Finding Time Period-Based Most Frequent Path in Big Trajectory Data

Wuman Luo, Haoyu Tan, Lei Chen, Lionel M. Ni
Department of Computer Science and Engineering
Guangzhou HKUST Fok Ying Tung Research Institute
Hong Kong University of Science and Technology, Hong Kong SAR, China
{luowuman, hytan, leichen, ni}@cse.ust.hk

ABSTRACT
The rise of GPS-equipped mobile devices has led to the emergence of big trajectory data. In this paper, we study a new path finding query which finds the most frequent path (MFP) during user-specified time periods in large-scale historical trajectory data. We refer to this query as time period-based MFP (TPMFP). Specifically, given a time period $T$, a source $v_s$ and a destination $v_d$, TPMFP searches the MFP from $v_s$ to $v_d$ during $T$. Though there exist several proposals on defining MFP, they only consider a fixed time period. Most importantly, we find that none of them can well reflect people’s common sense notion which can be described by three key properties, namely suffix-optimal (i.e., any suffix of an MFP is also an MFP), length-insensitive (i.e., MFP should not favor shorter or longer paths), and bottleneck-free (i.e., MFP should not contain infrequent edges). The TPMFP with the above properties will reveal not only common routing preferences of the past travelers, but also take the time effectiveness into consideration. Therefore, our first task is to give a TPMFP definition that satisfies the above three properties. Then, given the comprehensive TPMFP definition, our next task is to find TPMFP over huge amount of trajectory data efficiently. Particularly, we propose efficient search algorithms together with novel indexes to speed up the processing of TPMFP. To demonstrate both the effectiveness and the efficiency of our approach, we conduct extensive experiments using a real dataset containing over 11 million trajectories.

Categories and Subject Descriptors
H.2.8 [Database Applications]: Spatial databases and GIS

Keywords
path finding, big trajectory data

1. INTRODUCTION
In recent years, due to the continuing improvements in location-acquisition technology (e.g., GPS), large amounts of historical trajectory data have become available for emerging applications including urban planning [32], spatio-temporal data mining [17], and various location-based services [29, 31]. Path finding, also known as finding the most desirable path, plays a crucial role in these applications. Traditionally, the desirability of a path is usually measured by travel distance or time. In this paper, we study time period-based most frequent path (TPMFP), a new path finding query which aims at finding the most frequent path of a certain time period.

The objective of TPMFP is to reflect the common routing preference of the past travelers, which is very useful in many real-world applications. In map services and vehicle navigation systems, TPMFP provides users with extra routing options other than the shortest/fastest path. For example, when traveling in an unfamiliar city, people tend to follow the most common path for several considerations such as making sure that the path is not blocked by a recent road work, reducing the risks of getting lost, and avoiding unpaved roads and dangerous shortcuts. In these situations, a TPMFP such as ‘the MFP from my hotel to the airport during the last week’ is a better recommendation than the shortest/fastest path. Another application is trajectory data mining where the TPMFP query can be used as a critical subroutine for high-level data analysis. For example, we can find out ‘whether people’s travel habits during weekends and workdays in the last year are different’ by issuing two TPMFP queries (with segmented time periods). It is also possible to detect important events by mining the changes of MFP during different time periods (see Section 6).

Despite that the meaning of MFP is easy to understand, it is nontrivial to give a satisfactory definition to well reflect people’s common sense notion. We will illustrate this by a concrete example. Figure 1 shows a road network along with 44 trajectories. The road network is represented by a graph with each vertex being a road intersection and each edge being a road segment. The trajectories are divided into groups according to whether they traverse the same path. As depicted by dashed curves, there are 6 trajectory groups ($G_1$ through $G_6$), containing 8, 6, 4, 5, 1, 20 trajectories, respectively. For the ease of presentation, we use $v_1$–$v_3$ path/MFP to denote a path/MFP from $v_1$ to $v_3$. In this example, we want to find the $v_1$–$v_{12}$ MFP, assuming that all the trajectories are within the given time period.

A straightforward definition of MFP is to count the number of the trajectories going through each $v_1$–$v_{12}$ path and select the one with the highest support. In this case, there are four $v_1$–$v_{12}$ paths with non-zero supports, i.e., the paths traversed by $G_1$, $G_2$, $G_4$, $G_5$ whose supports are 8, 6, 5, 1, respectively. Hence, the $v_1$–$v_{12}$ MFP under this definition is $v_1 \rightarrow v_2 \rightarrow v_{12}$. Given the $v_1$–$v_{12}$ MFP, it is natural to infer that the $v_2$–$v_{12}$ MFP should be $v_2 \rightarrow v_{12}$, i.e., the suffix of the $v_1$–$v_{12}$ MFP starting from $v_2$. However, since the support of path $v_2 \rightarrow v_{12}$ is 8 and the support of path $v_2 \rightarrow v_3 \rightarrow v_{12}$ is 6+4 = 10, the resulting $v_2$–$v_{12}$ MFP is the
In this paper, we study the problem of TPMFP in big trajectory data. We assume that each trajectory in the dataset has been aligned to the corresponding road network and represents a meaningful trip in the past. To overcome the shortcomings of previous definitions, we use a sequence instead of a single scalar to describe the frequency of a path. We then define a total-order relation to compare path frequencies. To answer TPMFP in a very large dataset efficiently, we design efficient algorithms as well as special index structures. The main contributions of this paper are as follows:

- We propose a time period-based definition of MFP and demonstrate that it is suffix-optimal, length-insensitive, and bottleneck-free (Section 2).
- We propose a two-step framework to solve the TPMFP problem in the context of very large trajectory datasets (Section 3).
- We propose efficient footmark graph construction algorithms along with two novel indexes to significantly reduce the number of disk I/O operations (Section 4).
- We propose efficient algorithms for searching TPMFP on a footmark graph based on the more-frequent-than relation (Section 5).
- We conduct extensive experiments using a real dataset containing over 11 million trajectories to evaluate the performance of our algorithms and indexes. The results show that our approaches are both effective and efficient (Section 6).

We present the related works in Section 7 and conclude the paper in Section 8.

## 2. THE TPMFP PROBLEM

The problems of the existing definitions and the necessity of allowing user-specified time periods motivate us to seek a novel way to define TPMFP. In particular, we require the definition to satisfy the following three key properties to avoid the aforementioned drawbacks.

**PROPERTY 1 (SUFFIX-OPTIMAL).** Let \( P^* \) denote the \( v_s \rightarrow v_d \) MFP. For any vertex \( u \in P^* \), the sub-path (suffix) of \( P^* \) from \( u \) to \( v_d \) should be the \( u \rightarrow v_d \) MFP.

**PROPERTY 2 (LENGTH-INSENSITIVE).** The length of any path should not be a deciding factor of whether it is the \( v_s \rightarrow v_d \) MFP.

**PROPERTY 3 (BOTTLENECK-FREE).** The MFP \( P^* \) should not contain infrequent edges (i.e., bottlenecks).

In the rest of this section, we present formal definitions for the TPMFP problem and explain why our TPMFP definition satisfies the above properties.

### 2.1 Sketch of the Problem Definition

Our basic idea is to use a sequence instead of a scalar to describe path frequency. We first construct a weighted sub-graph of the road network called footmark graph for a given destination \( v_d \) and a given time period \( T \). In a footmark graph, the weight of an edge \((u, v)\) is defined as the number of trajectories going through \((u, v)\)
We use $P.s, P.d,$ and $P[i]$ to denote the source vertex $x_1$, the ending vertex $x_k$, and the $i$th vertex $x_i$, respectively. In addition, $P$ is often represented in the form of $x_1 \rightarrow x_2, \ldots, \rightarrow x_k$.

**Definition 3 (Trajectory)**. Given $G$, a trajectory $Y$ is a sequence $\{(x_1, t_1), (x_2, t_2), \ldots, (x_k, t_k)\}$ such that there exists a path $x_1 \rightarrow x_2, \ldots, \rightarrow x_k$ on $G$ and $t_i$ is a timestamp indicating the time when $Y$ passes $x_i$.

Similar to path, we use $Y.s, Y.d, Y[i], v$ and $Y[i], t$ to denote the starting vertex $x_1$, the ending vertex $x_k$, the $i$th vertex $x_i$, and the $i$th timestamp $t_i$, respectively. Besides, we use $Y.t_1, Y.t_e, Y.P$ to denote the starting time $t_1$, the ending time $t_e$, and the corresponding path, respectively.

Let $\Omega = (G, \Upsilon, v_{id}, v_{od}, T)$ denote the input for the TPMFP problem, where $G$ is a road network, $\Upsilon$ is a set of historical trajectories, $v_{id}, v_{od} \in V(G)$ are two vertices indicating the source and destination, respectively, and $T$ is a time period. Without loss of generality, we assume that $T$ is a continuous time period $[t_s, t_e]$, where $t_s$ is the starting time and $t_e$ is the ending time.

In the presence of $v_{id}, v_{od}$, and $T$, it is obvious that we should only consider the trajectories that can potentially contribute to calculating TPMFP. In specific, we focus on the trajectories that have gone through $v_{id}$ during $T$. Note that we do not factor out the trajectories that have not gone through $v_{id}$ because they can help calculating the parts of the TPMFP starting from intermediate vertices. For example, the trajectory group $G_3$ (Figure 1) contributes to the frequency of the sub-path from $v_2$ to $v_3$ in searching the $v_1 \rightarrow v_2$ TPMFP. Besides, for any trajectory reaching $v_{od}$, the sub-trajectory $v_{id} \rightarrow v_{od}$ can be safely discarded because after that point it does not go towards the destination $v_{od}$ any more (assuming that each trajectory does not traverse a vertex more than once). We summarize the above discussion by presenting the formal definition of footprint.

**Definition 4 (Footmark)**. Given $\Omega = (G, \Upsilon, v_{id}, v_{od}, T)$ and a trajectory $Y = \{(x_1, t_1), \ldots, (x_k, t_k)\} \in \Upsilon$, if there exists a non-empty sub-trajectory $Y’$ of $Y$ from $Y[i]$ to $Y[j]$ such that:

- $Y’.d = v_{id}, i.e., Y[j], v = v_{id}$,
- $[Y’.t_s, Y’.t_e] \subseteq T$, i.e., $[Y[i], t, Y[j], t] \subseteq T$,
- $Y[i-1].t \notin T$, if $i > 1$,

then path $Y’.P$ is the footmark of $Y$ w.r.t. $v_{id}$ and $T$, denoted as $\hat{Y}(v_{id}, T)$.

In the following, we abbreviate $\hat{Y}(v_{id}, T)$ as $\hat{Y}$ if it is clear from the context. Note that $\hat{Y}$ does not exist if $Y$ has not gone through $v_{id}$ during $T$. To better understand the definition, a concrete example is given in Figure 2. The input is $\Omega = (G, \Upsilon, v_{id}, v_{od}, [t_s, t_e])$, where $G$ is shown in Figure 2(a) and $\Upsilon$ contains 6 trajectories. Figure 2(b) illustrates how to derive the footmarks from the trajectories in $\Upsilon$. In general, when $\hat{Y}$ exists, it is obtained by removing from $Y$ all the prefix elements whose timestamps are not within $T$ and all the suffix elements after $v_{id}$. Here, the resulting footmarks are:

- $\hat{Y}_1: v_2 \rightarrow v_7 \rightarrow v_8$,
- $\hat{Y}_2: v_1 \rightarrow v_2 \rightarrow v_7 \rightarrow v_8$,
- $\hat{Y}_3: v_1 \rightarrow v_2 \rightarrow v_6 \rightarrow v_8$,
- $\hat{Y}_4 = \hat{Y}_5: v_2 \rightarrow v_6 \rightarrow v_8$.

Note that $\hat{Y}_6$ does not exist since it does not traverse $v_8$. In the following, we use $\hat{Y}(v_{id}, T)$ (or $\hat{Y}$ for short) to denote a footmark set of $\Upsilon$.
Figure 2: An example of footmark graph for input Ω = (G, Y, v_s, v_d, T), where Y = {Y_1, ..., Y_6}, v_s = v_1, v_d = v_8, and T = [t_s, t_e]

**Definition 5 (Edge Frequency).** Given G, Y_{(v_s, T)}, and an edge (u, v) ∈ G, the edge frequency F(u, v) is the number of the footmarks in Y_{(v_s, T)} containing (u, v).

In Figure 2(a), for example, F(v_1, v_2) = 2 because it is contained by two footmarks, i.e., Y_2 and Y_3.

**Definition 6 (Footmark Graph).** Given G and Y_{(v_s, T)}, a footmark graph G_f is a weighted sub-graph of G such that:
- for any edge (u, v) ∈ G, w_{uv} = F(u, v);
- edge (u, v) ∈ G_f, if and only if (u, v) ∈ G and w_{uv} > 0.

Figure 2(c) shows the footmark graph for the previous example. Figure 3 illustrates a footmark graph derived from the real dataset used in our experiments. The destination is located at the center of the plot region and the edge frequency is described by the line width where LineWidth(u, v) ~ O(log F(u, v)).

Figure 3: A footmark graph derived from a real dataset

Now we define path frequency in terms of edge frequency. The traditional approach is to use an aggregate function such as sum and product to compute a scalar-valued score for each path from the weights of the related edges. The problem is that these functions are not robust such that the resulting MFP tends to be length-sensitive and may contain infrequent edges. To address this issue, we propose a new approach to define path frequency.

**Definition 7 (Path Frequency).** Given G_f, the frequency of path P (to v_d) is a sequence F(P) = (f_1, ..., f_k) where:
- \{f_i| i \in \{1, ..., k\}\} = \{w_{uv}| (u, v) \in E(P)\},
- f_1 ≤ f_2 ≤ ⋯ ≤ f_k.

In other words, F(P) is a non-decreasing sequence of the weights of all the edges in P. For example, the frequency of path v_1 → v_2 → v_6 → v_8 in Figure 2(c) is (2, 3, 3) and the frequency of the other v_1-v_8 path is (2, 2, 2).

To define MFP, we further establish a ranking system for sequence-valued path frequencies by defining the more-frequent-than relation.

**Definition 8 (More-Frequent-Than Relation).** Given two path frequencies F(P) = (f_1, ..., f_m) and F(P') = (f'_1, ..., f'_n) w.r.t. the same G_f, F(P) is more-frequent-than F(P'), denoted as F(P) ⪰ F(P'), if one of the following statements holds:
- F(P) is a prefix of F(P');
- there exists a q ∈ {1, ..., min(m, n)} such that 1) f_i = f'_i for all i ∈ {1, ..., q - 1}, if q > 1, and 2) f'_q > f_q.

Particularly, F(P) is strictly-more-frequent-than F(P'), denoted as F(P) ⪰ F(P'), if F(P) ⪰ F(P') and F(P) ≠ F(P').

According to the definition, we have (10, 10, 10) ⪰ (1, 100), (1, 2) ⪰ (1, 2, 3), and (5, 6, 9) ⪰ (5, 6, 7, 12), to name a few instances. Essentially, the more-frequent-than relation is a special form of the lexicographic order, which directly leads to the following theorem.

**Theorem 1.** The more-frequent-than relation is a total order.

The above theorem implies that we can safely define the MFP as the path with the highest ranked frequency.

**Definition 9 (MFP).** Given G_f and a v_s-v_d path P^* ⊆ G_f, if F(P^*) ≥ F(P) holds for every v_s-v_d path P ⊆ G_f, then P^* is the v_s-v_d MFP w.r.t. G_f.

Finally, we can define the TPMFP problem based on the above definitions.

**Problem Statement:** Given Ω = (G, Y, v_s, v_d, T) where Y is a very large set of historical trajectories, we need to find the TPMFP which is the MFP w.r.t. G_f. Note that G_f is the footmark graph derived from Ω.

**2.3 Properties**

Now we explain why the TPMFP definition satisfies the three key properties. We begin with a theorem which demonstrates its suffix-optimal property (due to the page limitation, we omit the lengthy proof).
Algorithm 1: Two major steps for the TPMFP query

**Input:** \( \Omega = (G, \Upsilon, v_s, v_d, T) \)

**Output:** the TPMFP w.r.t. \( \Omega \)

\[
\begin{align*}
1 & \text{ step 1: build the footmark graph } G_f \text{ w.r.t. } \Omega ; \\
2 & \text{ step 2: find the MFP } P^* \text{ from } v_s \text{ to } v_d \text{ on } G_f ; \\
3 & \text{return } P^* ;
\end{align*}
\]

**Theorem 2.** Given \( \Omega = (G, \Upsilon, v_s, v_d, T) \), let \( P^* \) be the \( v_s - v_d \) TPMFP w.r.t. \( \Omega \). Then, for every vertex \( u \in \mathcal{V}(P) \), the sub-path of \( P^* \) from \( u \) to \( v_d \) is the \( u - v_d \) TPMFP.

The implications of the above theorem are two-fold. First, it ensures the stability of the TPMFP in the sense that the query result from any intermediate location to the destination will not deviate from the initial route, which is desirable for practical applications. Second, it guarantees the existence of efficient (polynomial-time) algorithms to compute the exact result of TPMFP, which is important for time-critical services.

In addition, the TPMFP is length-insensitive and bottleneck-free as well. This is a direct result of the definition of path frequency and the comparison rule. Specifically, given two paths \( P \) and \( P' \), \( P \) is more frequent than \( P' \) as long as \( F(P) \) wins \( F(P') \) in the comparison between their first different components. In other words, the path length is not a deciding factor in such comparisons, which is in line with the common sense that the road segments in a frequent path can be either few or many. For the bottleneck-free property, it is straightforward to prove that if \( P \) is the TPMFP, then for any edge \( (u, v) \in E \), there does not exist any other path \( P' \) such that all edge frequencies of \( P' \) are greater than the frequency of \( (u, v) \). It implies that the TPMFP does not contain any (infrequent) edge that can be avoided by introducing more frequent ones only.

3. **Solution Overview**

Given input \( \Omega = (G, \Upsilon, v_s, v_d, T) \), we perform the TPMFP query in two major steps, as illustrated in Algorithm 1.

For the first step, since the edge weights on a footmark graph depend on the input time period \( T \), we cannot calculate them in an offline manner. This fact poses great challenges on building the footmark graph, especially in the presence of huge amounts of trajectories. Assume that the data are stored in a disk-based key-value storage system and a trajectory can be fetched based on its id. A straightforward approach to build a footmark graph is to scan all the trajectories in \( \Upsilon \) and update the footmark graph \( G_f \) during the process. Initially, \( G_f \) is an empty graph. For each trajectory \( \Upsilon \) being scanned, we first check whether it generates a footmark \( \tilde{\Upsilon} \). If so, we update the footmark graph by incrementing the weight of each edge in \( E(\tilde{\Upsilon}) \) by 1 (if the edge does not exist in \( G_f \), we simply add it to \( G_f \) and set its weight to 1). The construction of \( G_f \) is completed once all trajectories are processed. However, this approach is inefficient from the perspectives of both I/O and computation: the number of trajectories need to be fetched is \( |\Upsilon| \times l \) and the computation complexity for edge weight update is \( O(|\Upsilon| \times l) \), where \( l \) is the average length of footmarks. To speed up I/O, we propose Footmark Index (FMI) to effectively filter out the trajectories having no footmarks. We further devise Container-Based Footmark Index (CFMI), an improved version of FMI, to further reduce the random page accesses by only fetching the ‘dominant’ trajectories.

To speed up computation, we develop efficient algorithms based on the proposed indexes which dramatically reduces the cost of updating \( G_f \). We will discuss the footmark graph construction in detail in Section 4.

The second step is described in Section 5. We first prove that searching MFP on a given footmark graph can be solved by the dynamic programming approach. Then we propose a new variant of the classical Bellman–Ford algorithm to deal with the sequence-valued path frequencies. We also demonstrate the correctness of the algorithm and analyze its time and space complexities.

4. **Footmark Graph Construction**

The main task of footmark graph construction is to calculate the footmark set \( \tilde{\Upsilon} \) w.r.t. \( G, \Upsilon, v_s, v_d, \) and \( T \). As mentioned before, it is inefficient to examine all the trajectories because only a small portion of them have footmarks for the specific \( v_d \) and \( T \). In addition, the large number of weight updating operations can lead to high computation cost. As such, we propose two special indexes to reduce disk reads together with efficient algorithms to construct \( G_f \). Note that we assume that all the trajectories are stored on disk and each trajectory can be fetched using one random page read.

4.1 **Footmark Index**

To calculate \( \tilde{\Upsilon} \), we need to find the trajectories in \( \Upsilon \) that have footmarks for the given \( v_d \) and \( T \) in the first place. To filter trajectories by \( v_d \) and \( T \), we design an index called Footmark Index (FMI).

In FMI, we build a B*-tree \( BT_{v_d} \) for each vertex \( v_i \in \mathcal{V}(G) \). \( BT_{v_d} \) indexes the time of the trajectories reaching \( v_d \) and stores the corresponding trajectory id’s. Each leaf entry of \( BT_{v_d} \) is of the form \((t_i, t_a)\), where \( t_i \) is the unique id of a trajectory, and \( t_a \) is the time trajectory \( t_i \) reaching \( v_d \). Besides, we keep a table in FMI to map each vertex to its corresponding B*-tree. Figure 4 illustrates an example of FMI. Since \( Y \) traversed both \( v_1 \) and \( v_3 \), \((Y_1, t_{1-v_2}, t_{1-v_1})\) and \((Y_2, t_{2-v_3}, t_{2-v_2})\) are the leaf entries in \( BT_{v_1} \) and \( BT_{v_2} \), respectively.

Let \( |\Upsilon| \) and \( |\Upsilon|_{\max} \) denote the number and the maximum length of the trajectories in \( \Upsilon \), respectively. We can easily derive that the storage space consumed by FMI is \( O(|\Upsilon| \times |\Upsilon|_{\max}) \).

Algorithm 2 illustrates the process of footmark graph construction using FMI. FMI-Search\((v_d, T)\) (line 2) is the searching process in FMI. Let \( Y_{(v_d, T)} \) denote the set of all the trajectories arriving at \( v_d \) during \( T \). Given \( v_d \) and \( T \), FMI-Search\((v_d, T)\) returns the id’s of all the trajectories in \( Y_{(v_d, T)} \) via searching \( BT_{v_d} \). To calculate the footmarks, we further examine the elements of each \( Y \in Y_{(v_d, T)} \) (line 3–11). Besides, we build a \( |\mathcal{V}(G)| \times |\mathcal{V}(G)| \) matrix \( FG \) whose entries are initialized to zeros. For each newly acquired edge \((vid, vid') \in Y\), the corresponding entry \( FG[vid][vid'] \) is incremented by 1 (line 10). In this way, we calculate and record the edge weights of the target footmark graph in \( FG \) which is finally returned as the output.

![Figure 4: An example of FMI](image-url)
Algorithm 2: FMI-FG($v_d, T$)

begin
1. $FG \leftarrow |V(G)| \times |V(G)|$ matrix with all entries zeros;
2. $TRID \leftarrow$ FMI-Search($v_d, T$);
3. for each $tid \in TRID$ do
4. $Y \leftarrow$ GetTraj($tid$);
5. $(vid, t) \leftarrow$ the first element of $Y$;
6. while $t \not\in T$ do
7. $(vid, t) \leftarrow$ the next element of $Y$;
8. while $vid \neq v_d$ do
9. $(vid', t') \leftarrow$ the next element of $Y$;
10. $FG[vid][vid'] \leftarrow FG[vid][vid'] + 1$;
11. $(vid, t) \leftarrow (vid', t')$;
12. return $FG$;

Let $\mathcal{T}_{v_d}$ denote the set of all the trajectories traversing $v_d$. Then the cost of FMI-Search($v_d, T$) is $O(|\mathcal{T}_{v_d}|)$, i.e., the size of $\mathcal{T}_{v_d}$. After fetching the trajectories from disk, the algorithm calculates the footprint graph in $O(|\mathcal{T}_{v_d}| \times |Y_{max}|)$ time. Note that the performance bottleneck of Algorithm 2 is the I/O cost caused by trajectory fetching in GetTraj($tid$) (line 4). Specifically, for each returned $tid$, the corresponding trajectory needs to be transferred from the disk, incurring $|\mathcal{T}_{v_d}|$ random page reads in total.

4.2 Containment-Based Footmark Index

Although FMI can prune all the trajectories not in $\mathcal{T}_{v_d}$, it incurs $|\mathcal{T}_{v_d}|$ page accesses. To further reduce the number of random reads, we improve FMI by organizing the involved trajectories into different groups. In each group, the front part of each trajectory $Y$ before reaching $v_d$ (including $v_d$), denoted as $Y_{v_d}$, is contained by a unique ‘dominant’ trajectory. As a result, we only need to fetch the ‘dominant’ trajectory and calculate the footmarks of the contained ones by simply recording their starting locations w.r.t. the ‘dominant’ trajectory. We refer to this new index as Containment-Based Footmark Index (CFMI).

Definition 10 (v_d-Containment). For two trajectories $Y$ and $Y'$ in $\mathcal{T}_{v_d}$, if $Y_{v_d} \subseteq P$ is a sub-path of $Y_{v_d} \cup P$, then $Y$ is $v_d$-contained by $Y'$. In particular, if $Y_{v_d} \cap P \neq Y_{v_d} \cap P$, then $Y$ is strictly $v_d$-contained by $Y'$.

Definition 11 (v_d-Dominant). A trajectory $Y$ in $\mathcal{T}_{v_d}$ is $v_d$-dominant if there exists no $Y' \in \mathcal{T}_{v_d}$ such that $Y$ is strictly $v_d$-contained by $Y'$.

If there are multiple $v_d$-dominant trajectories in $\mathcal{T}_{v_d}$ which are $v_d$-contained by each other, we pick one of them as $v_d$-dominant and treat the rest as normal trajectories. As such, each $Y \in \mathcal{T}_{v_d}$ is $v_d$-contained by exactly one $v_d$-dominant trajectory. Consider the example in Figure 4. We have $\mathcal{T}_{v_d} = \{Y_1, Y_2\}$, $Y_1 \rightarrow v_0 \rightarrow v_1 \rightarrow v_2$, and $Y_2 \rightarrow v_0 \rightarrow v_1 \rightarrow v_2$. Obviously, $Y_1$ is $v_d$-contained by $Y_2$. This means that $Y_2$ follows the sub-path of $Y_1$ when traveling to $v_2$. Hence, we only need to fetch $Y_1$ to calculate both $Y_1$ and $Y_2$ by simply recording the starting location of $Y_2$ in $Y_1$. Similarly, $Y_2$ and $Y_1$ are both $v_1$-dominant in $\mathcal{T}_{v_d}$.

The basic idea of CFMI is to try to adopt the $v_d$-dominant trajectories to calculate $\mathcal{T}$. This is motivated by the following two observations.

- First, the number of the $v_d$-dominant trajectories in $\mathcal{T}_{v_d}$ tends to be much smaller than $|\mathcal{T}_{v_d}|$. In other words, the path to $v_d$ taken by a $v_d$-dominant trajectory is usually (partially) followed by many other trajectories. As we will show in Section 6, this is indeed the case in many real-world scenarios. Second, the range of $T$ is often much larger than the time spans of the trajectories. This means that the starting time of most of the trajectories in $\mathcal{T}_{v_d}$ are within $T$. Thus, we only need to fetch their $v_d$-dominant trajectories to calculate the corresponding footmarks.

CFMI improves the structure of each $B^*$-tree in FMI. Specifically, each leaf entry of $BT_{v_d}$ is in the following new form:

$\{tid, t_s, t_e, did, sloc\}$

where $t_s$ is the starting time of trajectory $tid$, $did$ is the id of the $v_d$-dominant trajectory of trajectory $tid$, and $sloc$ is the starting location of trajectory $tid$ in trajectory $did$. Besides, we keep a table $v_d$-Dom for each $BT_{v_d}$, in which we record the length of $Y_{v_d}$, $P$ for each $v_d$-dominant trajectory $Y$.

Figure 5 illustrates the CFMI for the example in Figure 4. Consider $BT_{v_2}$. Since $Y_1$ is the $v_2$-dominant trajectory, its $did$ equals to its $tid$. Besides, the $sloc$ of $Y_2$ is 3 since the starting location of $Y_2$, i.e., $v_3$, is the third location traversed by its $v_2$-dominant trajectory $Y_1$. In $v_2$-Dom, the length of $Y_1 \rightarrow v_2 \rightarrow P$ is recorded, which equals to the number of the vertices in $Y_1 \rightarrow v_2 \rightarrow P$.

Now, we use this example to illustrate the process of the footprint graph construction (Algorithm 3). Suppose the destination is $v_1$ and only $Y_2$ and $Y_3$ arrive at $v_1$ during $T$. Then CFMI returns two sets via searching $BT_{v_1}$ and $v_1$-Dom (line 2): 1) $TREC = \{(tid, t_s, did, sloc)\}$, which records the information of trajectories in $\mathcal{T}_{v_d}$, and 2) $DOM = \{(did, len)\}$, which records the $did$’s appeared in $TREC$ and their corresponding values in $v_1$-Dom. In this example, $TREC$ contains two elements, namely $\{(Y_2, v_0 \rightarrow Y_2, 1)\}$ and $\{(Y_3, v_2 \rightarrow v_3, 2, 4)\}$. $DOM$ contains one element, namely $\{(Y_2, 5)\}$. Note that $Y_4$ is not in $DOM$ because it is not returned as a $did$ in any elements of $TREC$. To calculate the footmarks, we create an array with length $len$ for each entry $(did, len)$ in $DOM$ and organize them in a structure $DA$ (line 3–6). Initially, the elements in each array are set to zeroes. In this example, $DA$ contains only one array $DA.Y_2$ with length 5. Next, for each $(tid, t_s, did, sloc)$ in $TREC$, we examine whether $t_s$ is in $T$. Assume that this is true for both $Y_2$ and $Y_3$. Then we get $Y_2$.did and $Y_3$.did, both of which point to $Y_2$. Further, we find the array $DA.Y_2$ and increment the values of $DA.Y_2[Y_2.sloc]$ and $DA.Y_2[Y_2.sloc]$ by one (line 10), respectively. Note that $DA.Y_2[2] = k$ means that the sub-path of $Y_2.P$ from the $k$th vertex to $v_1$ has been traversed by $k$ trajectories during $T$. In this way, we record the footmarks of the trajectories in the arrays of $DA$.

Then, we adopt $DA.Y_2 = [1, 0, 0, 1, 0]$ to calculate the footprint graph. Specifically, we fetch $Y_2$ from disk and do the following calculations (line 11–21). Since $DA.Y_2[1] = 1$, the first edge $(v_2, v_3)$ of $Y_2.P$ has been traversed once. Thus, $FG[v_d][v_4] = 1$. Since
Algorithm 3: CFMI-FG($v_k$, T)

\begin{algorithm}
\begin{algorithmic}
\State $FG \leftarrow |V| \times |V|$ matrix with all entries zeros;
\State $(TREC, DOM) \leftarrow$ CFMI-Search($v_k$, T);
\State $DA \leftarrow \emptyset$;
\For {each $(did, len) \in TREC$}
\State create array $DA.did[len]$ with all entries zeros;
\State $DA \leftarrow DA \cup DA.did[len]$;
\EndFor
\For {each $(tid, ts, did, sloc) \in TREC$}
\If {$ts \notin T$}
\State Modify-FG($tid$);
\Else
\State $DA.did[sloc] \leftarrow DA.did[sloc] + 1$;
\EndIf
\EndFor
\State $Y \leftarrow \text{GetTraj}(did)$;
\State $vid \leftarrow$ the first location of $Y.P$;
\State $k \leftarrow 1$, $w \leftarrow 0$;
\While {$vid \neq v_k$}
\State $vid' \leftarrow$ the next location of $Y.P$;
\If {$DA.did[k] \neq 0$ or $w \neq 0$}
\State $w \leftarrow w + DA.did[k]$;
\State $FG[vid][vid'] \leftarrow FG[vid][vid'] + w$;
\State $k \leftarrow k + 1$;
\State $vid \leftarrow vid'$;
\EndIf
\State $Y \leftarrow \text{GetTraj}(did)$;
\State $vid \leftarrow$ the first location of $Y.P$;
\State $k \leftarrow 1$, $w \leftarrow 0$;
\While {$vid \neq v_k$}
\State $vid' \leftarrow$ the next location of $Y.P$;
\If {$DA.did[k] \neq 0$ or $w \neq 0$}
\State $w \leftarrow w + DA.did[k]$;
\State $FG[vid][vid'] \leftarrow FG[vid][vid'] + w$;
\State $k \leftarrow k + 1$;
\State $vid \leftarrow vid'$;
\EndIf
\State $Y \leftarrow \text{GetTraj}(did)$;
\EndWhile
\State return $FG$;
\end{algorithmic}
\end{algorithm}

$DA.Y_2[2] = 0$ and $DA.Y_2[3] = 0$, no more trajectories have traversed the next two edges. We have $FG[v_d][v_2] = FG[v_2][v_5] = 1$. Besides, $DA.Y_2[4] = 1$ means that one more trajectory has traversed the 4th edge. It follows that $FG[v_5][v_2] = 1$. Since $v_1$ is reached, the calculation is over. Note that there are a small portion of trajectories whose starting time is not in $T$. To record their footmarks in $FG$, we read each of them from the disk and follow the lines 4–11 in Algorithm 2 (i.e., Modify-FG($tid$) in line 9).

Compared with FMI, CFMI calculates $DOM$ in an extra cost of $O(|Y_{v_d} \times \log |Y_{v_d}||)$. However, CFMI is more preferable due to two performance gains. First, instead of fetching each involved trajectory from the disk, CFMI only fetches their dominant ones. Random page reads are therefore largely reduced. Second, instead of checking the elements for each involved trajectory, CFMI records its $sloc's in the arrays and examines the dominant ones instead. Thus, the performance of computation is improved.

\section{SEARCHING THE TPMFP}

So far, we have constructed the footmark graph $G_f$ w.r.t. $\Omega = (G, T, v_s, v_d, T)$, where each edge $(v_s, v_i)$ in $G_f$ is associated with a positive weight $w_{v_i}$. The value of $w_{v_i}$ stands for the number of trajectories in $T$ that have reached $v_i$ via edge $(v_s, v_i)$ during $T$. Given $G_f$, we now proceed to the problem of searching the MFP from $v_s$ to $v_d$ on $G_f$, namely the TPMFP w.r.t. $\Omega$. In this section, we first prove that the MFP is a simple (acyclic) path on $G_f$ (Lemma 1 and Lemma 2). Based on this result, we then demonstrate that the problem has a recurrence structure that can be solved by dynamic programming (Lemma 3). Finally, we propose an efficient MFP searching algorithm to deal with sequence-valued path frequencies. Note that our algorithm has a similar structure to the Bellman-Ford algorithm.

\textbf{Lemma 1.} Let $u \leadsto v$ denote a path from $u$ to $v$. Suppose $P^* = v_s \leadsto v_k \leadsto v_k \leadsto v_d$ is a path with cycles on $G_f$. We have $F(P) > F(P^*)$, where $P$ is the resulting path after removing the portion of $P^*$ between consecutive visits to $v_k$.

\textbf{Proof.} Let $P_{v_k}$ be the cycle $v_k \leadsto v_k$ on $P^*$ and $w^*$ the first component of $F(P_{v_k})$. Let $F(P) = (w_1, \ldots, w_n)$. Thus, $w_1 \leq \cdots \leq w_n$. Consider the following cases:

- $w^* < w_1$: in this case, $w^*$ is the first component of $F(P^*)$ as well. Since $w_1$ is the first component of $F(P)$, we have $F(P) > F(P^*)$.

- $w_1 \leq w^* < w_n$: let $w_i \in F(P)$, $1 < i \leq n$, such that $w_{i-1} \leq w^* < w_i$. Thus, the first $i-1$ components of $F(P^*)$ are the same as those of $F(P)$. Note that $w^*$ and $w_i$ are the ith elements of $F(P^*)$ and $F(P)$, respectively. Hence $F(P) > F(P^*)$.

- $w_n \leq w^*$: in this case, $F(P)$ is the prefix of $F(P^*)$. Hence $F(P) > F(P^*)$.

In summary, we have $F(P) > F(P^*)$. \hfill $\square$

\textbf{Lemma 2.} Given $G_f$ w.r.t. $\Omega$, there exists an MFP from $v_s$ to $v_d$ that is simple, i.e., has at most $|V_f| - 1$ edges.

\textbf{Proof.} Since adding a cycle does not make a path more frequent, the $v_s$-to-$v_d$ MFP $P^*$ with the fewest number of edges does not repeat any vertex. If $P^*$ did repeat $v$, we could remove the cycle $v \leadsto v$, resulting in a more frequent path with less edges.

Let $F^*(v_s, i)$ be the frequency of the $v_s$-to-$v_d$ MFP using at most $i$ edges. By Lemma 2, the frequency of the $v_s$-to-$v_d$ MFP is $\hat{F}^*(v_s, |V_f| - 1)$. Let $P^*$ be the $v_s$-to-$v_d$ MFP. If $P^*$ uses at most $i-1$ edges, then $F^*(v_s, i) = F^*(v_s, i-1)$. Besides, by Theorem 2, if $P^*$ uses $i$ edges and the first edge is $(v_s, v)$, we have $F^*(v_s, i) = (w_{v_{s,v}}) + F^*(v, i-1)$, where the operator ‘+’ is defined as follows:

- If the two inputs are non-decreasing sequences of positive integers, ‘+’ merges them into a non-decreasing sequence. For example: $(20) + (5, 20) = (5, 20, 20)$;

- If one input is $\emptyset$, then the other input is returned. For example: $\emptyset + (5, 20) = (5, 20)$;

- If one input is $\#$, then $\#$ is returned. For example: $\# + (5, 20) = \#$.

We use $\|$ and $\#$ to represent the frequencies of empty path $(v_s = v_d)$ and null path, respectively. For completeness, we define $\emptyset$ and $\#$ as the highest and the lowest ranked path frequencies, respectively.

Let $Z = (F_1, F_2, \ldots, F_n)$ be an ordered sequence of path frequencies under the \guo relation, namely $F_1 \guo F_2 \guo \cdots \guo F_n$. Moreover, we define the maximum function over $Z$ as $\max(Z) = F_1$. Thus, we can recursively express $\hat{F}^*(v_s, |V_f| - 1)$ in smaller sub-problems.

\textbf{Lemma 3.} Given $G_f = (V_f, E_f)$, if $i > 0$, then we have $F^*(v_s, i) = \max_{(v_{s,v})} \left( (w_{v_{s,v}}) + \max_{(v, i-1) \in E_f} \left( F^*(v, i-1) \right) \right)$.
Based on this recursive formula, we design a dynamic programming algorithm to calculate MFP (Algorithm 4). In particular, for each u ∈ Vf, we use u,ξ to record F∗(u, i) found so far. Initially, we set u,ξ = #, if u ≠ vd and u,ξ = 0, otherwise (line 1–5). This means that the u–vd MFP using zero edge does not exist unless u = vd. To help with recovering the MFP, we maintain the next vertex of u in the u–vd MFP (found in every step) in u.suc. At the beginning, we set u.suc = null (line 5). Then, for i iterates from 1 to |Vf| − 1 (from line 7), we calculate the u–vd MFP using the recurrence formula in Lemma 3. The inner for loop (from line 8) is used to calculate the value of max,(wuv,v)∈Ef ((wuv,v) + F∗(v, i − 1)) and update u,ξ as well as u.suc if the u–vd MFP is changed. Finally, we return the v–vd MFP by sequentially retrieve the vertices along the successor links from vs to vd.

The correctness of Algorithm 4 follows directly by induction from the recursive formula in Lemma 4. Assume that the average number of vertices of all the MFP to vd is α. Then Algorithm 4 calculates the v–vd MFP in time O(α|Vf||Ef|) and requires O(α|Vf|) working memory (note that α ≤ |Vf|).

In real scenarios, one may start from a position along an edge instead of a vertex, or some vertex via which no trajectories has traversed to the specified destination. One possible solution is to retrieve this position’s k-nearest neighbors that are in Gf, calculate the MFP to vd from these k vertices, and return the most frequent one. Note that k is a user-specified parameter. This augmented algorithm has the same computational complexity as the original one.

6. EVALUATION

We use a real dataset to conduct the experiments. In this section, we first describe the dataset, the experiment environment and some implementation details. Then we analyze the effectiveness of the TPMFP query and the differences between TPMFP and MPR [3] by concrete examples. Finally, we present the evaluation results of the efficiency of our proposed indexes and algorithms.

6.1 Experiment Settings

6.1.1 Dataset

We conduct our experiments on a real dataset containing over 0.5 billion GPS records. These GPS records are collected from around 6,000 taxis in Shanghai in 2007. Along with time and location, each record has an attribute indicating whether the taxi is empty or occupied at that time. We use this attribute to extract meaningful trips of passengers and treat each trip as a single trajectory. This preprocessing technique is also used in [32]. Then we perform a specially tuned map-matching algorithm derived from [18] to align all trajectories to Shanghai’s digital map which contains 22,180 vertices and 65,510 edges. After map-matching, each trajectory is represented in the form defined in Section 2.

The preprocessed dataset consists of 11,547,611 trajectories. The total number of vertices is 245,276,717 and the average trajectory length (in terms of the number of vertices) is 21. For each trajectory, the storage size of the trajectory id, the time and the vertex id is 18 bytes, 8 bytes and 4 bytes, respectively. We can see that the dataset is very large and that the all-in-memory techniques are no longer applicable. Since some algorithms may use too much memory or take too long to finish, we also create two subsets of the original data containing the trajectories of a month and a day, respectively. For clarity, we name these datasets as Year Dataset, Month Dataset and Day Dataset. A summary of the datasets is given in Table 2.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Trajectories</th>
<th>Total Length</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year Dataset</td>
<td>11,547,611</td>
<td>245,276,717</td>
<td>3,335</td>
</tr>
<tr>
<td>Month Dataset</td>
<td>1,650,134</td>
<td>35,619,454</td>
<td>484</td>
</tr>
<tr>
<td>Day Dataset</td>
<td>54,579</td>
<td>1,217,890</td>
<td>17</td>
</tr>
</tbody>
</table>

6.1.2 Implementation

We implement all our algorithms in Java. The FMI and CFMI indexes are stored as regular files. To handle big datasets, we develop a compact file format optimized for both sequential full scan and single trajectory retrieval.

To compare with MPR, we also implement the related algorithms in [3]. As we have a digital map on hand, we skip the steps of generating a road network from raw trajectories. We believe this change is acceptable as it makes the results more realistic. For our implementation of MPR, most algorithms are written in Java except that matrix operations are written in R. We do so because R is much more efficient than Java in manipulating matrices.

6.1.3 Experiment Environment

We perform all evaluations using a single server. It has a quad-core Intel(R) Xeon(R) E5506 CPU (2.13GHz), 12GB memory and 10,000RPM sever-level hard disks. The operating system is Linux 2.6.32 x86_64 and the Java VM version is 1.7.0_4 64-Bit.

6.2 Effectiveness

To evaluate the effectiveness, we compare the results of four path finding queries, namely TPMFP (or MFP when T is fixed), MPR (Most Popular Route), STP (shortest path) and LRS (path with the least number of road segments). LRS represents the paths with minimum number of hops and it can be easily calculated using any shortest path algorithm by setting all edge weights to 1. Note that the calculations of STP and LRS do not need any trajectories. For TPMFP and MPR, if not otherwise specified, we use Year Dataset and the time period is fixed to the whole year of 2007 (as if T is [0, +∞)).

6.2.1 General Comparison

The very first question we are curious about is whether these queries always find the same paths. To examine this, we fix the des-
is more direct and its travel distance is much smaller than that of this case the MPR may be faster than MFP, we notice that the MFP consists of many road segments of regular roads. Though in most of which are viaducts or high-speed roads; while the last example uses Year Dataset.

In Figure 6, the first three examples use Month Dataset and the last example uses Year Dataset.

In Figure 6(a), the MPR consists only a few number of road segments, most of which are viaducts or high-speed roads; while the MFP consists of many road segments of regular roads. Though in this case the MPR may be faster than MFP, we notice that the MFP is more direct and its travel distance is much smaller than that of the MPR. Moreover, we find that there does exist several trajectories traveling along the MFP, whereas there is no trajectory traveling along the MPR in Month Dataset. Based on these observations, we believe most drivers would choose the more direct path as our approach suggests.

As we have mentioned before, the MPR algorithm is prone to select paths with fewer vertices. The reason is that the popularity score of a path is the product of the weights (\( \in [0, 1] \)) of all vertices along the path. The experiment result also confirms this inference, as is shown in Figure 7(b). There are over 90% of MPRs are also LRSs. In other words, even if a path is taken by most travelers, it has little chance to be chosen by the MPR algorithm if it contains many more vertices than other less frequently used paths. In contrast, there are less than 10% of the MFPs overlapped with LRSs. Therefore, we can safely conclude that MFP is much more length-insensitive than MPR.

6.2.2 Case Study

Below we conduct several case studies to exhibit the effectiveness of the TPMFP query. One thing worth noting is that there is neither any ground truth of truly frequent paths nor any widely-accepted definitions of it, the analysis below might be more or less subjective. Despite that, we believe they do give some strong arguments of why the TPMFP query are useful.

In Figure 6, the first three examples use Month Dataset and the last example uses Year Dataset.

In Figure 6(a), the MPR contains too many twists and turns while the shape of the MFP is much nicer. There is no clear reason making us believe that many past travelers would take the former instead of the latter. The interesting thing here is why the MFP turns...
left at $M_5$ instead of $M_6$. We discover that the sub-path of the MFP from $S_4$ to $M_5$ is a high-speed road and there is no exit at $M_6$. Therefore, most drivers would exit the high-speed road at $M_5$ when traveling to $D_8$.

We perform many TPMFP queries with various time periods and try to discover the changes of the MFP. Figure 6(d) is a representative result, in which the MFP from $S_4$ to $D_8$ changes around September 2007. By searching the web we discover that the road between $M_7$ and $M_8$ is under construction before then. This example demonstrates that time period-based MFP can be more useful than MFP with a fixed time period.

6.3 Efficiency
In this section, we evaluate the performance of the indexes, the footmark construction algorithms and searching the MFP on a footmark graph.

6.3.1 Index Creation
The creation of FMI and CFMI consists of two steps. First, we create a file for each vertex recording all the trajectories traversing it. This step can be performed efficiently with a single pass of scanning all trajectories. Second, we process each of the resulting files to build FMI or CFMI. For FMI, it is sufficient to sort the records by the arrival time and then pack a B+tree on disk. For CFMI, we use a nested-loop algorithm to calculate the dominant trajectories before building the index. Specifically, we first sort all the trajectories in the same file by length in descending order. Then we iterate over all these trajectories in the sorted order and mark a trajectory as a dominant one if it is not dominated by any of the dominant trajectories found so far. For Year Dataset, the index creation time of FMI and CFMI is 72 minutes and 127 minutes, respectively. Since we only need to create the index once, the speed of the index building algorithms is acceptable.

6.3.2 Index Size
Figure 8 shows the total storage size of different indexes. The size of FMI is slightly smaller than twice of the data size. The size of CFMI is about three times of the data size. Note that we can further improve the space efficiency of CFMI by removing the trajectory id’s in the tree structure because they are not used when constructing footmark graphs using CFMI. As a result, the size of CFMI is reduced by 40%. Moreover, we discover that the ratio of the size of the dominant trajectory table in CFMI to the total size of CFMI decreases as the data size grows.

Figure 9 illustrates the decreasing trend of the percentage of dominant trajectories of a representative vertex as more trajectories are added. The result shows that when there are 150,000 trajectories passing through the vertex, we only need to fetch 10% of the trajectories from disk. Figure 10 shows the percentages of dominant trajectories of all vertices. We observe that this ratio is below 0.2 for most vertices, especially when the vertex is traversed by a relatively large number of trajectories (e.g., greater than 10,000). For TPMFP queries over Year Dataset, the above results demonstrate that CFMI can reduce the number of random disk accesses by a factor of 5 compared with FMI.

6.3.3 Query Performance of TPMFP
The response time of an TPMFP query consists of two parts, namely the time of constructing the footmark graph and the time of searching the MFP in the footmark graph.

To evaluate the time of footmark graph construction, we compare three algorithms including Algorithm 2, Algorithm 3 and an algorithm that performs full scan on all trajectories. Moreover, we consider three memory modes. The Tiny-Dataset mode loads all trajectories and indexes into memory to avoid any disk access. The Small-Dataset mode loads all trajectories and part of the indexes including all dominant trajectory tables (CFMI only) and all non-leaf B+−tree nodes into memory. The Big-Dataset mode differs from the Small-Dataset mode in that it does not load trajectories into memory. In this experiment, we fix the destination and use different time periods to vary the number of trajectories involved. The performance of footmark graph construction on Month Dataset is shown in Figure 13. We can see that the algorithm using CFMI outperforms the other two in all cases. For both FMI and CFMI, the footmark construction time is increasing linearly with the number of trajectories.

The footmark construction time using CFMI on Year Dataset is depicted in Figure 11. The destination is set to the busiest road intersection, i.e., the vertex with the greatest number of trajectories passed through. Therefore, the result can reflect the worst cases among all possible queries. We can see that our algorithm takes less than 40 seconds to construct the largest footmark graph, which is fairly efficient for a dataset containing over 11.5 million trajectories.

Once the footmark graph is created, the MFP searching algorithm will be executed in memory. The performance of Algorithm 4 is only affected by the size of the graph, which is depicted in Figure 12. The result shows that searching MFP is slower than searching the shortest path. The reason is that the computation complexity of updating vertex status in the MFP algorithm is $O(\alpha)$, which is higher than that in the other two algorithms ($O(1)$). However, for big datasets, the performance of MFP searching algorithm is not critical as the execution time of the whole TPMFP query is dominated by the footmark construction.

6.3.4 TPMFP vs. Time Period-Based MPR
Since the original MPR algorithm does not deal with user-specified time periods, it performs a lot of time-consuming pre-computations and saves the results for future MPR queries. When dealing with dynamic time periods, these pre-computations must be performed in an online manner, which leads to extremely slow response time. For example, we conduct a rough comparison using Month Dataset with $T$ set to the whole month. The response time of ‘time period-based MPR’ is around 200 seconds, while the TPMFP algorithm takes less than 1 second.
7. RELATED WORK

Finding the most desirable path has received tremendous research interests for decades. The most popular topic in this area is shortest path finding [6, 1, 22], which has been extensively studied for over fifty years. If the weight on each edge represents travel time, shortest path finding becomes fastest path finding. Time-dependent shortest path problem [7, 13, 21] regards the travel time of an edge as a single-valued function on time of day. To improve routing services, new approaches [11, 29, 30] for fastest path finding are proposed aiming at using user-generated GPS trajectories to estimate the distribution of travel time on a given road network. Unlike shortest/fastest path finding, our work studies the desirability of a path from a different perspective, i.e., how frequently the path has been taken during a certain time period.

Popular route searching [3, 27] is the most closely related work to the TPMFP problem. Zaiben et al [3] are the first to study the common routing preferences of the past travelers. Utilizing historical trajectory datasets, it proposes a novel popularity function for path desirability evaluation. However, this method tends to favor the paths with fewer vertices. Moreover, the most popular route (MPR) may contain at least frequent edges. The work in [27] aims at finding top-k popular routes from uncertain trajectories. Assuming that there is no map at hand, they focus on deriving routes from uncertain trajectory data. They calculate the popularity of each edge by simply counting the number of trajectories traversed it, without considering whether they have passed through the specified destination. It is therefore not suitable for the application scenarios of the TPMFP problem. Note that both of these works only consider the fixed time range, while the TPMFP problem can be carried out in arbitrary time periods specified by the users.

Hot route and trajectory pattern detection also try to reflect the common routing behaviors of the past travelers. Informally, a hot route is a path with heavy traffic. Various trajectory clustering approaches [16, 15, 9, 14] can be utilized to discover hot routes. In [24], an online algorithm is developed to detect the hot motion paths that have been frequently traversed by the past travelers. Similar to hot routes, trajectory patterns represent the frequent travel behaviors in terms of both time and locations. Typical approaches for trajectory pattern mining include T-pattern mining [10], periodic pattern mining [19], interesting locations and travel sequences mining [33], etc. Unlike the problem of TPMFP, these approaches find the desired hot routes/trajectory patterns in a global manner and no specific source and destination are considered. They are therefore not suitable for finding the MFP from a specified location to another.

Another relevant problem to our work is the management of trajectory data. So far, a number of data access methods, which are variants of R-tree [12], have been proposed for moving object trajectories. Typical examples include 3D R-trees [26], MR-trees [28], HR-trees [20], MV3R-trees [25], TB-trees [23], SETI [2], TrajStore [4], FNR-trees [8] and MON-trees [5]. In particular, FNR-trees and MON-trees are designed for indexing trajectories that have already been matched to a road network. However, these structures are not suitable for efficient footmark retrieving due to two reasons. First, the query of footmarks is much more complex than the ones (e.g., window/range queries and timestamp/interval queries) that these structures are primarily designed for. Second, they become inefficient when the results are very large. In our work, we proposed special techniques (e.g., CFMI) to calculate most of the results instead of fetching them all from disk.

723
8. CONCLUSION

In this paper, we study the problem of TPMFP, i.e., finding the time period-based most frequent path. We propose a novel definition of TPMFP which satisfies three key properties, namely suffix-optimal, length-insensitive and bottleneck-free. This is a new query and the definition of TPMFP well reflects the people’s common sense notion. Moreover, we devise a two-step framework to efficiently perform TPMFP queries on very large trajectory datasets. The first step is to construct a footmark graph which can be used to calculate the frequencies of the candidate paths. To this end, we propose two novel indexing schemes to reduce the number of random disk accesses. The second step is to search the most frequent path in the footmark graph and an efficient algorithm is proposed. In addition, we conduct extensive experiments using a real big trajectory dataset containing 11.5 million trajectories. The results demonstrate the effectiveness and the efficiency of our index schemes and algorithms.

Acknowledgement

This research was supported in part by Hong Kong, Macao and Taiwan Science & Technology Cooperation Program of China under Grant 2012DFH10010, Science and Technology Planning Project of Guangzhou China under Grant 2012Y2-00030, Huawei Corp. Contract YBCC2009041-27, Huawei Corp. Contract HWLB06-15C032121/3PN, Hong Kong RGC GRF Project No.611411, National Grand Fundamental Research 973 Program of China under Grant 2012-CB316200, HP IRP Project, and Microsoft Research Asia Grant.

9. REFERENCES