discarded (Algorithm 1: lines 14 to 15). It indicates a lack of training data in these cells and the probability is always confined to zero in such case. Hence there is no need to compute their total transition probabilities.

4.2 SubSyn-Prediction

The *SubSyn-Prediction* algorithm is responsible for computing the posterior probability $P(n_d = n_k|T^p)$ in Equation (11), which is the probability of n_k being the destination given a query trajectory T^p.

We summarise the *SubSyn-Prediction* algorithm in Algorithm 2. Given a total transition matrix M^T in a grid, all pairs of $P(n_d|n_s)$, and a query trajectory T^p, the *SubSyn-Prediction* algorithm works as follows:

1. For each cell n_k, we compute the destination probability $P(n_d = n_k|T^p)$ based on $p_{s\to d}$, $p_{c\to d}$ and $P(n_d = n_k|n_s)$ using Equation (11);
2. We select the top-k cells according to their destination probabilities;
3. We return the top-k elements that have been selected.

Algorithm 2: SubSyn-Prediction$(M^T, P(n_d|n_s), T^p)$

```
list ← ∅;                                   // a list to store the output
foreach n_k in grid do
    retrieve p_{c→k} and p_{s→k} from M^T;
    compute P(n_d = n_k|T^p) ;
    store P(n_d = n_k|T^p) in list ;
select top-k elements in list;
return: top-k elements
```

Discussion: We briefly discuss the performance of the baseline prediction and SubSyn-Prediction algorithm. It is clear that SubSyn-Prediction requires little time to run due to the already-completed training phrase. In the prediction phrase, we simply retrieve probability values computed by SubSyn-Training and use Equation (11) to compute destination probability for each cell. The baseline prediction algorithm, in contrast, needs to perform a sequential scan through the entire historical trajectory dataset for each query trajectory in order to find matching trajectories. As shown by the cost analysis (Section 8) and experimental study (Section 9), the running time of SubSyn-Prediction is reduced from that of the baseline prediction algorithm by several orders of magnitudes.

5 Improving Runtime Efficiency of SubSyn

Although the *SubSyn-Training* algorithm has avoided excessive amount of matrix computation, it still requires noticeable number of matrix multiplications. If we examine the core equation in *SubSyn-Training* algorithm (Equation 12), we observe that neither of the factors (i.e., matrices M^{ik} or $\sum_{r=0}^{l_{de,ik}} M^r$) is sparse even though M itself is. Therefore time complexity of $O(m^3)$ is still high since we cannot apply sparse matrix multiplication techniques to reduce the time complexity.

In this section, we take advantage of the fact that M is a sparse matrix and propose improvements to the *SubSyn-Training* algorithm, which drastically reduces the training time and confines memory occupation within acceptable range. We call the resultant algorithm the *Improved SubSyn-Training* algorithm. With this improvement, we are able to run SubSyn on much finer grid (e.g., $g = 70$) and design algorithm accordingly to improve prediction accuracy (cf. Section 6).

This algorithm is based on the observation that, in our transition matrix M (cf. Equation (5)), there are at most four non-zero entries in each row, i.e, we can only travel *directly* from one cell to its four adjacent cells (i.e., left, right, top, and bottom cells). As a result, multiplying two $m \times m$ transition matrices M and M' only requires $4m^2$ multiplications, i.e., there are m^2 entries in the resultant matrix M'' to be computed, and each entry M''_{ik} is computed by $M''_{ik} = \sum_{j=1}^{m} M_{ij}M'_{jk}$, which takes only four multiplications. Therefore, the computation complexity can be reduced to $O(m^2)$.

To take full advantage of this observation, we rewrite Equation (6) as follows so that the transition matrix M, which is sparse, is involved in as many matrix multiplications as possible. In other words, we minimise the number of matrix multiplications that do not involve M since other matrices are much denser.

We start by replacing $l_{de,ik}$ with $\lceil \alpha l_{ik} \rceil$ in Equation (6):

$$\begin{aligned} p_{i\to k} &= \sum_{r=l_{ik}}^{l_{ik}+l_{de,ik}} M^r_{ik} \\ &= \sum_{r=l_{ik}}^{l_{ik}+\lceil \alpha l_{ik} \rceil} M^r_{ik} \\ &= \left(\sum_{r=l_{ik}}^{l_{ik}+\lceil \alpha l_{ik} \rceil} M^r \right)_{ik} . \end{aligned} \tag{13}$$

We further introduce two ancillary arrays, where each element is a matrix, to achieve the objective of reducing complexity, and we name them arrays A and B. Let $A[l]$ be the l^{th} element of A, where

$$A[l] = \sum_{r=l}^{l+\lceil \alpha l \rceil} M^r .$$

Then we can rewrite Equation (13) as

$$p_{i\to k} = \left(\sum_{r=l_{ik}}^{l_{ik}+\lceil \alpha l_{ik} \rceil} M^r \right)_{ik} = (A[l_{ik}])_{ik} .$$

Now the problem of computing $p_{i\to k}$ for all pairs of (n_i, n_k) becomes computing $A[1], A[2], \cdots, A[l_{\max}]$, where $l_{\max}$ is the longest ℓ_1 distance between two cells in the grid.

Next we derive a recurrence relationship on $A[l]$ and $A[l+1]$ using the transition matrix M to help compute $A[l]$. By definition $A[l+1] = \sum_{r=l+1}^{l+1+\lceil\alpha(l+1)\rceil} M^r$. Comparing it with $A[l]$, the main deference lies in $\lceil\alpha(l+1)\rceil$ and $\lceil\alpha l\rceil$. Let $\phi(l)$ be $\lceil\alpha(l+1)\rceil - \lceil\alpha l\rceil$. Since $\alpha l < \lceil\alpha l\rceil \leq \alpha l + 1$, We have

$$\begin{aligned}\phi(l) &= \lceil\alpha(l+1)\rceil - \lceil\alpha l\rceil \\ &< \alpha(l+1) + 1 - \alpha l \\ &= \alpha + 1.\end{aligned}$$

Besides, the detour factor α satisfies that $0 \leq \alpha < 1$ as discussed in Section 3.2. Thus, we have $\phi(l) < 2$. Further, since both $\lceil\alpha(l+1)\rceil$ and $\lceil\alpha l\rceil$ are integers, ϕ must be an integer. Therefore, $\phi(l) = 0$ or 1. We will discuss the two cases separately below:

(i) If $\phi(l) = 0$, we have $\lceil\alpha(l+1)\rceil = \lceil\alpha l\rceil$, and

$$\begin{aligned}A[l+1] &= \sum_{r=l+1}^{l+1+\lceil\alpha(l+1)\rceil} M^r \\ &= M \sum_{r=l}^{l+\lceil\alpha l\rceil} M^r \\ &= M \cdot A[l].\end{aligned}$$

(ii) If $\phi(l) = 1$, we have $\lceil\alpha(l+1)\rceil = \lceil\alpha l\rceil + 1$, and

$$\begin{aligned}A[l+1] &= \sum_{r=l+1}^{l+1+\lceil\alpha(l+1)\rceil} M^r \\ &= M \sum_{r=l}^{l+\lceil\alpha l\rceil+1} M^r \\ &= M \cdot \left(A[l] + M^{l+\lceil\alpha l\rceil+1}\right).\end{aligned}$$

Apart from array A, we define array B whose l^{th} element $B[l] = M^{l+\lceil\alpha l\rceil+1}$. Then we have:

$$\begin{aligned}B[l+1] &= M^{l+1+\lceil\alpha(l+1)\rceil+1} \\ &= \begin{cases} M^{l+1+\lceil\alpha l\rceil+1} & \text{if } \phi(l) = 0 \\ M^{l+1+\lceil\alpha l\rceil+2} & \text{if } \phi(l) = 1 \end{cases} \\ &= \begin{cases} M \cdot B[l], & \text{if } \phi(l) = 0 \\ M^2 \cdot B[l], & \text{if } \phi(l) = 1 \end{cases}\end{aligned}$$

We compute the first elements of the two arrays A and B as follows.

$$\begin{aligned}A[0] &= \sum_{r=0}^{0+\lceil\alpha\times 0\rceil} M^r = M^0 = I \\ B[0] &= M^{0+\lceil\alpha\times 0\rceil+1} = M^1 = M\end{aligned}$$

We have discussed the means to reduce the complexity of the SubSyn-Training algorithm above. To summarise, we rewrite Equation (6) as:

$$\therefore \quad \boxed{p_{i\to k} = (A[l_{ik}])_{ik}}, \tag{14}$$

where

$$A[l] = \begin{cases} I, & \text{if } l = 0 \\ M \cdot A[l-1], & \text{if } \phi(l-1) = 0 \\ M \cdot (A[l-1] + B[l-1]), & \text{if } \phi(l-1) = 1 \end{cases}$$

$$B[l] = \begin{cases} M, & \text{if } L = 0 \\ M \cdot B[l-1], & \text{if } \phi(l-1) = 0 \\ M^2 \cdot B[l-1], & \text{if } \phi(l-1) = 1 \end{cases}$$

Now we can use Equation (14) to simplify the computation of the total transition probabilities, as summarised by Algorithm 3. In this algorithm, we have avoided all the matrix multiplications that do not involve the transition matrix M. Meanwhile, we just need to store one element for each of the two arrays A and B, which reduces the space consumption significantly as shown by the cost analysis provided in Section 8.

Algorithm 3: SubSynE-Training($\mathbb{D}, g$)

```
M^T ← 0;                                          // total transition matrix
l_max = 2 × (g − 1);   // the longest distance between two cells
A[0] ← I;                                         // an array that stores M^l Σ_{r=0}^{⌈αl⌉} M^r
B[0] ← M ← 𝔻;                                     // an array used when φ(l) = 1
for l ← 1 to l_max do
    if φ(l − 1) = 0 then
        A[l] = M · A[l − 1] ;
        B[l] = M · B[l − 1] ;
    else if φ(l − 1) = 1 then
        A[l] = M · (A[l − 1] + B[l − 1]) ;
        B[l] = M · (M · B[l − 1]) ;
    foreach (n_i, n_k) satisfies l_ik = l do
        M^T_ik = A[l]_ik;                         // i.e., p_{i→k}
P(n_d|n_s) ← 𝔻 ;
return: M, M^T, and P(n_d|n_s)
```

6 Improving Prediction Accuracy of SubSyn

The substantially improved runtime efficiency of SubSyn enables us to investigate more computationally expensive techniques such as finer grid partition and higher-order Markov models. These techniques help improve the prediction accuracy of the SubSyn algorithm by up to 20% as experiments in Section 9 show.

6.1 Methods for the Improvement

We improve the prediction accuracy of SubSyn from two aspects: studying the effect of the detour factor α and using the second-order Markov model.

Detour Factor α: The detour factor α introduced in Section 3.2 quantifies unusual behaviour and models how much detour people usually take. When we set $\alpha > 0$, we assume that people usually take a detour that is greater than the ℓ_1 distance between their origin-destination pairs. Since the ℓ_1 distance between an origin-destination pair is usually much larger than the shortest path in Euclidean space, a detour factor $\alpha > 0$ corresponds to a path much larger than the shortest path and therefore is unusual behaviour.

The effect of α is more significant when the grid is coarse. This can be explained as follows. First of all, we found that the unusual behaviour of detour mostly comes from the process of grid partitioning rather than actual detour taken by drivers. In a coarse grid (e.g., $g = 30$), when we allocate GPS points to cells, it generates larger detour because the resolution of such grid is low. Consequently, the proportion of training trajectories that have a large detour is high, and a larger α tends to yield better prediction accuracy. During an experiment conducted for the conference paper [28], we discovered that setting $\alpha = 0.2$ for the SubSyn algorithm gives the best prediction accuracy for a grid with granularity $g = 30$.

We have also learnt that a larger dataset reduces the effect of α because a larger dataset contains more samples, which make our model more deterministic and less random. Now we have obtained much more GPS trajectory data and developed SubSynE, an algorithm enabling us to run on much finer grid. According to the experimental result in Fig. 6, we found that the effect of α in finer grids and on large datasets does become very small. As a matter of fact, setting α to 0 in these cases yield better prediction accuracy. Even though $\alpha = 0.2$ still gives the best prediction accuracy in median-size grids (e.g., $g = 20$ and $g = 30$), but the prediction error across all values of α and g become more stable due to abundant data provision. The difference in the prediction error for any grid we have tested is within a range of 2% when varying the value of α.

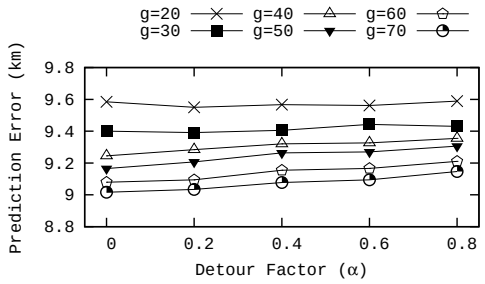


Fig. 6 Choosing the Detour Factor α

Based on the above investigation, we decided to set α to zero. This simplifies the SubSyn algorithm, reduces the space usage and reduces the running time. Hence it is an important step before employing the second-order Markov model.

With α set to 0, the equation for total transition probability, Equation (6), can be simplified to

$$p_{i \to k} = M_{ik}^{l_{ik}}. \tag{15}$$

We will use this equation in the second-order Markov model to develop the new algorithm.

The Second-Order Markov Model: Because of the improved training runtime efficiency and a larger trajectory dataset gathered, we are able to investigate *the Second-Order Markov Model* for its ability to improve prediction accuracy.

A naive way to apply the second-order Markov model to SubSyn is to use three-dimensional matrix for the transition matrix M instead of a two-dimensional one, because in the second-order Markov model the transition is defined to be amongst three states: the current state (i.e., current position n_c) and two previous states. However, this will significantly increase both the time and space complexities exponentially. We will describe how we confine the transition matrix to remain in two-dimensional space.

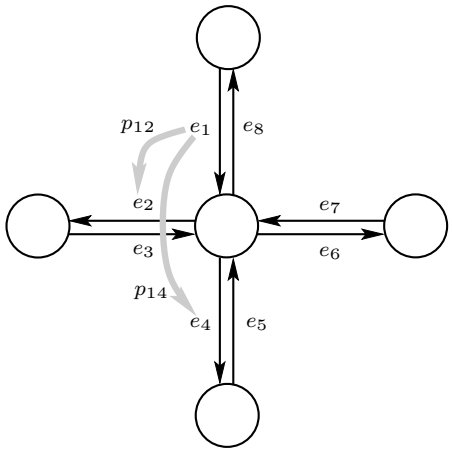


Fig. 7 The second-order Markov model state transition diagram

In a first-order Markov model, we define a state to be a cell. However, in the second-order Markov model, we define a state to be an edge. The transition probability now becomes the probability of travelling from an edge e_i to an adjacent edge e_j (i.e., the end cell of e_i is the start cell of e_j) and is denoted by p_{ij}. This effectively uses two-dimensional transition matrix in the second-order Markov model. Such transition probabilities are conditional probabilities and can be computed as the number of trajectories that contain the sequence $\{e_i, e_j\}$ divided by the number of trajectories that

contain the edge e_i. Formally,

$$p_{ij} = P(e_j|e_i) = \frac{|T_{i,j}|}{T_i}. \quad (16)$$

The transition probabilities of adjacent edges can be precomputed by the above equation and the rest will be computed in Section 6.2. Matrix (17) is the transition matrix of the example presented in Fig. 7.

$$M = \begin{pmatrix} 0 & p_{12} & 0 & p_{14} & 0 & p_{16} & 0 & p_{18} \\ 0 & 0 & p_{23} & 0 & 0 & 0 & 0 & 0 \\ 0 & p_{32} & 0 & p_{34} & 0 & p_{36} & 0 & 0p_{38} \\ 0 & 0 & 0 & 0 & p_{45} & 0 & 0 & 0 \\ 0 & p_{52} & 0 & p_{54} & 0 & p_{56} & 0 & 0p_{58} \\ 0 & 0 & 0 & 0 & 0 & 0 & p_{67} & 0 \\ 0 & p_{72} & 0 & p_{74} & 0 & p_{76} & 0 & p_{78} \\ p_{81} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (17)$$

Fig. 7 shows a 3×3 grid with four corner cells omitted for simplicity. If we include them, the matrix (17) would expand from 8×8 to a 24×24 square matrix since there are a total of 24 directed edges in a grid with $g = 3$. Therefore, the runtime efficiency and space occupation in the second-order Markov model is still much higher than the original SubSyn algorithm even if their complexities are the same as presented in Section 8. Therefore the improvement in runtime efficiency (i.e., the SubSynE-Training algorithm) and larger trajectory dataset are essential in using the second-order Markov model.

6.2 Computing the Destination Probability

We integrate the two aforementioned methods for improving prediction accuracy into SubSynE, and obtain the algorithm, which we call *SubSynEA* (*A* for *Accuracy*). The main differences between SubSynEA and SubSynE are outlined below.

Since the destination probability $P(n_d|T^p)$ uses node n_d instead of edge e_d , we convert the edge probability to the node probability by summing up the probabilities of all the edges that have the same destination as shown in Equation (18).

$$P(n_d|T^p) = \sum_{e_d \text{ ends at } n_d} P(e_d|T^p). \quad (18)$$

One exception occurs when T^p contains only one cell, i.e., $T^p = \{n_s\}$. In this case, we simply let

$$P(n_d|T^p) = P(n_d|n_s),$$

where $P(n_d|n_s)$ can be calculated by Equation (10). In other cases, we expand T^p to be the series of edges contained in T^p, denoted by $\{e_s, \cdots, e_c\}$, where e_s and e_c denote the starting and current edges of T^p. Formally, the destination probability $P(e_d|T^p)$ is given by

$$P(e_d|T^p) = P(e_d|e_s, \cdots, e_c) = P(e_d|e_c) = p_{c \to d}. \quad (19)$$

The above changes to the SubSynE algorithm are summarised below in Algorithms 4 and 5.

Algorithm 4: SubSynEA-Training $(\mathbb{D}, g)$

```
M^T ← 0;                                   // total transition matrix
l_max = 2 × g;          // the longest distance between two edges
A ← I;                                     // a matrix that stores M^r
for l ← 1 to l_max do
    A = M × A ;
    foreach (e_i, e_k) satisfies l_ik = l do
        M^T_ik = A_ik;                     // i.e., p_{i→k}
P(e_d|e_s) ← D ;
return: M, M^T, and P(n_d|n_s)
```

Algorithm 5: SubSynEA-Prediction$\big(M^T, P(e_d|e_s), T^p\big)$

```
list ← ∅;                                  // a list to store the output
P(n_d|T^p) ← 0;                  // initialize all profanities to 0
foreach e_k in grid do
    retrieve p_{c→k} from M^T;
    P(n_d = the end of e_k|T^p) += p_{c→k} ;
select top-k probabilities among all cells;
return: top-k elements
```

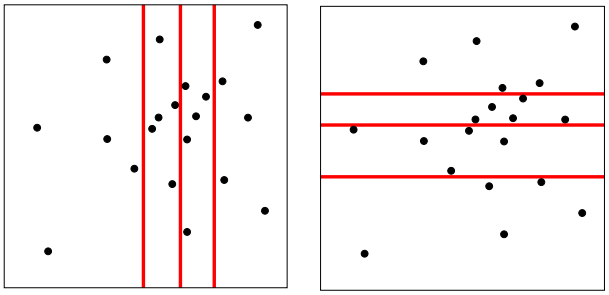
7 Comparison of Various Grid Partitioning Strategies

Until now we have assumed a uniform grid to represent the map, as what existing studies have done. In Section 7.1, we investigate other partitioning strategies, a quantile-based grid partitioning and a k-d tree based grid partitioning. Then we demonstrate mathematically in Section 7.2 that these strategies that yield more balanced number of points in each cell achieve smaller information loss than a uniform grid, and it leads to better prediction accuracy as verified by experiments in Section 9.5.

7.1 Two Partitioning Strategies

Quantile-Based Partitioning Strategy: The quantile-based strategy also represents the map with a $g \times g$ grid, but the grid cells have different sizes following the quantile distribution of the trajectory data points. Specifically, we first partition

the data space vertically and horizontally based on the data densities in these two dimensions, respectively, as shown in Figs. 8(a) and (b). A vertical (horizontal) partitioning can be easily done by a sequential scan of the data points in ascending latitude (longitude) coordinate. During the scan we count the number of data points met so far. Once this number reaches $\frac{d}{g}$, a new column (row) is produced, where d denotes the total number of location points and g denotes the number of columns (rows) to be obtained. The process repeats for g times for each dimension. This way, we obtain a grid that follows the data point density in linear time. The resultant grid is shown in Fig. 10(b).



(a) Step One: Vertical partitioning (b) Step Two: Horizontal partitioning

Fig. 8 Quantile-Based Grid Partitioning for a grid with 20 GPS data points and $g = 4$. Each partition contains exactly 5 GPS data points.

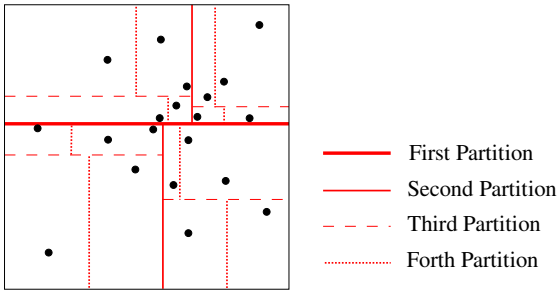


Fig. 9 K-d Tree Based Grid Partitioning for a grid with 20 GPS data points. The space is partitioned four times (i.e., $g = 4$). Each partition contains either 1 or 2 GPS data points.

K-d Tree Based Partitioning Strategy: K-d trees are binary trees, in which each node is associated with a coordinate in dimension k. Each non-leaf node divides the space into two half-spaces. The points in each of the two half-spaces are stored in the left and right branches of the current node. In this section, we ignore the data indexing structure in k-d tree and focus on the space partitioning method. As illustrated in Fig. 9, we divide the space using either horizontal or vertical lines into two partitions of equal number of points. In other words, we pick the median point in one dimension and separate the space from its location. Then in each of the two partitions, we repeat the same process recursively in a round-robin fashion for all dimensions until a desired grid granularity is reached.

Discussion Compared with a uniform grid as shown in Fig. 10, both quantile based and k-d tree based partitioning strategies are able to achieve higher prediction accuracy as the experiment result in Fig. 20 shows. This is because, in a city, regions with dense trajectory coverage (e.g., CBD region) will be mapped to more cells with each cell having smaller area. It improves the prediction accuracy of queries that involve these regions. The best result is given by k-d tree based partitioning strategy because it achieves the most even distribution of points. In what follows, we formally investigate the relationship between prediction accuracy and grid partitioning strategies using the *entropy* of grid representations, which quantifies the information loss when the trajectory data points are represented by the cells of a grid.

7.2 Theoretical Analysis

In this subsection, we show analytically that both quantile based and k-d tree based partitioning strategies are superior to uniform grid in terms of the amount of information loss during the process of grid partitioning.

Let i be the index of a grid cell and p_i be the probability that a data point locates in the i^{th} grid cell. Then by definition, the entropy of a grid representation, denoted by H, can be computed as

$$\begin{aligned} H &= -\sum_{i} p_i \ln p_i \\ &= \sum_{i=1}^{m} p_i \ln \frac{1}{p_i}, \end{aligned} \tag{20}$$

where m denotes the number of cells of a grid.

We derive upper and lower bounds for H, denoted by $H_{\max}$ and $H_{\min}$, respectively.

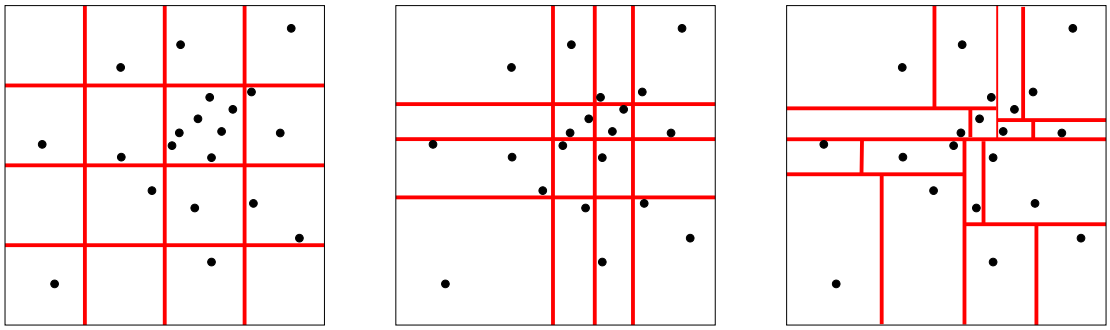
An Upper Bound of the Entropy: Since $f(x) = \ln x$ is a convex function, Jensen's inequality [14] applies. Thus,

$$H = \sum_{i=1}^{m} p_i \ln \frac{1}{p_i} \leq \ln \sum_{i=1}^{m} p_i \frac{1}{p_i} = \ln m.$$

Hence,

$$H_{\max} = \ln m. \tag{21}$$

According to Jensen's inequality [14], this maximum will be reached when p_i is the same for any i, i.e., $\forall i \in [1, m], p_i = \tilde{p}$, where $\tilde{p} = \frac{1}{m} \sum_{i=1}^{m} p_i = \frac{1}{m}$ is the average of all p_i.



(a) Uniform Grid (b) Quantile-Based Grid (c) K-d Tree Based Grid

Fig. 10 A visual comparison amongst all grid partitioning strategies for a grid with 20 GPS data points and $g = 4$. The k-d tree based grid has the most even distribution of GPS points. Quantile-based grid achieves less evenly distributed GPS points when compared with the k-d tree based grid. Uniform grid has the worst GPS point distribution.

A Lower Bound of the Entropy: We use the first-order Taylor series to expand function $f(x) = \ln x$:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(\xi)(x - x_0)^2$$
$$= \ln x_0 + \frac{1}{x_0}(x - x_0) + \frac{1}{2!}\left(-\frac{1}{\xi^2}\right)(x - x_0)^2,$$

where $\frac{1}{2!}(-\frac{1}{\xi^2})(x-x_0)^2$ denotes the Lagrange remainder for some real number ξ between x and x_0. Since this remainder is either zero or negative, we have

$$f(x) \leq \ln x_0 + \frac{1}{x_0}(x - x_0).$$

Replacing x with p_i and x_0 with $\tilde{p} = \frac{1}{m}$, we have

$$\sum_{i=1}^{m} p_i \ln p_i \leq \sum_{i=1}^{m} p_i \left(\ln \tilde{p} + \frac{1}{\tilde{p}}(p_i - \tilde{p})\right)$$
$$= \sum_{i=1}^{m} -p_i \ln m + m p_i^2 - p_i$$
$$= -\ln m \sum_{i=0}^{m} p_i + m \sum_{i=1}^{m} p_i^2 - \sum_{i=1}^{m} p_i$$
$$= -\ln m + m \sum_{i=1}^{m} p_i^2 - 1.$$

Further, we make use of the variance σ^2 of the grid to quantify the diversity of distribution of GPS points in a grid. Let the variance be

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(p_i - \tilde{p})^2, \tag{22}$$

which indicates how diversely the trajectory data points are distributed in the grid cells. The implication of the variance σ^2 is as follows. Compared with a uniform grid, both quantile based and k-d tree based grids have **smaller** variances because, for these two partitioning strategies, the data points tend to distribute evenly in all grid cells, and hence have similar p_i values.

We use the variance to find the lower bound of H by expanding Equation (22) as follows:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(p_i - \tilde{p})^2$$
$$= \frac{1}{m}\sum_{i=1}^{m}\left(p_i^2 - 2p_i\tilde{p} + \tilde{p}^2\right)$$
$$= \frac{1}{m}\sum_{i=1}^{m} p_i^2 - \frac{1}{m}\sum_{i=1}^{m} 2p_i\tilde{p} + \frac{1}{m}\sum_{i=1}^{m}\tilde{p}^2$$
$$= \frac{1}{m}\sum_{i=1}^{m} p_i^2 - \frac{2}{m^2} + \frac{1}{m^2}$$
$$= \frac{1}{m}\sum_{i=1}^{m} p_i^2 - \frac{1}{m^2}.$$

Thus, $\sum_{i=1}^{m} p_i^2 = m\sigma^2 + \frac{1}{m}$, and we have

$$\sum_{i=1}^{m} p_i \ln p_i \leq -\ln m + m\left(m\sigma^2 + \frac{1}{m}\right) - 1$$
$$= -\ln m + m^2\sigma^2 + 1 - 1$$
$$= -\ln m + m^2\sigma^2.$$

Therefore, a lower bound of the entropy H is

$$H_{\min} = \ln m - m^2\sigma^2. \tag{23}$$

Conclusion: After giving the definitions of entropy and diversity factor (Equations (20) and (22)), and the upper and lower bounds of the entropy (Equations (21) and (23)), we show that both quantile based and k-d tree based partitioning strategies have less information loss than that of the uniform grid.

Applying the upper and lower bounds of H, we have

$$\ln m - m^2\sigma^2 \leq H \leq \ln m. \quad (24)$$

Hence,

$$\boxed{\lim_{\sigma^2 \to 0} H = \ln m = H_{\max}}.$$

Based on the equation above, the entropy of a grid representation reaches its maximum value $H_{\max}$ as the variance σ^2 approaches zero. Also since the term $m^2\sigma^2$ in Equation (24) is directly proportional to the variance σ^2, we conclude that the variance is inversely proportional to H. Compared to the uniform grid partitioning strategy, the two proposed strategies result in data points more evenly distributed in different cells, and hence a smaller variance, i.e., a smaller variation in the number of data points per cell. This leads to a larger H, i.e., less information loss. Such analysis is verified in the experimental study in Section 9.5. □

8 Cost Analysis

In this section, we analyse the time and space complexities of all the algorithms described in this paper in both the training and the prediction phases: (i) SubSyn, (ii) SubSynE, (iii) SubSynEA, and (iv) the baseline prediction algorithm (cf. Section 2.3).

8.1 The Training Algorithms

In the following analysis, we let $l_{\max} = 2(g-1)$ be the longest distance in the grid and $l_{de,\max} = \lceil \alpha l_{\max} \rceil$ be the maximum detour distance, where α is the detour factor and satisfies $0 \leq \alpha < 1$.

The SubSyn-Training Algorithm:

Time Complexity: The matrix multiplications are the most time consuming steps. We first discuss the complexity of matrix multiplication before summarising the time complexity of SubSyn-Training.

Sparse matrix multiplication does not apply to SubSyn-Training because M^r (e.g., M^{10}) is usually dense. As a result, the matrix multiplication step takes $O(m^3)$ for a naive algorithm. Although more efficient algorithms do exist, they are deemed unsuitable due to specific reasons. The fastest matrix multiplication algorithm currently known is the improved version of *Coppersmith-Winograd algorithm* [27], which has an asymptotic complexity of $O(m^{2.3727})$. However, this is merely a theoretical bound with no practical usage. In practice, the most feasibly efficient algorithm is the *Strassen algorithm* [25] with $O(m^{2.8})$ asymptotic complexity. This algorithm is not employed because of its reduced numerical stability (i.e., numerical rounding errors) and significantly more memory usage, which exceeds by far the amount of memory in regular commodity computers. Hence, we implemented SubSyn-Training using the naive matrix multiplication algorithm with average time complexity $O(m^3)$.

In Algorithm 1, the matrix multiplications (lines 21-26) will be executed for $l_{\max}$ times, i.e., $2(g-1) = 2\sqrt{m} - 1$ times, where $m = g^2$. Therefore, the time complexity of SubSyn-Training is $O(\sqrt{m} \times m^3) = O(m^{3.5})$.

Space Complexity: The matrices that Algorithm 1 needs to store are $A[0], A[1], \ldots A[l_{de,\max}]$, M, M_{power}, M^T, and M^T_{temp}, i.e., there are a total of $l_{de,\max} + 5$ matrices (all matrices are $m \times m = g^4$), which occupy $(l_{de,\max}+5) \cdot m^2 \times 8$ Bytes, assuming the each element in the matrices is an 8-Byte double precision floating point number. By rewriting $l_{de,\max}$ to be a function of m, we get $(l_{de,\max}+5) \cdot m^2 \times 8 = (\lceil 2\alpha(\sqrt{m}-1) \rceil + 5)m^2 \times 8$ Bytes, which is in $O(m^{2.5})$.

The SubSynE-Training Algorithm:

Time Complexity: In Algorithm 3, the fundamental instruction is still matrix multiplication. However, as discussed in Section 5, this algorithm only has matrix multiplications that involve the transition matrix M, which runs in $O(m^2)$ time. In each iteration, we perform either two matrix multiplications (lines 7-8) or three matrix multiplications (lines 10-11). Thus, the total number of matrix multiplications is at most $3l_{\max}$ times, and the time complexity is $O(l_{\max} \cdot m^2) = O(\sqrt{m} \times m^2) = O(m^{2.5})$.

Table 2 Time and Space Complexity of All the Algorithms

Complexity	Time	Space
SubSyn-Training	$O(m^{3.5})$	$O(m^{2.5})$
SubSynE-Training	$O(m^{2.5})$	$O(m^2)$
SubSynEA-Training		

Complexity	Time	Space
Baseline-Prediction	$O(\lvert\mathbb{D}\rvert)$	$O(s\lvert\mathbb{D}\rvert)$
SubSyn-Prediction	$O(m)$	$O(m^2)$
SubSynEA-Prediction		

Table 3 Space Occupation for Various Grid Granularities

Grid Granularity	40	50	60	70
SubSyn-Training	410M	1.2G	2.8G	5.9G
SubSynE-Training	78M	191M	396M	733M
SubSynEA-Training	937M	2.2G	4.6G	8.6G
Baseline-Prediction	typically 1-2G			
SubSyn-Prediction	39M	95M	198M	366M
SubSynEA-Prediction	332M	810M	1.6G	3.0G

Space Complexity: In Algorithm 3, the matrices need to be stored are M, M^T, A, B, and a temporary matrix for the swapping operation, which contain $4m$, m^2, m^2, m^2 and m^2 elements, respectively. Therefore, the total space needed is $(4m^2 + 4m) \times 8$ Bytes, which is in $O(m^2)$.

The SubSynEA-Training Algorithm:

Time Complexity: Algorithm 4 is similar to Algorithm 3. However, in each iteration, we only perform one matrix multiplication (lines 5). The input size is $O\left((\text{number of cells})^2\right)$, i.e., $O\left(m^2\right)$. Since the grid in this problem is planar, every node has at most 4 adjacent cells, which corresponds to at most 4 edges. the number of edges is at most $4m$. Thus, the total number of matrix multiplications is $l_{\max}$, and the time complexity is $O(l_{\max} \cdot (4m)^2) = O(\sqrt{m} \times m^2) = O(m^{2.5})$.

Space Complexity: In Algorithm 4, the matrices need to be stored are M, M^T, A, and a temporary matrix for the swapping operation, which contain $4 \times 4m$, $(4m)^2$, $(4m)^2$ and $(4m)^2$ elements, respectively. Therefore, the total space needed is $(48m^2 + 16m) \times 8$ Bytes, which is in $O(m^2)$.

8.2 The Prediction Algorithms

In the following analysis, we let $\mathbb{D}$ be the trajectory dataset, k be the number of predicted destinations to be obtained for each query, and s be the average number of GPS points in each trajectory.

The SubSyn-Prediction and SubSynEA-Prediction Algorithms:

Time Complexity: A major operation in both Algorithm 2 and Algorithm 5 is the loop, which visits each cell in the grid, and the time complexity is $O(m)$. Another major operation of the algorithms is the selection of top-k elements in the list. We implement this selection operation using a max-heap, and the time complexity is $O(m+k \log m)$. Therefore, both algorithms process one query in $O(m+m+k \log m) = O(m + k \log m)$ time. Since it is almost always the case that $m \gg k$ in real datasets (e.g., $m = 70^2 = 4,900$ and $k = 5$), we simplify the time complexities of both algorithms to $O(m)$.

Space Complexity: In SubSyn-Prediction, Algorithm 2 only requires the total transition matrix M^T and $P(n_d|n_s)$ for all pairs of cells, whose sizes sum up to $2m^2 \times 8$ Bytes. Thus, the space complexity of SubSyn-Prediction is $O(m^2)$. Similarly in SubSynEA-Prediction, the space occupation is $((4m)^2 + m^2) \times 8$ and the complexity is still $O(m^2)$.

The Baseline Prediction Algorithm:

Time Complexity: Similar to the SubSyn-Prediction algorithm, the baseline prediction algorithm mainly consists of a loop and a selection operation. The loop is responsible for comparing the query trajectory with the whole dataset, which runs in $O(|\mathbb{D}|)$ time. The selection operation is just the same as that of the SubSyn-Prediction algorithm which takes $O(m + k \log m)$ time. Therefore, the time complexity of the baseline prediction algorithm to process one query is $O(|\mathbb{D}| + m + k \log m)$. Applying the aforementioned inequality, we simplify the time complexity to $O(|\mathbb{D}|)$.

Space Complexity: This algorithm requires the whole dataset $\mathbb{D}$ to be stored in the memory so that it can compare the query trajectory with every trajectory in $\mathbb{D}$. Therefore, the space complexity is $O(s|\mathbb{D}|)$.

8.3 Summary

Table 2 summaries the complexity of the algorithms, and Table 3 provides an intuitive display of the space that the algorithms may consume on grids of different granularities. From these tables we can see that SubSynE-Training shows clear advantage over SubSyn-Training in terms of both time and space complexities. Although the two training algorithms SubSynE and SubSynEA have the same space complexity, the latter has a higher space occupation since it utilises the second-order Markov model. Meanwhile, SubSyn-Prediction outperforms baseline prediction algorithm in terms of both runtime efficiency and space occupation since practically $|\mathbb{D}| \gg m \gg k$ all the time. SubSynEA-Prediction has similar time and space complexities as SubSyn-Prediction, but it requires more memory space because it employs the second-order Markov model. Still, we can run SubSynEA-Prediction easily on a very fine grid (e.g., $g = 70$).

9 Experimental Study

In this section, we evaluate both the runtime efficiency and prediction accuracy of the proposed algorithms. We first present the experiment setup in Section 9.1 including the dataset used, the baseline prediction algorithm, and our experiment measurements. Then we compare the runtime efficiency of the two SubSyn training algorithms in Section 9.2. We compare the SubSyn-Prediction algorithms with the baseline prediction algorithm for both runtime efficiency in Section 9.3 and prediction accuracy in Section 9.4. Finally, we evaluate the prediction accuracy of the proposed grid partitioning strategies in Section 9.5.

9.1 Setup

Dataset: We use a real-world and large-scale taxi trajectory dataset from the *T-drive* project [30, 31] in our experiments. It contains a total of 1.9 million taxi trajectories in the city of Beijing, 16 million kilometres of distance travelled,

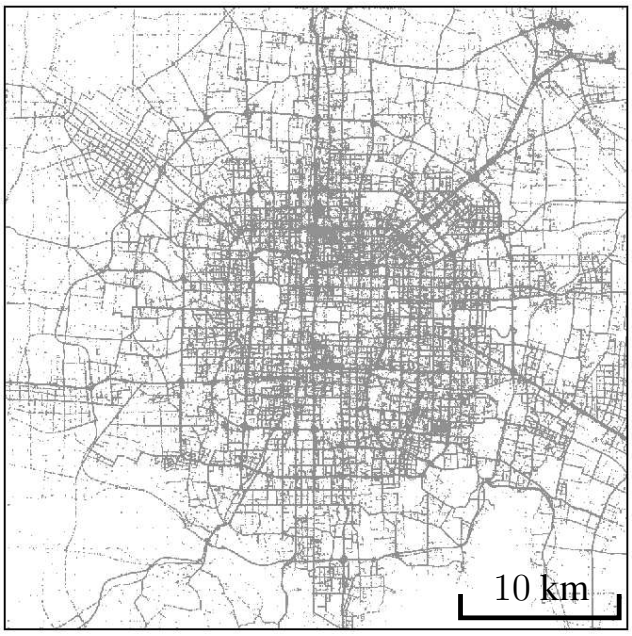


Fig. 11 A visualisation of the training dataset (65 million taxi GPS points) in Beijing within an $40km \times 40km$ region.

and 65 million GPS data points. The GPS points are plotted in Fig. 11. We randomly pick 10,000 trajectories from this dataset to be the query trajectories and the remaining trajectories are used as training data.

Baseline Prediction Algorithm: The baseline prediction algorithm described in Section 2.3 can only give predicted destinations when a query trajectory has a partial match in the training dataset. Consequently it can not be compared with our algorithm when non-matching query trajectories are present. To still produce a result in these cases, we use the current cell n_c as the predicted destination for the baseline algorithm.

Measures of Runtime Efficiency: Runtime efficiency is essential for both the training and prediction algorithms. The training algorithm needs to be run as frequently as possible to update the training dataset. The prediction algorithm needs to be evoked to answer real-time queries. For each user supplied query, it must report a list of predicted destinations instantaneously. Otherwise the whole purpose of the solution is meaningless. To be consistent with the time complexities derived in Section 8, results of runtime efficiency experiments are presented with respect to varying the grid granularity g. It is worth mentioning that this part of the experiment was programmed in Java1.7, used single thread, and run on a workstation computer with Intel Xeon-W3670 CPU (3.2GHz) and 24GB RAM.

Measures of Prediction Accuracy: To evaluate the performance of our system on various user queries, we use the following two means of measurement: *Coverage* and *Prediction Error*.

The coverage counts the number of query trajectories for which at least k suggested destinations are provided. The parameter k is determined by the number of predicted destinations that we set. For instance, when we examine top three predicted destinations, k is set to three. In other words, due to the problem of data sparsity presented, it is highly likely that insufficient predicted destinations will be suggested for certain non-matching query trajectories. Hence we utilise this property to demonstrate the difference in robustness between the baseline prediction algorithm and SubSyn.

The prediction error for a single predicted destination of a query trajectory is the ℓ_1 distance between this predicted destination and the true destination of the query trajectory. The aggregated prediction error is the average of all distance deviations across each predicted destinations of all query trajectories. It is used to indicate how far the prediction results deviate from the true destinations. It should be made clear that the prediction error does not indicate the best prediction accuracy that an algorithm can achieve. For instance, a prediction error of 2km for the top three predicted destinations is the averaged distance deviation of all of these three predicted destinations, and it is likely that the true destination is amongst these three predicted destinations. Better algorithms are the ones that have a higher coverage and a lower prediction error (i.e., lower average distance deviation).

The two aforementioned means of measurement (*Coverage* and *Prediction Error*) will be evaluated against varying four parameters one at a time: Firstly we vary the *grid granularity* g (20-70 with 10 units increment) to select a best grid granularity for our training dataset. This chosen grid granularity will be used for the remainder of the experiment. The second and third parameters are the *trip completed percentage* (10%-90% with 20% increment) and the *top-k predicted destinations* (1-5 with 1 unit increment). Finally, instead of randomly selecting query trajectories from the training dataset, we manually mix the proportion of matching and non-matching query trajectories and vary the *Match Ratio* (denoted by τ) which is the proportion of matching query trajectories in the test dataset (0-1 with 0.25 increment).

Discussion on the choice of measurements: We have chosen the average prediction error rather than the maximum prediction error in the list of top-k predicted destinations because the average prediction error is much more representative. Fig. 12 shows the distribution of the prediction errors of 10,000 query trajectories. The average prediction error is $3.4km$. We observe that the first quartile (i.e., 25th percentile) is $1.9km$, the third quartile (i.e., 75th percentile) is $4.2km$, and the prediction error hardly goes beyond $10km$ whereas the maximum is $27.4km$ with only one occurrence. Therefore we decided to use the average as it is more informative.

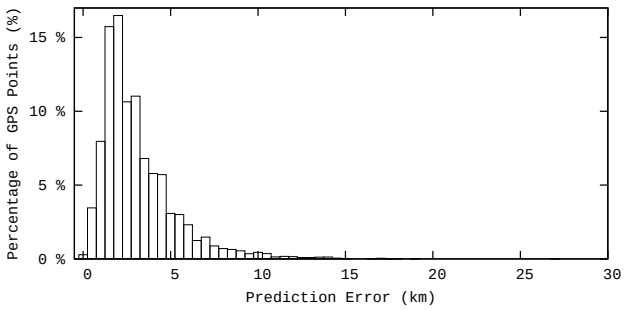


Fig. 12 Distribution of Prediction Errors ($k = 3$, $g = 70$, and the trip completed percentage $= 70\%$)

We did not employ the *Precision* or *Precision@k* as the measure for prediction accuracy due to the following reasons. Precision is defined as the proportion of correct number of predictions in all the queries. The main purposes and applications of SubSyn are to recommend sightseeing places and send targeted advertisements. For these applications, we are more interested in providing recommendations and advertisements at locations near the destination of a user: these locations need not be precisely the user's destination. Therefore, when designing the algorithms, we focus on improving the average prediction accuracy instead of precision, and we use the average distance deviation of the top-k predicted destinations as the evaluation measure for prediction accuracy.

9.2 Efficiency: Training Algorithms

Fig. 13 shows the running time of the three SubSyn training algorithms, namely SubSyn-Training, SubSynE-Training, and SubSynEA-Training, with respect to varying the grid granularity. SubSynEA-Training utilises the same algorithm as SubSynE-Training for improved runtime efficiency. It runs slightly slower due to its employment of the second-order Markov model for enhanced prediction accuracy. In the following paragraphs, we will focus on comparing SubSynE-Training with the original SubSyn-Training algorithm because they use the same mathematical model (i.e., first-order Markov model).

We can see that SubSynE-Training outperforms SubSyn-Training by orders of magnitude constantly, and the advantage grows as the grid granularity increases. This observation is in consistence with the cost analysis in Section 8, where the exponents in the time complexity (i.e., $m^{3.5}$ and $m^{2.5}$) correspond to the different *gradients* of the two linear polynomials on the log-scale figure. We fit both curves using $y = Am^B + C$, where A, B, and C are coefficients / fitting parameters, and obtain the following results. A small discrepancy in the exponent (e.g., $m^{2.6}$ rather than $m^{2.5}$) is the result of approximations in the fitting and the cost analysis.

$$\text{SubSyn-Training: } y = 9.9 \times 10^{-9} m^{3.7} + 390$$
$$\text{SubSynE-Training: } y = 2.6 \times 10^{-8} m^{2.6} - 0.7$$

Moreover, the gap in the y-axis is a result of SubSyn-Training performing much more matrix multiplication operations than SubSynE-Training as discussed in Section 4.1 and Section 5. Hence its running time is much higher. Thirdly, the graph is plotted against g instead of m, which means that the difference in the exponents is more significant: g^7 versus g^5.

Even in a medium granularity grid representation with $g = 50$, SubSyn-Training needs more than 17 hours to train a Markov model (i.e., compute the total transition matrix M^T) using the aforementioned workstation computer. In comparison, SubSynE-Training only needs 26 seconds. SubSyn-Training cannot handle grids with granularities higher than 50 due to excessive training time. Therefore, the results of the SubSyn-Training algorithm are censored for grid granularities greater than 50. When the grid granularity is set to $g = 70$, SubSynE-Training only takes 170 seconds, and SubSynEA-Training takes 550 seconds. This demonstrates the advantages of SubSynE and SubSynEA, where the cost of matrix multiplication has been significantly reduced.

The training phase may be re-run when the distribution of the data changes significantly as more data are collected. In practical applications, it takes much longer than 550 seconds for the data distribution to change significantly, so the training algorithm efficiency is not a problem. We can run the training algorithm every few minutes if needed.

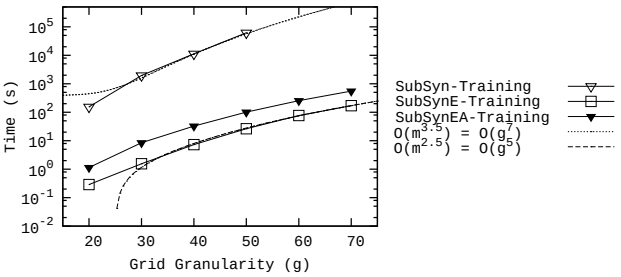


Fig. 13 Runtime Efficiency of Training Algorithms

9.3 Efficiency: Prediction Algorithms

We compare the runtime efficiency of SubSyn-Prediction, SubSynEA-Prediction, and the Baseline algorithm in terms of online query response time in Fig. 14. The vertical axis represents milliseconds per query in log scale. By taking advantage of the information obtained in the training stage, both SubSyn-Prediction and SubSynEA-Prediction require little extra computation when answering a user's query. As

Fig. 14 shows, the baseline prediction algorithm requires too much time to run, whereas the SubSyn-Prediction algorithm (and SubSynEA-Prediction) is, in most cases, at least two orders of magnitude better. SubSynEA-Prediction takes less than $100\mu s$ to answer a query. The reason is that the baseline prediction algorithm is forced to make a full sequential scan of the entire training dataset in order to compute the posterior probability, whereas the two SubSyn prediction algorithms can fetch most probability values directly from the stored total transition matrix M^T. It is worth mentioning that varying grid granularity only has marginal influence on the performance of the baseline prediction algorithm since its time complexity is $O(|\mathbb{D}|)$ (cf. Section 8), which is not correlated to the grid granularity g.

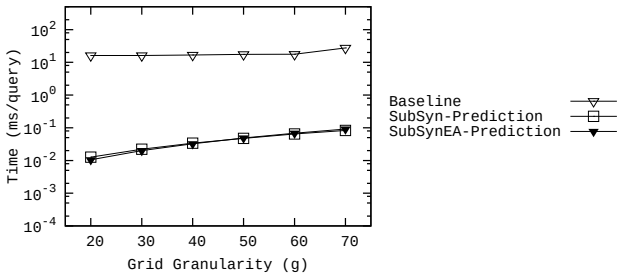
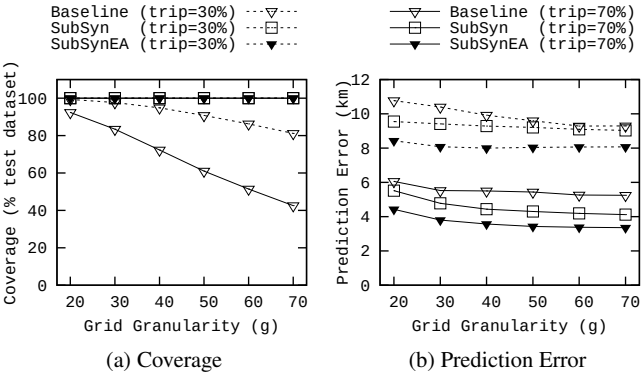


Fig. 14 Runtime Efficiency of Prediction Algorithms

9.4 Accuracy: Prediction Algorithms

As described in Section 2.3, the baseline prediction algorithm employs uniform grid partitioning. For fair comparison, we also use uniform grid partitioning for the SubSyn and SubSynEA prediction algorithms in this set of experiments. In the following experimental result figures, a convention is set that same line style (e.g., solid or dashed) represents a same group of results.

Varying the grid granularity: First of all, a suitable grid granularity needs to be decided for our training dataset. On one hand, a coarse grid (e.g., 10×10) may have a very low prediction accuracy because the area covered by each grid cell is too large. On the other hand, it has the benefit that the number of matching query trajectories is much higher since more trajectories in the training dataset may fall into identical cells, hence increasing prediction accuracy. A fine grid (e.g., 100×100) has the advantage of higher prediction accuracy that the small cell area brings, but training data become even sparser because less locations will lie in a same cell, making the task of destination prediction more difficult. Furthermore, a find grid requires much more time to train. Therefore, we need to find a balanced and compromised grid granularity g that is neither too small nor too large, and can achieve the best prediction accuracy.



(a) Coverage (b) Prediction Error

Fig. 15 Varying the grid granularity g

Fig. 15 shows the trends in both coverage and prediction error with respect to grid granularity with top-k=3. Due to limitations in memory occupation, we run the experiments with grid granularity up to $g = 70$. The coverage of the baseline prediction algorithm drops rapidly due to the data sparsity problem caused by smaller cells in a fine grid, but the drop in coverage of SubSyn-Prediction and its improved version is extremely small. The grid granularity for our training dataset is selected to be 70, for that the coverage and prediction error are both fine, these values are relatively stable, and the amount of memory occupied is within limit. In other words, a larger grid granularity will have little influence on them. In the selected setting where $g = 70$ and trip completed percentage is 70%, the coverage of the two SubSyn algorithms achieves more than twice the coverage of the baseline prediction algorithm while SubSynEA-Prediction has a more than 2km reduction in prediction error. All following experiments are done using the grid granularity $g = 70$ (cf. Fig. 16).

Varying the percentage of trip completed: Fig. 17 shows the performance in prediction accuracy versus the percentage of trip completed for both top-k values 1 and 3. For the baseline prediction algorithm, the amount of query trajectories for which sufficient predicted destinations are provided decreases as the length of the trip increases due to the fact that longer query trajectories (i.e., higher trip completed percentage) are less likely to have a partial match in the training dataset. Specifically, when trip completed percentage increases towards 90%, the coverage of the baseline prediction algorithm decreases to almost 20% for top-$k = 3$. Our two SubSyn prediction algorithms successfully coped with it as expected with only an unnoticeable drop in coverage, and can constantly answer almost 100% of queries. It proves that the baseline prediction algorithm cannot handle (relatively) long trajectories since the chances of finding a matching trajectory decrease when the length of a query trajectory grows. The coverage performance of the baseline prediction algorithm when top-$k = 3$ is even worse then that of top-$k = 1$

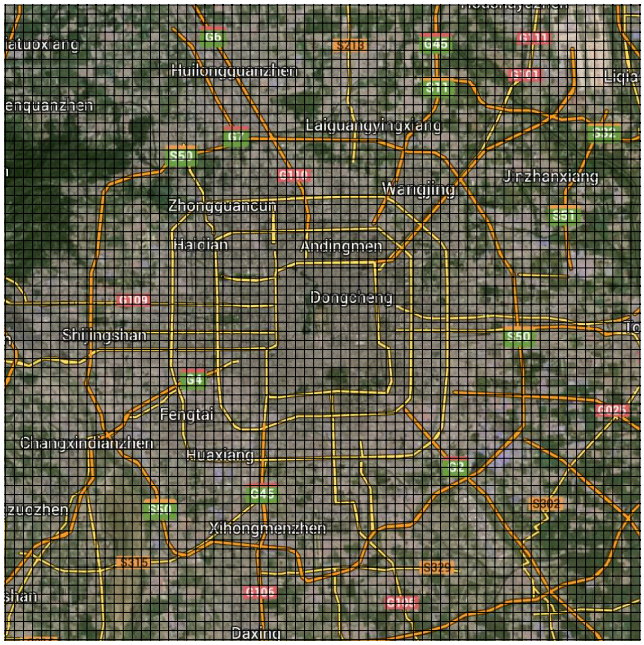
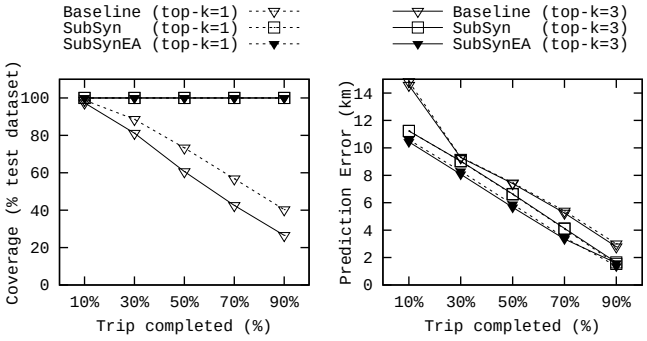


Fig. 16 Map of Beijing ($40km \times 40km$ region) with a 70×70 uniform grid overlay. Each cell is roughly a $570m \times 570m$ square.

because the metric *coverage* counts the number of query trajectories that gives k predicted destinations.

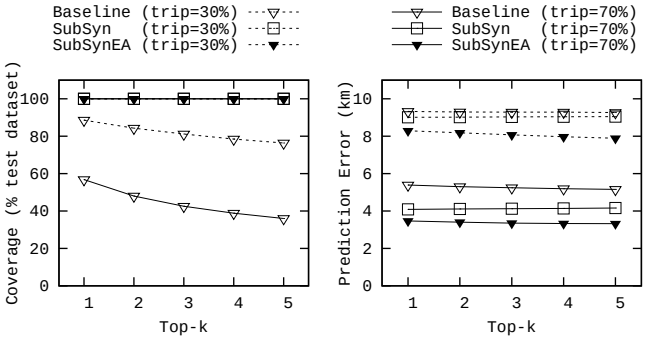


(a) Coverage (b) Prediction Error

Fig. 17 Varying the percentage of trip completed

Apart from the huge advantages of the two SubSyn prediction algorithms in coverage, their prediction errors are comparable with that of the baseline prediction algorithm. For the baseline prediction algorithm, despite the negative influence of the coverage problem, its prediction error reduces as the trip completed percentage increases for a simple reason. When the baseline prediction algorithm fails to find adequate predicted destinations, we use the current cell in the query trajectory as the predicted destination. Because higher trip completed percentage yields a closer distance between the current cell and the actual destination, the prediction error reduces accordingly. For SubSyn and SubSynEA, closer to the true destination means that there are fewer potential destinations and intuitively the prediction errors reduce. It is observed that SubSynEA-Prediction outperforms SubSyn-Prediction, which in turn outperforms the baseline prediction algorithm throughout the progress of a trip.

Varying the number of predicted destinations: We also investigate the effect of the number of predicted destinations on the performance of both algorithms by examining the top-k (from 1 to 5) predicted destinations. We are interested in this metric since it reveals more vulnerability of the baseline prediction algorithm in that although it can make prediction for matching trajectories, the number of predicted destinations may still be insufficient (e.g., only one). Therefore in such circumstances where insufficient predicted destinations are returned, we consider them unsatisfactory in the coverage test. The experimental results are shown in



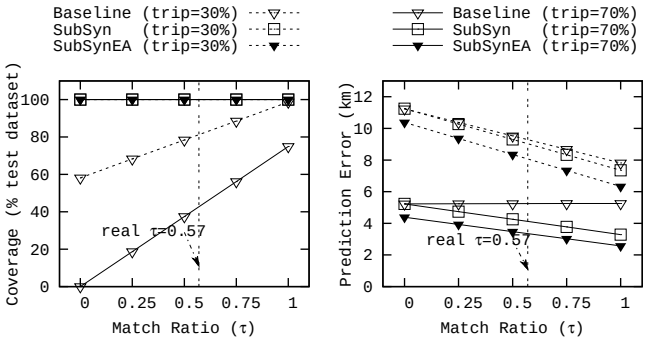
(a) Coverage (b) Prediction Error

Fig. 18 Varying the value of k

Fig.18. In this figure, the comparative performances of the three prediction algorithms are similar to that of the experiment of varying the percentage of trip completed. Specifically, observations that can be made from the figure are as follows. The two SubSyn algorithms show a more stable coverage and a more accurate prediction accuracy than the baseline prediction algorithm. For the baseline prediction algorithm, the number of query trajectories which have sufficient suggestions (i.e., the coverage) drops due to the data sparsity problem since, for certain query trajectories, it cannot find adequate (i.e., no less than k) predicted destinations. The same problem affects neither SubSyn-Prediction nore SubSynEA-Prediction, and they remain an almost 100% suggestion offer rate. In a common setting when $k = 3$, the coverage of SubSyn (and SubSynEA) is more than twice the coverage of the baseline prediction algorithm.

Varying top-k has little correlation with the prediction error because we compute the prediction error (i.e., average distance deviation) by averaging amongst all predicted destinations.

Varying the ratio of matching and non-matching query trajectories: The query trajectories used in the above experiments are drawn randomly from the training dataset. They reflect the real distribution of matching and non-matching query trajectories in both the test dataset and the training dataset. It is found that the real match ratio (denoted by τ) decreases while the grid granularity g increases because finer grid yields sparser data. For a 70×70 grid, the averaged real match ratio is found to be approximately 0.57 (indicated by the vertical dashed line in Fig. 19). It indicates that, in average, 57% of query trajectories will be able to find a partial match in the training dataset. Such a match ratio seems to be helpful to the baseline prediction algorithm. In this experiment, we elaborate further on the concept of match ratio by manually selecting a mixture of matching and non-matching query trajectories, and comparing the influence of different match ratios. For simplicity while maintaining an indicative results, a trajectory is said to have a partial match if the first 70% cells have an exact match in the training dataset. This indicates that, when $\tau = 0.57$ and the trip completed percent is higher than 70%, the coverage is at most 57% for the baseline prediction algorithm.



(a) Coverage (b) Prediction Error

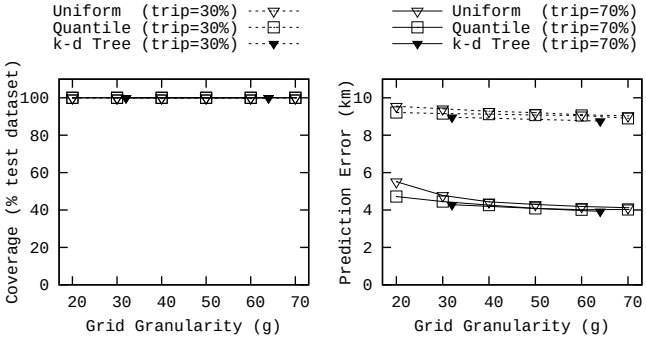
Fig. 19 Varying the match ratio τ

As shown in Fig. 19, by varying the match ratio τ, the performance of the baseline prediction algorithm deteriorates rapidly when τ is tuned towards zero while little impact is observed for either SubSyn-Prediction or SubSynEA-Prediction. The baseline prediction algorithm functions well provided that abundant data are given (i.e., $\tau \rightarrow 1$), but the performance starts to decrease to an unacceptable status when there are insufficient training data. Particularly, when the match ratio is low (i.e., $\tau \rightarrow 0$) and the trip completed percentage is high (e.g., 70%), the baseline prediction algorithm has a coverage towards 0%. From Fig. 19a, our two SubSyn prediction algorithms provides adequate (i.e., at least three since the default value of top-k is 3) predicted destinations for almost every query trajectory. It proves that our algorithms can overcome the data sparsity problem while maintaining a stable performance, whereas the baseline prediction algorithm is unable to achieve this objective.

In Fig. 19b, it is observed that the prediction errors of all algorithms drop when more relevant training data are available (i.e., $\tau \rightarrow 1$). Once again, it proves that the prediction accuracy of SubSynEA-Prediction is the highest, and SubSyn-Prediction leads that of the baseline prediction algorithm.

9.5 Accuracy: Grid Partitioning Strategies

In this experiment, we run the SubSyn prediction algorithm with different grid partitioning strategies including uniform, quantile, and k-d tree based grid partitioning strategies, and their performance of prediction accuracy are compared against each other. Both uniform grid and quantile grid are plotted for grid granularity $g \in \{20, 30, 40, 50, 60, 70\}$, and k-d tree grid is plotted with $g \in \{32, 64\}$ due to its constraints in partitioning algorithm.



(a) Coverage (b) Prediction Error

Fig. 20 Prediction Accuracy of Grid Partitioning Strategies

As shown in Fig. 20, the coverages are identical for all the three partitioning strategies because they use the same prediction algorithm. Thus, we focus on the different performance in prediction error. We observe that the two new grid partitioning strategies (quantile and k-d tree based) outperform the uniform grid partitioning strategy constantly in terms of prediction accuracy. Moreover, the advantage is more significant when the grid granularity is coarser. This can be explained by the fact that more balancing grids (quantile and k-d tree based) represent the data distribution better than the uniform grid does, and that the difference in the variance σ^2 is more significant when the grid granularity is coarser. The experimental results show that a more balancing grid partitioning strategy is particularly effective when computing power or the available data is limited. Actually such circumstances are quite common for small businesses or for medium sized cities. In these cases, only a coarse grid

will provide meaningful results, and having a more balancing grid partitioning strategy (such as the k-d tree based partitioning) is important.

10 Conclusion

In this paper, we have identified the data sparsity problem in destination prediction and proposed a novel method named *Sub-Trajectory Synthesis* (SubSyn) to address this problem. SubSyn decomposes each trajectory in the historical dataset into smaller segments (i.e., sub-trajectories) and combines them to generate "synthesised" trajectories. The underlying process is formulated by the Markov model and the Bayesian inference framework. We proposed an algorithm to improve the efficiency of the training phase substantially. We also investigated the use of the second-order Markov model to further boost prediction accuracy.

Experiments based on real datasets have shown that the performances of the SubSyn algorithms (i.e., SubSyn, SubSynE, and SubSynEA) exceed that of the baseline algorithm in terms of both runtime efficiency and prediction accuracy. The two SubSyn-Prediction algorithms (i.e., SubSyn and SubSynEA) can predict destinations for up to ten times more query trajectories than the baseline prediction algorithm. At the same time, SubSynEA-Prediction maintains a competitive prediction accuracy. In terms of runtime efficiency, the SubSynEA-Training algorithm requires only a few minutes to run for a fine granularity of grid partitioning, and SubSynEA-Prediction runs over two orders of magnitude faster than the Baseline-Prediction algorithm.

Acknowledgements This work is supported by *Australian Research Council (ARC) Discovery Project* DP130104587 and *Australian Research Council (ARC) Future Fellowships Project* FT120100832.

References

1. Ali ME, Zhang R, Tanin E, Kulik L (2008) A motion-aware approach to continuous retrieval of 3d objects. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, IEEE, pp 843–852
2. Ali ME, Tanin E, Zhang R, Kulik L (2010) A motion-aware approach for efficient evaluation of continuous queries on 3d object databases. The VLDB JournalThe International Journal on Very Large Data Bases 19(5):603–632
3. Alvarez-Garcia JA, Ortega JA, Gonzalez-Abril L, Velasco F (2010) Trip destination prediction based on past GPS log using a hidden markov model. Expert Systems with Applications: An International Journal 37:8166–8171
4. Alvarez-Lozano J, García-Macías JA, Chávez E (2013) Learning and user adaptation in location forecasting. In: Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication, UbiComp '13 Adjunct, pp 461–470
5. Ashbrook D, Starner T (2003) Using GPS to learn significant locations and predict movement across multiple users. Personal Ubiquitous Computing 7:275–286
6. Bhattacharya A, Das SK (1999) Lezi-update: An information-theoretic approach to track mobile users in pcs networks. In: MobiCom, pp 1–12
7. Chen L, Lv M, Chen G (2010) A system for destination and future route prediction based on trajectory mining. Pervasive and Mobile Computing 6:657–676
8. Gogate V, Dechter R, Bidyuk B (2005) Modeling transportation routines using hybrid dynamic mixed networks. In: UAI, pp 217–224
9. GPSExchange (2012) GPS track log route exchange forum. URL `http://www.gpsexchange.com/`
10. Hashem T, Kulik L, Zhang R (2010) Privacy preserving group nearest neighbor queries. In: EDBT, pp 489–500
11. Hashem T, Kulik L, Zhang R (2013) Countering overlapping rectangle privacy attack for moving knn queries. Information Systems 38(3):430–453
12. Horvitz E, Krumm J (2012) Some help on the way: opportunistic routing under uncertainty. In: UbiComp, pp 371–380
13. Jensen CS, Lin D, Ooi BC, Zhang R (2006) Effective density queries on continuouslymoving objects. In: Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on, IEEE, pp 71–71
14. Jensen J (1906) Sur les fonctions convexes et les ingalits entre les valeurs moyennes. Acta Mathematica 30(1):175–193, DOI 10.1007/BF02418571, URL `http://dx.doi.org/10.1007/BF02418571`
15. Krumm J, Horvitz E (2006) Predestination: Inferring destinations from partial trajectories. In: UbiComp, pp 243–260
16. Krumm J, Horvitz E (2007) Predestination: Where do you want to go today? IEEE Computer 40(4):105–107
17. Liao L, Patterson DJ, Fox D, Kautz H (2007) Learning and inferring transportation routines. Artificial Intelligence 171(5-6):311–331
18. Marmasse N, Schmandt C (2002) A user-centered location model. Personal and Ubiquitous Computing 6:318–321
19. Microsoft Research (2012) T-drive trajectory data sample. URL `http://research.microsoft.com/apps/pubs/?id=152883`
20. Nutanong S, Zhang R, Tanin E, Kulik L (2010) Analysis and evaluation of V*-*k*nn: an efficient algorithm for moving *k*nn queries. VLDB J 19(3):307–332
21. Patterson DJ, Liao L, Fox D, Kautz H (2003) Inferring high-level behavior from low-level sensors. In: UbiComp, pp 73–89
22. Qiu D, Papotti P, Blanco L (2013) Future locations prediction with uncertain data. In: Machine Learning and Knowledge Discovery in Databases, vol 8188, Springer Berlin Heidelberg, pp 417–432
23. ShareMyRoute (2012) Share my route. URL `http://www.sharemyroutes.com`
24. Simmons R, Browning B, Zhang Y, Sadekar V (2006) Learning to predict driver route and destination intent. In: ITSC, pp 127–132
25. Strassen V (1969) Gaussian elimination is not optimal. Numerische Mathematik 13(4):354–356
26. Tiesyte D, Jensen CS (2008) Similarity-based prediction of travel times for vehicles traveling on known routes. In: GIS, pp 14:1–14:10
27. Williams VV (2011) Breaking the coppersmith-winograd barrier. unpublished manuscript
28. Xue AY, Zhang R, Zheng Y, Xie X, Huang J, Xu Z (2013) Destination prediction by sub-trajectory synthesis and privacy protection against such prediction. In: ICDE
29. Xue AY, Zhang R, Zheng Y, Xie X, Yu J, Tang Y (2013) Desteller: A system for destination prediction based on trajectories with privacy protection. In: International Conference on Very Large Data Bases (VLDB)
30. Yuan J, Zheng Y, Zhang C, Xie W, Xie X, Sun G, Huang Y (2010) T-drive: Driving directions based on taxi trajectories. In: GIS, pp

99–108
31. Yuan J, Zheng Y, Xie X, Sun G (2011) Driving with knowledge from the physical world. In: KDD, pp 316–324
32. Zhang R, Qi J, Lin D, Wang W, Wong RCW (2012) A highly optimized algorithm for continuous intersection join queries over moving objects. The VLDB JournalThe International Journal on Very Large Data Bases 21(4):561–586
33. Zheng Y, Zhou X (eds) (2011) Computing with Spatial Trajectories. Springer
34. Ziebart BD, Maas AL, Dey AK, Bagnell JA (2008) Navigate like a cabbie: Probabilistic reasoning from observed context-aware behavior. In: UbiComp, pp 322–331