

4-2018

A novel representation and compression for queries on trajectories in road networks

Xiaochun YANG
Northeastern University

Bin WANG
Northeastern University


Kai YANG
Northeastern University

Chengfei LIU
Swinburne University of Technology

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

DOI: <https://doi.org/10.1109/TKDE.2017.2776927>

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the [Databases and Information Systems Commons](#), and the [Numerical Analysis and Scientific Computing Commons](#)

Citation

YANG, Xiaochun; WANG, Bin; YANG, Kai; LIU, Chengfei; and ZHENG, Baihua. A novel representation and compression for queries on trajectories in road networks. (2018). *IEEE Transactions on Knowledge and Data Engineering*. 30, (4), 613-629. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/3870

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

A Novel Representation and Compression for Queries on Trajectories in Road Networks

Xiaochun Yang, *Member, IEEE*, Bin Wang, Kai Yang, Chengfei Liu, and Baihua Zheng, *Member, IEEE*

Abstract—Recording and querying time-stamped trajectories incurs high cost of data storage and computing. In this paper, we explore several characteristics of the trajectories in road networks, which have motivated the idea of coding trajectories by associating timestamps with relative spatial path and locations. Such a representation contains large number of duplicate information to achieve a lower entropy compared with the existing representations, thereby drastically cutting the storage cost. We propose several techniques to compress spatial path and locations separately, which can support fast positioning and achieve better compression ratio. For locations, we propose two novel encoding schemes such that the binary code can preserve distance information, which is very helpful for LBS applications. In addition, an unresolved question in this area is whether it is possible to perform search directly on the compressed trajectories, and if the answer is yes, then how. Here we show that directly querying compressed trajectories based on our encoding scheme is possible and can be done efficiently. We design a set of primitive operations for this purpose, and propose index structures to reduce query response time. We demonstrate the advantage of our method and compare it against existing ones through a thorough experimental study on real trajectories in road network.

Index Terms—Road network, Trajectory, Compression, Representation

1 INTRODUCTION

For the purpose of reducing overhead in data storage and processing, trajectory compression is to compress the size of trajectories while maintaining their utility. In this paper, we consider both storing and querying trajectories in road network. We aim to store trajectories using relatively small space and support queries with high performance.

There are mainly two types of representations for trajectories in road networks. A typical type of expressions combines a timestamp t and a 2D position (x, y) together in the form of (t, x, y) to express a time-stamped position in a trajectory [28]. Such a representation causes big overhead of data storage and computing. The other type of expressions separates spatial locations from timestamps, using consecutive edges $\langle e_i, \dots, e_j \rangle$ to represent a spatial path of a trajectory, and a sequence of distance-time pairs, each of which is represented as (d_i, t_i) , to capture the temporal information. Accordingly, lossy compression approaches are proposed for spatial and temporal information respectively. PRESS [20] and the generalized in-network trajectory data model proposed by Sandu-Popa et al in [17] are the latest representative works of the second type of expressions. However, a good compression ratio can only be achieved under a large error bound/error threshold. [17] does not report how to do query processing on their compressed trajectories, and the query processing in [20] heavily relies on decompression.

In this paper, we propose a novel representation, a lossless compression for both spatial path and timestamps, and an error-bounded compression for locations. Such representation and compression can achieve high compression ratio under a small error bound and can support queries efficiently. The first challenge of this work is to design a good representation with small entropy (i.e. the representation contains large amount of duplicate information) to facilitate storing and querying trajectories in road networks, considering both space overhead and efficiency. This means that, from the compression perspective, it prefers entropy of a trajectory representation to be low; and from the querying perspective, it aims at being able to support query processing efficiently. To attain the above goals, we propose a novel representation of trajectories in road networks (called TED-representation), where a trajectory is represented by a spatial entry path (E), distances (D) that locations appear in the spatial entry path, and a time flag sequence (T) to indicate if a position appears in a certain edge of a spatial entry path at a certain timestamp (see Section 2). This separation provides ideal properties to support both compression and location-based query processing, with mainly two advantages. First, it enables us to capture characteristics of trajectories in road networks, and enables our expression to achieve lower entropy than existing representations. Second, it allows us to easily associate these three dimensions to build up a close relation among a spatial path, locations, and timestamps. Follow the most representative work PRESS [20] that separates each trajectory into a spatial path and a temporal sequence, we further separate the temporal sequence into locations and timestamps, which enables us to effectively cut down the error bound.

The second challenge addressed in this paper is to propose compression algorithms to transform TED representation into shorter binary words \hat{T} , \hat{E} , and \hat{D} , respectively, such that TED representation can be recoverable from \hat{T} , \hat{E} , and \hat{D} (see Section 3). We propose several techniques to compress spatial entry paths and locations separately. For spatial entry paths, we

- X. Yang, B. Wang, and K. Yang are with the School of Computer Science and Engineering, Northeastern University, China.
Email: {yangxc, binwang}@mail.neu.edu.cn, yangkai@stunmail.neu.edu.cn
- C. Liu is with the Faculty of Science, Engineering and Technology, Swinburne University of Technology, Australia.
Email: cliu@swin.edu.au
- B. Zheng is with the School of Information Systems, Singapore Management University, Singapore.
Email: bhzheng@smu.edu.sg

propose a fixed-length encoding for a single path and consider the feature of trajectories in a road network to compress multiple trajectories. Such a compression can support fast location and it is also able to achieve a high compression ratio, especially when the total number of trajectories that need compression is large. We will demonstrate that this compression can drastically reduce the storage costs, achieve a high compression ratio, and support all kinds of paths, including non-shortest paths, acyclic single trajectories, periodical trajectories, and multiple trajectories. For locations, we first propose a distance-preserving encoding scheme called *DP-encoding* to encode locations. Then we propose a novel encoding scheme *DDP-encoding* to make the code decodable, and a pruned *DDP-encoding* to further save space so that the size of codes for location is close to that of Huffman encoding. We show that these two encoding schemes can preserve distance information as well as process queries efficiently for Location-Based Service (LBS) applications.

The last but not the least challenge addressed in this work is to devise techniques to answer typical queries on compressed trajectories. As we know there are so many queries supporting different LBS applications, it is impossible and unnecessary to enumerate all of them in a paper. Therefore, we list two types of primitive operations, which are fundamental functions to support LBS related applications. We show that query processing can be effectively limited to a small candidate region in compressed trajectories, and only a small part of data needs to be decompressed, which is efficient. We propose novel index structures and algorithms for this purpose, and reduce the primitive operation response time. We then present our algorithms for four types of commonly used LBS queries (see Section 4). We finally demonstrate the advantage of our method and compare it against existing ones through a thorough experimental study on real trajectories in road network (see Section 5).

2 TRAJECTORY REPRESENTATION AND FRAMEWORK

A trajectory Tr is a series of time-stamped raw positional data p in the form of (t, x, y) , where t is a timestamp, and (x, y) refers to a location in a 2D Euclidean space with a latitude x and a longitude y . Fig. 1(a) shows a raw trajectory. A road network is generally defined as a directed graph $G=(V, E)$, where V is the vertex set and E is the edge set. Each vertex has different exit entries pointing to different edges. The exit entries of a vertex are unique consecutive numbers starting from 1. Fig. 1(b) shows an example of a road network, where v_i represents a vertex and e_i represents an edge. Each v_i has a few exit entries. For instance, v_6 has three exit entries labeled by 1 to 3.

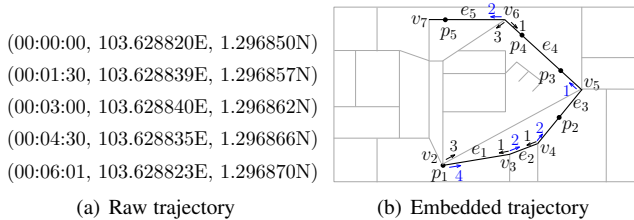


Fig. 1: A trajectory Tr_1 embedded in a road network.

A raw trajectory can be embedded to an *embedded trajectory* in a road network using map matching process [20], with each

time-stamped raw positional data p mapped to an *embedded data*. The bold path in Fig. 1(b) represents an embedded trajectory Tr_1 corresponding to the raw trajectory containing five time-stamped raw positional data shown in Fig. 1(a). Take PRESS [20] as an example. It transforms a raw trajectory in the form of (t, x, y) s to a spatial path with a sequence of edges and a temporal sequence formed by a set of time-distance pairs. To be more specific, PRESS converts each time-stamped data p of a trajectory Tr to $(e, \langle d, t \rangle)$, where e indicates the edge that p locates, t is the time-stamp where p is sampled, and d records the distance travelled along Tr from the start of the journey to p . For instance, PRESS expresses p_2 in Fig. 1(b) as $(e_3, \langle 00:01:30, |e_1| + |e_2| + \|p_2 - v_4\|_1 \rangle)$ indicating that p_2 locates in the edge e_3 at 00:01:30 and the distance between v_2 (the start point of its trajectory) and p_2 is $|e_1| + |e_2| + \|p_2 - v_4\|_1$.

2.1 Framework

We propose a new framework to compress trajectories for physical storage and to support query processing for LBS applications. Fig. 2 shows our framework. Given a set of GPS trajectories, each of which can be converted to an embedded trajectory via map matching process [12], [20]. Like PRESS, we represent each time-stamped data p via a triple $(\underline{time}, \underline{edge}, \underline{distance})$, therefore an embedded trajectory can be represented by a *time sequence* (T), a *spatial entry path* (E), and a *distance sequence* (D) (with details presented in Section 2.2). Unlike PRESS, spatial information is represented by spatial entry paths and distances with a much lower entropy and hence corresponding compression algorithms are able to achieve much higher compression ratios. The representation of timestamps (T) facilitates the association between entry paths (E) and distances (D). A trajectory represented in this TED format is called a *TED-trajectory*.

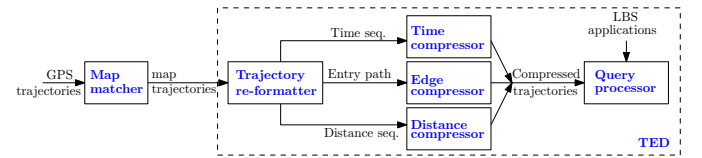


Fig. 2: Our framework for representing and compressing trajectories.

We then compress TED-trajectories by compressing T , E , and D separately, with details presented in Section 3. Efficient search algorithms are developed to process LBS queries by only partially decompressing compressed trajectories, with details presented in Section 4.

2.2 A New Representation: TED Format

We represent a raw trajectory using the following TED format.

E: Entry path for a trajectory. Entry path $E(Tr)$ captures a sequence of edges passed by a trajectory Tr . Exiting path representations use a sequence of consecutive edges [9], [19], [20], [24], while we would like to propose a new representation. Recall that each vertex $v \in V(G)$ in a road network G has different exit entries pointing to different edges. Inspired by this, a directed edge from a vertex v can be represented by vertex v and one of its exit entries (i.e. the i -th exit entry, where $i \geq 1$). Accordingly, an entry path could be represented by the start vertex of the path and a sequence of integers, each of which corresponds to an exit entry. In order to distinguish an edge e that has $h (> 1)$ raw positional

data located on from an edge e' that has zero/one raw positional data located on, we append $(h-1)$ 0s after the exit entry of e . For example, the entry path $E(Tr_1)$ for the trajectory Tr_1 in Fig. 1 is represented as $v_1 \rightarrow 4, 2, 2, 1, 0, 2$, and the bit 0 is to indicate that there are two raw positional data on edge e_4 .

D: Distance sequence for a trajectory. The sequence of relative distances $D(Tr)$ records the exact locations along the edges corresponding to the embedded data. Note that a raw positional data can be located at any point of an edge, not necessarily at a vertex. PRESS [20] uses distance-time $\langle d, t \rangle$ pairs to record the distance to the start point of the whole trajectory at a certain timestamp. This expression is incompressible, because (i) for any two tuples $\langle d_i, t_i \rangle$ and $\langle d_j, t_j \rangle$ of a trajectory, t_i is definitely different from t_j ; and (ii) for a tuple $\langle d_i, t_i \rangle$ of trajectory Tr_i and another tuple $\langle d_{i'}, t_{i'} \rangle$ of trajectory $Tr_{i'}$, d_i is very likely to be different from $d_{i'}$.

In this work, we propose to separate locations from the timestamps. Given a raw positional data p on the edge e ($v_i \rightarrow en$), we use the distance from the start vertex v_i of e to the point p to indicate the exact location of p along the edge. Since the lengths of edges in a road network vary greatly, we normalize distances to *relative distance* within $[0, 1)$ range, as defined in Definition 1.

Definition 1. Relative distance. Given an edge e ($v \rightarrow en$) and a location $p = (x, y)$ on this edge, let d be the network distance between the start vertex v and p . The relative distance of p w.r.t. e is a ratio of d to the length of e , denoted $r(p) = \frac{d}{|e|}$.

Then, locations of raw positional data points of a trajectory can be expressed as a sequence of relative distances. For instance, the locations w.r.t. five points of Tr_1 in Fig. 1 are represented as $\langle 0, 0.5, 0.375, 0.75, 0.75 \rangle$. Note if two locations have the same relative distances to their own start vertices, they are duplicate and become compressible.

T: Time sequence and a time flag bit-string. $T(Tr)$ of a trajectory Tr is to record the list of timestamps $\langle t_1, t_2, \dots \rangle$ w.r.t. Tr , together with a binary string of time flags $\hat{T}(Tr)$. Besides $T(Tr)$, we introduce $\hat{T}(Tr)$ to associate time stamps in $T(Tr)$ with entry paths in $E(Tr)$ and relative distances in $D(Tr)$. Obviously, the mapping between timestamps in $T(Tr)$ and relative distances in $D(Tr)$ follows one-to-one mapping strictly, but not the mapping between time stamps in $T(Tr)$ and entry paths $E(Tr)$ as an edge passed by the trajectory Tr could have no embedded data (e.g. the underlined digit 2 in the entry path $v_2 \rightarrow 4, \underline{2}, 2, 1, 0, 2$ in Fig. 3). Therefore, we introduce a bit-string $\hat{T}(Tr)$ sharing the same length as $E(Tr)$. The j^{th} bit in $\hat{T}(Tr)$ is associated with the j^{th} digit in $E(Tr)$. If the j^{th} digit in the entry path $E(Tr)$ is to indicate the j^{th} edge passed by the trajectory Tr without any embedded data, the j^{th} bit in $\hat{T}(Tr)$ is set to 0; otherwise, it is set to 1. In other words, all the bits in $\hat{T}(Tr)$ corresponding to underlined digits in $E(Tr)$ are set to 0 while the rest bits are set to 1. Notice that, the first j bits in $\hat{T}(Tr)$ (i.e. the substring $\hat{T}(Tr)[1, j]$) must contain i bit 1s since it corresponds to i timestamps in $T(Tr)$. Therefore, the number of 1s in $\hat{T}(Tr)$ must be equivalent to the number of timestamps captured by $T(Tr)$ and the number of relative distances captured by $D(Tr)$. For example, the time flag sequence $\hat{T}(Tr_1)$ is $(101111)_2$, where $(\cdot)_2$ is the binary bits. Fig. 3 shows the mapping relationship among relative distances $D(Tr_1)$, timestamps $T(Tr_1)$, time flag bit-string $\hat{T}(Tr_1)$, and the entry path $E(Tr_1)$.

Entry path $E(Tr_1)$	$v_2 \rightarrow 4$	<u>2</u>	2	1	<u>0</u>	2
Time flags $\hat{T}(Tr_1)$	1	0	1	1	1	1
Time seq. $T(Tr_1)$	00:00:00		00:01:30	00:03:00	00:04:30	00:06:01
Distance seq. $D(Tr_1)$	0		0.5	0.375	0.75	0.75

Fig. 3: An example of TED representation.

2.3 Converting Raw Data to TED trajectory

In the following, we explain how to form the entry path E , the distance sequence D , and the time sequence T for a given trajectory.

Algorithm 1 shows the conversion from a sequence of raw positional data to a TED trajectory. It first invokes MapMatch to embed raw positional data in edges $\langle e_1, \dots, e_k \rangle$ in a road network. Then it iteratively checks if a positional data (x_j, y_j) locates in an edge e_i until all edges and positional data have been processed (lines 4 – 17). For each edge $e_i = v_i \rightarrow en_i$, if there exists a positional data locating in e_i , it appends $\frac{d_j}{|e_i|}$ to D , t_j to T , and a bit 1 to \hat{T} , otherwise, it only appends a bit 0 to \hat{T} . If there are l (> 1) positional data locating in e_i , it appends $l-1$ 0s to E (lines 11 – 13). Finally, it returns T, E, D, \hat{T} . The time complexity is linear to the number of raw positional data.

Notice that, our TED representation is not affected by the shape of trajectories (e.g. a trajectory with a circle).

Algorithm 1: CONVERT.

Input: A sequence of raw data $\langle t_1, x_1, y_1 \rangle, \dots, \langle t_m, x_m, y_m \rangle$;
Output: The corresponding TED format;

- 1 Get edges $\langle e_1, \dots, e_k \rangle$ and positions $\langle d_1, \dots, d_m \rangle$ by invoking $MapMatch(\langle t_1, x_1, y_1 \rangle, \dots, \langle t_m, x_m, y_m \rangle)$;
- 2 $E \leftarrow \emptyset; D \leftarrow \emptyset; T \leftarrow \emptyset; \hat{T} \leftarrow \emptyset$;
- 3 $i \leftarrow 1; j \leftarrow 1; FLAG \leftarrow \text{true}$;
- 4 **repeat**
- 5 **if** $FLAG$ **then**
- 6 **if** (x_j, y_j) locates at $e_i = (v_i, en_i)$ **then**
- 7 $D.append(\frac{d_j}{|e_i|}); T.append(t_j); \hat{T}.append(\text{bit } 1)$;
- 8 **else**
- 9 $\hat{T}.append(\text{bit } 0)$;
- 10 $E.append(en_i); FLAG \leftarrow \text{false}; j++$;
- 11 **while** (x_j, y_j) locates at $e_i = (v_i, en_i) \&\& !FLAG$ **do**
- 12 $E.append(0); D.append(\frac{d_j}{|e_i|})$;
- 13 $T.append(t_j); \hat{T}.append(\text{bit } 1); j++$;
- 14 $FLAG \leftarrow \text{true}$;
- 15 **else**
- 16 $i++$;
- 17 **until** $i > k \&\& j > m$;
- 18 **return** T, E, D, \hat{T} ;

Benefit of TED Format. By using the new format, the spatial path $\langle e_1, e_2, e_3, e_4, e_5 \rangle$ of the trajectory Tr_1 shown in Fig. 1(b) can be represented as $v_2 \rightarrow 4, 2, 2, 1, 0, 2$. As compared with the existing representation, the new representation contains more duplicates. The entropy of the entry path using our representation is $H(P)_{TED} = -\sum_{i=1}^m f(en_i) \log_2 f(en_i)$, where en_i is the exit entry number and $f(en_i)$ is the proportion of en_i in all exit entries in the collection of trajectories. For example, for the entry path $v_2 \rightarrow 4, 2, 2, 1, 0, 2$, the Shannon entropy using our representation is $H(P)_{TED} = -\frac{1}{6} \log_2 \frac{1}{6} - \frac{3}{6} \log_2 \frac{3}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} = 1.79$. Compared with the

entropy $H(P) = -\sum_{i=1}^m f(e_i) \log_2 f(e_i) = 2.32$ using the existing representation $\langle e_1, e_2, e_3, e_4, e_5 \rangle$, our representation for spatial path has a much lower entropy.

In addition, we separately represent distance d from its time stamp. By proposing relative distances, a non-cyclic trajectory has a higher probability to contain duplicate relative distances and becomes compressible.

3 TED-TRAJECTORIES COMPRESSION

Given a trajectory Tr in the form of (t, x, y) sequence, and a compressed trajectory Tr' of Tr based on its TED representation, the effectiveness of the TED representation and compression is evaluated by the *compression ratio* which is defined as the rate of Tr 's storage cost to that of Tr' , i.e., $\frac{|Tr|}{|Tr'|}$. In this section, we explain how to perform compression on TED-trajectories. The high compression ratio achieved by TED-trajectories well justifies the advantage of TED representation in terms of compression.

3.1 Lossless Multiple Entry Paths Compression

Given an entry path $E(Tr)$, each exit entry en_i in $E(Tr)$ occupies $\lceil \log_2 k \rceil + 1$ bits, where k is the maximal number of entries in any vertex $v \in V$. We call these $\lceil \log_2 k \rceil + 1$ bits an *entry code* w.r.t. en_i . Let $\hat{E}(Tr)$ be a sequence of entry codes w.r.t. entries in $E(Tr)$. For example, if $k = 7$, the entry path $E(Tr_1): v_2 \rightarrow 4, 2, 2, 1, 0, 2$ has a corresponding $\hat{E}(Tr_1) = (\underline{100} \ 010 \ \underline{010} \ \underline{001} \ \underline{000} \ \underline{010})_2$, where every 3 bits represent an entry. Accordingly, given an entry path $E(Tr)$ of m entries, $\hat{E}(Tr)$ occupies in total $(\lceil \log_2 k \rceil + 1) \cdot m$ bits. In the following, we consider n trajectories $\{Tr_1, \dots, Tr_n\}$ with each $\hat{E}(Tr_i)$ having m entries. We present a basic compression algorithm and also an enhanced version to compress and store these entry code sequences w.r.t. the n entry paths in less than $size_O = (\lceil \log_2 k \rceil + 1) \cdot m \cdot n$ space.

3.1.1 A Basic Approach for Compressing Paths

Recall that each entry in $\hat{E}(Tr_i)$ occupies $\lceil \log_2 k \rceil + 1$ bits and the exit entries of each vertex always start from 1, i.e., $v_s \rightarrow i$ with $i \geq 1$. Assume exit entries of a vertex share the same probability to appear in the trajectory, then the first bit of each $\lceil \log_2 k \rceil + 1$ bits (corresponding to an exit entry) has a higher probability to be 0. In other words, if we extract the first bit of m entries in each $\hat{E}(Tr_i)$, those bits form an $n \times m$ binary matrix with majority of bits being 0. This inspires us to propose BASICPATHCOMP, which uses one extra base binary sequence and auxiliary columns to represent the $n \times m$ binary matrix in a compressed form.

Fig. 4 plots an example to illustrate the main idea of BASICPATHCOMP. Assume we need to compress $\hat{E}(Tr_1)$, $\hat{E}(Tr_2)$, and $\hat{E}(Tr_3)$, with $k=7$ (i.e., $n=3$ and $m=6$). $\hat{E}(Tr_1)$, $\hat{E}(Tr_2)$, and $\hat{E}(Tr_3)$ occupy in total $\lceil \log_2 k \rceil \cdot m \cdot n = 54$ bits. BASICPATHCOMP first extracts the first bit of entry code in every entry code sequence (underlined bit) to form matrix M of 3×6 , and only maintains the remaining bits of entry code (non-underlined bits) in every $E(Tr_i)$ in compressed entry code sequences (i.e., $\hat{E}'(Tr_1)$, $\hat{E}'(Tr_2)$, and $\hat{E}'(Tr_3)$ in Fig. 4). It next forms 6-bits base binary sequence B where the first bit and the sixth bit are 1. Accordingly, an auxiliary matrix A is formed which contains m' ($=2$) columns, corresponding to the first and the sixth columns of M respectively. In total, the compressed entry code sequences,

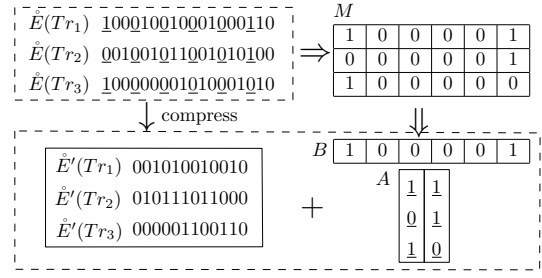


Fig. 4: An example of compressing entry codes.

B and A , occupy $\lceil \log_2 k \rceil \cdot m \cdot n + m + n \cdot m' = 48$ bits. In other words, if we assume β is the probability that B contains at least c 0s (with $c \geq 1$), then the size of n compressed entry code sequences has its $size_B \leq \lceil \log_2 k \rceil \cdot m \cdot n + m + (1 - \beta) \cdot m \cdot n$. Now we discuss the value of β . Let α be the probability that $M(i, j) = 1$ in the i -th row in M , then for any position j in B , the probability that $B[j] = 0$ is α^n . Therefore the probability that B contains at least c bits of 0 is

$$\beta = \sum_{x=c}^m \binom{m}{x} \alpha^{n \cdot x} (1 - \alpha^n)^{m-x}. \quad (1)$$

According to Eq. 1, when $\beta > \frac{1}{n}$, the approach BASICPATHCOMP can lead to space reduction. In addition, β decreases as $\frac{c}{m}$ gets larger. If we use one word (e.g. 32 bits) to store a column in A , when c is less than 10% of m , the probability that B has at least c 0s is very high, approaching 100%. Accordingly, we can physically store each column in A as a word for every 32 trajectories (i.e. $n = 32$). Then increasing the number of trajectories will not affect the compression ratio.

3.1.2 Improving Compression Using Multiple Bases

Each bit-0 in the base binary sequence B enables BASICPATHCOMP to achieve certain space saving. As n increases, the chance for all the n bits in j -th column of M to be 0s becomes smaller and hence the potential amount of space that can be saved might be reduced. Motivated by this observation, we propose an improved approach IMPRVPATHCOMP to partition n entry code sequences into q groups G_1, \dots, G_q of different sizes such that the matrix M of each group has many columns with only zero values. Instead of using a single base B , IMPRVPATHCOMP uses q bases B_1, \dots, B_q , where each B_x is used to compress entry code sequences in G_x . Then, the size for storing $\{\hat{E}(Tr_1), \dots, \hat{E}(Tr_n)\}$ by using IMPRVPATHCOMP becomes $size_I^{(q)} \leq \lceil \log_2 k \rceil \cdot m \cdot n + q \cdot m + \sum_{x=1}^q (m - c_x) |G_x|$, where base B_x for group G_x contains at least c_x bits of 0. Notice that when a group contains all m bits of 0, the corresponding base is empty.

Now, the problem of compressing $\{\hat{E}(Tr_1), \dots, \hat{E}(Tr_n)\}$ can be converted to decide a q and a partition of n trajectories into q groups such that $size_I^{(q)}$ is minimum, which can be divided into the following two sub-problems.

Sub-problem 1: Matrix transformation. We firstly adopt matrix reordering approach to transform the matrix M to a new matrix M' , such that M' is divided into left and right parts, where the right part only stores 0s. We assume the left part is distinguished from the right part by a set of boundary nodes $(i_1, j_1), \dots, (i_z, j_z)$ in M' such that $i_a \leq i_{a+1}$ and $j_a \leq j_{a+1}$, as marked by cross signs in Fig. 5(a).

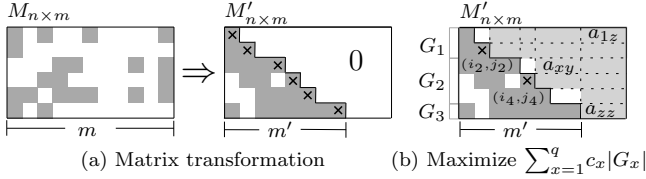


Fig. 5: Matrix transformation and partition.

To achieve this, we start examining the first row of matrix M . We choose a row with minimal number of bit-1s in M and then switch this row with the first row, e.g., in Fig. 5(a) the second row and the examining row (e.g. first row) of M need to be switched. Then, we examine columns in M . Let l be the smallest unexamined column number ($l = 1$ in this case). We start from the l -th column and keep checking every unexamined column corresponding to the first row in M till the last column. Let r capture the column that is currently examined with $r \in [l, m]$. If the r -th column in the first row is 1 (i.e. $M[1, r] = 1$), we switch l -th column with the r -th column of M and increase l value by one to indicate the first column has been examined. By this way, the first row is separated into two parts: the left parts containing only bits 1s and the right part containing only bits 0s. We then examine the submatrix M_{tmp} from the second row to the n -th row and from the l -th column to the m -th column. We repeat the above process until all rows in the matrix have been processed and we get the final transformed matrix M' . Fig. 5(a) shows two matrices before and after transformations. Only 0s appear in the right part in $M'_{n \times m}$, with $(1, 1)$, $(2, 2)$, $(3, 4)$, $(4, 5)$, $(5, 6)$, and $(6, 8)$ being the boundary nodes.

Theorem 1. The sub-problem of transforming $M_{n \times m}$ to $M'_{n \times m}$ is NP-hard.

We show the proof sketch in Appendix A.1.

Then minimizing space problem for multiple entry code sequences becomes the following problem of partitioning M' .

Sub-problem 2: Partitioning $M'_{n \times m}$ to q groups. Given a set of boundary nodes $I = \{(i_1, j_1), \dots, (i_z, j_z)\}$ in M' and an integer $q (> 1)$, we need to choose a set I_q of $q-1$ boundary nodes ($I_q \subseteq I$) to minimize $size_I^{(q)}$. This is equivalent to choosing I_q to maximize $\sum_{x=1}^q c_x |G_x|$, since both $\lfloor \log_2 k \rfloor \cdot m \cdot n$ and $q \cdot m$ are constant values for q . Here $\sum_{x=1}^q c_x |G_x|$ equals the covered areas in the right part with 0s in M' , e.g., the light grey area in Fig. 5(b) indicates the covered areas by choosing boundary nodes (i_2, j_2) and (i_4, j_4) .

Theorem 2. The sub-problem of partitioning $M'_{n \times m}$ to q groups is NP-hard.

We show the proof sketch in Appendix A.2.

Algorithm 2 shows the algorithm PARTITION, which costs $\mathcal{O}(qz)$ time since the main cost is for adjusting *gain* values for nodes in $I - I_{q-1}$ (see line 7). Here, gain of (i_x, j_x) , denoted as $gain(i_x, j_x)$, refers to the number of bits that a boundary node (i_x, j_x) could reduce (i.e. the areas that (i_x, j_x) could cover). For an M' with z boundary nodes, we can partition it into a set of small areas $a_{11}, \dots, a_{1z}, a_{22}, \dots, a_{2z}, \dots, a_{zz}$ (see Fig. 5(b)) with $gain(i_x, j_x) = \sum_{x=1}^i \sum_{y=j+1}^z a_{xy}$.

Theorem 3. Let OPT* and OPT be the covered areas in $M'_{n \times m}$ using the greedy algorithm PARTITION and the optimal solution, respectively, then $\frac{OPT}{OPT^*} < \frac{3}{2}$.

We show the proof sketch in Appendix A.3.

Algorithm 2: PARTITION.

Input: Gain values for every boundary node in $I = \{(i_1, j_1), \dots, (i_z, j_z)\}$ in $M'_{n \times m}$ in ascending order, an integer q ;

Output: A set $I_{q-1} (\in I)$ that are used for partition;

```

1  $I_{q-1} \leftarrow \emptyset$ ;  $count \leftarrow 1$ ;  $totalGain \leftarrow 0$ ;
2 while  $count < q$  do
3   Choose  $(i_x, j_x)$  with largest  $gain(i_x, j_x)$  from  $I$ ;
4    $totalGain \leftarrow totalGain + gain(i_x, j_x)$ ;
5    $I_{q-1} \leftarrow I_{q-1} \cup \{(i_x, j_x)\}$ ;
6   Remove calculated cells from  $M'$ ;
7   Adjust gain values for nodes in  $I - I_{q-1}$ ;
8    $count \leftarrow count + 1$ ;
9 return  $I_{q-1}$ ;
```

Then the algorithm IMPRVPATHCOMP invokes PARTITION using a q value from 1 to z and chooses the maximum $size_I^{(q)}$. Notice that, the above approach can be easily extended to the case where trajectories have variable lengths. We firstly rank trajectories in ascending order of their lengths. Then, for those with the same length, we use BASICPATHCOMP and IMPRV-PATHCOMP for further compression. The time for compressing consists of three parts, which are (i) scanning and generating entry path to $\{\hat{E}(Tr_1), \dots, \hat{E}(Tr_n)\}$ using $\mathcal{O}(mn)$ time, (ii) invoking MATRIX TRANSFORM using $\mathcal{O}(mn)$ time, and (iii) invoking PARTITION using $\mathcal{O}(q)$ time, where m is the average length of trajectories, n is the number of trajectories, and $q (\ll n)$ is the number of groups. Therefore, the time complexity for compressing entry path is $\mathcal{O}(mn)$.

Decompressing entry code sequences. The process of decompressing the j -th entry en_j in the entry path of a trajectory Tr_i is as follows. Let w_j be the entry code with length $\lfloor \log_2 k \rfloor + 1$ to represent entry en_j . The first step is to get the low order bits of w_j . We chop $\hat{E}'(Tr_i)$ and let its j -th segment between positions $1 + (j-1)\lfloor \log_2 k \rfloor$ and $j\lfloor \log_2 k \rfloor$ be the low order bits of w . The second step is to get the first bit of w_j . Assume that the trajectory Tr_i belongs to group G_x . If $B_x[j]$ equals to 0, the first bit of w_j is 0; otherwise, let $B_x[j]$ be the a th bit-1 in B_x and Tr_i correspond to the y th path in G_x , then the first bit of w_j is $A[a][y]$. The corresponding integer value of w_j is the entry value en_j . We develop COUNTBIT($B_x[1, j]$) to calculate a efficiently as follows. Let the first j bits in B_x occupy z bytes, i.e., $B_x[1, j] = c_1 c_2 \dots c_z$, where c_i is a byte and $z = \lceil \frac{j}{8} \rceil$. Let $C_f(c_i)$ be the summation of 1s in a byte c_i . Each byte corresponds to an unsigned char, and we can use a pre-computed mapping table to store the mapping pairs between every unsigned char and number of bit 1s in its byte [1]. The size of this mapping table is small (i.e., only $2^8 = 256$ bytes) and counting bit 1s of a word can be done in constant time [1]. Therefore, the decompressing operation can be done in constant time.

3.2 Error-Bounded Distance-Preserving Compression

Distance sequence $D(Tr)$ records a sequence of relative distances. We could adopt the well known Huffman encoding [3] to compress $D(Tr)$. That is to transform every relative distance $r(p_i)$ to a binary code w_i and to assign to the most frequently occurring $r(p_i)$ the shortest w_i , and the least frequently occurring $r(p_i)$ the longest w_i . It has been proved that Huffman encoding is an optimal solution in terms of compression ratio [3].

However, Huffman code is designed based on frequencies of relative distances instead of their values. For example, given two relative distances $r(p_1) < r(p_2)$, the Huffman code of $r(p_1)$ could be larger than that of $r(p_2)$. If we want to search relative distance values within a range $[r(p_1), r(p_2)]$, we have to scan $\hat{D}(Tr)$ and traverse the Huffman tree to get all relative distances in $\hat{D}(Tr)$. It causes a delay in answering some LBS queries that are based on distances.

Therefore, we aim to provide a distance-preserving encoding scheme (DP-encoding scheme for short), where each code w can be associated with a relative distance in $D(Tr)$. Given two relative distances $r(p_1) < r(p_2)$, let w_1 and w_2 be their corresponding compressed codes. A DP-encoding scheme guarantees $w_1 < w_2$. A straightforward way is to adopt the typical encoding scheme that approximates a relative distance $r(p_i)$ using an accumulation of some pre-selected values in a geometric sequence $\alpha = \frac{1}{2}, \frac{1}{4}, \dots, \frac{1}{2^e}$ and encode $r(p_i)$ to a code $w_i \in \{0, 1\}^{\ell_i}$, where w_i is a sequence of binary bits. Let $w_{i,j}$ be the j -th bit in w_i ($j \geq 1$). Any relative distance $r(p_i) \in (0, 1)$ could be represented as $r(p_i) = \sum_{j=1}^{\infty} w_{i,j} \alpha[j]$. When $r(p_i)$ is 0, we encode it to $(0)_2$.

Notice that, theoretically w_i could end in an *infinite* string of zeros. For instance, the relative distance 0.4 is encoded as $(011001100\dots)_2$. In this paper, we propose an accuracy of positioning η to bound the inaccuracy that could be caused by the distance compression.¹ Given the length $|e|$ of a path e and an error bound (i.e. accuracy of positioning) η , when $r(p) - \sum_{j=1}^{\infty} w_{i,j} \alpha[j] \leq \frac{\eta}{|e|}$, we use $\sum_{j=1}^{\infty} w_{i,j} \alpha[j]$ to approximately encode $r(p)$. For example, when $|e| = 500$ meters and $\eta = 1$ meter, $r(p) = 0.4$ could be encoded as $(0110011)_2$.

Although DP-encoding is error-bounded and preserves the relative distances, it brings disadvantages. If we store relative distances of a trajectory in one binary file (e.g., $(0, 0.5, 0.375, 0.75, 0.75)$ will be stored as $(010111111)_2$), we could not decode them as we do not know where to segment each binary code w_i , and we even cannot tell how many relative distances are stored in the compressed sequence. Can we expand w_i to w'_i , such that w'_i keeps the ability to describe relative distance and is recoverable, and the cost of such expansion is small?

Decodable DP-Encoding Scheme. If each relative distance is encoded into w_i of fixed length, the problem is solved. However, it wastes lots of space to store 0s which could be actually saved. In this work, we propose an expansion scheme based on *prefix condition* [8]. A list $w'_1 \dots w'_p$ of binary words *satisfies the prefix condition* iff $\forall 1 \leq i < j \leq p$, w'_i is not a prefix of w'_j and vice versa, and such list could be decoded easily. Therefore, given a list D with its relative distances encoded by DP-coding scheme into w_i s, we propose to expand w_i to a *prefix code* $w'_i = w_i v_i$ if necessary, such that w'_i s of D satisfy the prefix condition and can be decoded into $r(p_1), \dots, r(p_p)$ easily. As mentioned above, DP-coding will encode the relative distance list $0, 0.5, 0.375, 0.75, 0.75$ of our example D to $w_1 = (0)_2$, $w_2 = (1)_2$, $w_3 = (011)_2$, and $w_4 = (11)_2$. Here, w_1 is a prefix of w_3 , and w_2 is a prefix of w_4 and hence they do not satisfy the prefix condition. We need to expand w_1 and w_2 to make all four binary words satisfy the prefix condition.

Before we explain how to expand w_i to w'_i via appending v_i after w_i , we first analyze the DP-encode scheme $r(p_i) \rightarrow$

$w_i \in \{0, 1\}^{\ell_i}$, which has the following “good” property to make it expandable by using v_i with a small size.

Property 3.1. Given a relative distance $r(p_i) \in [0, 1)$ and the DP-encoding scheme $r(p_i) \rightarrow w_i \in \{0, 1\}^{\ell_i}$, any code w_i of a non-zero relative distance $r(p_i)$ must end with 1.

Property 3.1 inspires us to assign v_i a sequence of 0s so that $w'_i = w_i v_i$ becomes decodable. Below we propose an algorithm EXPANSION to generate w'_i . EXPANSION first uses a binary tree, called *DP-tree*, to express all DP-encoded words. In DP-tree, the left child node corresponds to label 0 while the right one corresponds to label 1; and the labels from the root to a black node represent a DP-encoded word. For k distinct relative distances, EXPANSION spends $O(k)$ time to scan them and to build a DP-tree. Given two black nodes n_1 and n_2 in DP-tree, let w_1 and w_2 be their corresponding binary words. If n_1 is an ancestor of n_2 , w_1 must be a prefix of w_2 . In other words, an internal black node (i.e. non-root, non-leaf node) must be such an ancestor that needs expansion as it must have at least one black descendant node. In the following, we assume internal node n_l is such an internal node and explain how EXPANSION expands n_l .

Started from node n_l , EXPANSION keeps visiting the left child (labelled 0) until it meets a descendant node n' of n_l without left child. EXPANSION inserts a node n_c as a child of n' and adds 0 as the label from n' to n_c . Property 3.1 guarantees the existence of n' as all the binary words corresponding to relative distances must end with 1. Let x be the number of 0s from n_l to n_c , and then EXPANSION appends x consecutive 0s after w_i to generate a decodable w'_i . The DP-tree after expansion is called *decodable DP-tree* (DDP-tree for short), in which every code corresponds to a path from the root to a leaf satisfying prefix condition. For example, Fig. 6(a) shows a DP-tree before expansion and Fig. 6(b) shows the DDP-tree after expansion. Using DDP-encoding, the binary code for $0, 0.5, 0.375, 0.75, 0.75$ is $(00100111111)_2$, which requires 11 bits. The accuracy of EXPANSION is guaranteed by Theorem 4.

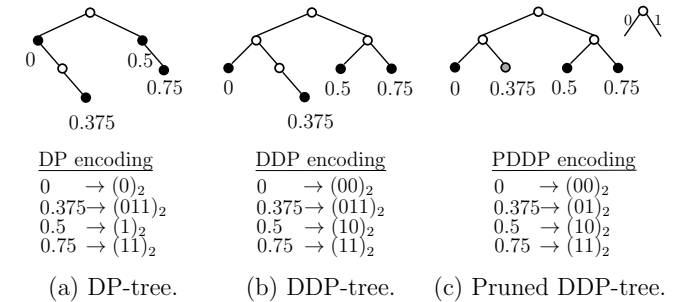


Fig. 6: Tree expansion and pruning.

Theorem 4. Let w be a code and w' be its expanded code using the algorithm EXPANSION. Then w' must be unique and shortest.

We show the proof sketch in Appendix A.4.

Given a DDP-tree, the decoding of the whole sequence $\mathcal{D}(Tr_i)$ is straightforward. It traverses the DDP-tree node by node as each bit is read from the input binary code, and the search for that particular distance is terminated when a left node is reached.

Shortening decodable codes by maximized pruning DDP-tree. We could further reduce the space by pruning long codes in DDP-tree. The main idea is that if a leaf node n_l has no sibling and its parent node n_p does not represent any distance (i.e., not black), it could be represented by its parent node n_p by copying labels

1. The latest report GPS accuracy is from 3.5 meters to 7.8 meters (please see <http://www.gps.gov/systems/gps/performance/accuracy/>).

of n_l to n_p ; thereafter n_l can be removed and n_p becomes a leaf node. We examine all the leaf nodes in DDP-tree until no new leaf node could be pruned to form a *pruned DDP-tree* (PDDP-tree for short). Then, every path in the PDDP-tree corresponds to a binary code, called a *pruned binary code*.

Fig. 6(c) shows the pruned DDP-tree and its PDDP encoding scheme. Using PDDP-encoding, the binary code for 0, 0.5, 0.375, 0.75, 0.75 is $(0010011111)_2$, which occupies only 10 bits. Our experimental results in Section 5 show that PDDP encoding scheme can compress relative distances in trajectories significantly. Its compression size is close to the size of using Huffman encoding, the one that can minimize the average code lengths [3].

In addition, each leaf node in a PDDP-tree is associated with subsequences in $\dot{D}(Tr)$. Then given a code, we can easily identify its corresponding subsequences in $\dot{D}(Tr)$ to avoid scanning $\dot{D}(Tr)$ from the beginning. For example, 0.5 is associated with position 3 (the starting position of subsequence 00 in $(0010011111)_2$). The time complexity of generating a DDP-tree (or PDDP-tree) is $O(p)$, and we only need to use $O(k+p)$ space to store distinct codes and positions associated with subsequences, where k is the number of distinct codes and p is the number of raw positional data.

PDDP codes help to achieve two main advantages. First, it helps to reduce the storage cost of $D(Tr)$. Our location compression using PDDP-encoding scheme can achieve a higher compression ratio. Second, PDDP-encoding scheme can preserve distance information in the binary code, which makes LBS query processing more efficient.

3.3 Temporal Compression

Using TED format, the i -th timestamp t_i in $T(Tr)$ corresponds to the i -th bit-1 in the time flag bit-string $\dot{T}(Tr)$. Let the j -th bit in $\dot{T}(Tr)$ be the i -th bit-1, then $T(Tr)[i]$ matches $\dot{T}(Tr)[j]$.

Compressing consecutive unchanged time intervals. Given a trajectory Tr and any three embedded data (t_i, e_i, d_i) , $(t_{i+1}, e_{i+1}, d_{i+1})$, and $(t_{i+2}, e_{i+2}, d_{i+2})$, if $t_{i+2} - t_{i+1} = t_{i+1} - t_i$, we could remove t_{i+2} from $T(Tr)$. In order to record a remaining t_i in the i -th timestamp in the compressed time sequence, we store remaining timestamps in a sequence $T'(Tr)$ where each element in $T'(Tr)$ is a pair (i, t_i) . For example, $T(Tr_1) = \{00:00:00, 00:01:30, 00:03:00, 00:04:30, 00:06:01\}$. The time intervals for the first four timestamps are the same (i.e. equals to 00:01:30), so $T'(Tr)$ is $\{(1, 00:00:00), (2, 00:01:30), (5, 00:06:01)\}$. It implies that two missing pairs follow the same time interval as their previous timestamps. Finally, we store the compressed codes of the trajectory Tr_1 as shown in Fig. 7.

$$\begin{array}{l}
 \dot{E}'(Tr_1): \quad \underline{00} \quad \underline{10} \quad \underline{10} \quad \underline{01} \quad \underline{00} \quad \underline{10} \quad + \quad B \begin{array}{|c|c|c|c|} \hline \square & \square & \dots & \square \\ \hline \end{array} \\
 \dot{T}(Tr_1): \quad \begin{array}{cccccc} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array} \quad \begin{array}{|c|c|} \hline A \begin{array}{|c|c|} \hline \square & \square \\ \hline \end{array} \\ \hline \end{array} \\
 \dot{D}(Tr_1): \quad \underline{00} \quad \quad \underline{10} \quad \underline{01} \quad \underline{11} \quad \underline{11} \\
 T'(Tr_1): \quad (1, 00:00:00), (2, 00:01:30), (5, 00:06:01)
 \end{array}$$

Fig. 7: Compressed codes for trajectory Tr_1 .

In addition, when an object stops at a certain place for a while, the TED format requires large space to store duplicate 1s in \dot{T} , 0s in \dot{E} , and same distance codes in \dot{D} . In order to

avoid it, we preprocess each trajectory as follows. Given a set of consecutive raw positional data corresponding to the same location, i.e., (t_1, x, y) , (t_2, x, y) , \dots , (t_i, x, y) , we only keep (t_1, x, y) and (t_i, x, y) but remove those in between. In this way, we save spaces without losing anything. The time complexity is linear to the number of raw positional data.

Compression multiple $\dot{T}(Tr)$ and $\dot{D}(Tr)$. Given a set of time flag sequences $\dot{T}(Tr_1), \dots, \dot{T}(Tr_n)$, they construct a bitmap. Many Bitmap compression algorithms have been proposed to compress bitmap in a bitmap index to save space [2], [5], [7], [22], which require very little effort to compress and decompress. We adopt the recently reported Partitioned Word-Aligned Hybrid (PWAH) compression algorithm in [22] to compress $\dot{T}(Tr_1), \dots, \dot{T}(Tr_n)$. We use the same way to compress $\dot{D}(Tr_1), \dots, \dot{D}(Tr_n)$.

4 DIRECT QUERY ON COMPRESSED TRAJECTORIES

In this section, we show that compressed TED-trajectories can easily associate the three dimensions T, E, and D of trajectories together to support queries in the road network efficiently. Ideally, we hope to query compressed trajectories directly, i.e. without fully decompressing the trajectories, or in the ideal case without decompressing the trajectories at all.

As we know, many LBS applications require different LBS queries [16], [26], [27]. First, we list two types of primitive operations that are fundamental functions to support LBS related applications in Section 4.1. Next, we propose index structures to efficiently support these primitive operations in Section 4.2. Finally, we demonstrate the flexibility of primitive operations in supporting LBS queries by showcasing four very typical types of LBS queries in Section 4.3.

4.1 Primitive Operations

There are two types of primitive operations. One is for *transformations* from $\dot{T}(Tr)$ to $\dot{E}(Tr)$ and $\dot{D}(Tr)$, respectively. The other is for *mapping* between compressed trajectories and their logical representations in the form (t, x, y) (see Fig. 8). In the following, we discuss the implementation of these primitive operations.

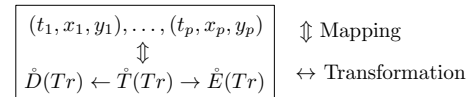


Fig. 8: Primitive operations.

4.1.1 Transformation Operations among TED codes

As we know, there is a one-to-one matching between $\dot{T}(Tr)$ and $\dot{E}'(Tr)$. Here, $\dot{E}'(Tr)$ is the compressed entry code sequence of Tr where the first bit of each entry code in $\dot{E}'(Tr)$ is skipped. The j -th bit in $\dot{T}(Tr)$ corresponds to the j -th entry in the entry path. Since each entry is encoded as a $\lfloor \log_2 k \rfloor$ -bit in $\dot{E}'(Tr)$, the j -th entry starts at position $(j-1)\lfloor \log_2 k \rfloor + 1$ in $\dot{E}'(Tr)$. The matching between $\dot{T}(Tr)$ and $\dot{D}(Tr)$ is not one-to-one. However, for the i -th location in $\dot{D}(Tr)$, there are exactly i bits of 1 in $\dot{T}(Tr)[1, j]$ and hence we can also associate each i with j . Below we elaborate transformation operations. As all these operations are

for one given trajectory Tr , we skip Tr for brevity (e.g., T means $\hat{T}(Tr)$).

- **TtoE**($\hat{T}, j, \hat{E}', B, A$) returns the binary code of the j -th entry in the compressed entry path in the form (\hat{E}', B, A) given the j -th bit in \hat{T} . This operation first uses the process of decompressing entry paths (see Section 3.1.2) to get the code w_j . If w_j is non-zero, it returns w_j directly. Otherwise, it keeps decompressing its previous code w_{j-1} for the $(j-1)$ th entry until a non-zero code is reached. It then returns this non-zero code. This operation can be done in constant time $\mathcal{O}(C)$, where C is the average number of embedded data within an edge.

- **TtoD**(\hat{T}, j, \hat{D}), given the j -th bit in \hat{T} , returns the start position pos in \hat{D} if $\hat{T}[j]$ is a bit of 1 (i.e. it corresponds to a relative distance code at position pos in \hat{D}); -1 otherwise. If $\hat{T}[j]$ is a bit of 1, we first need to locate the bits corresponding to the i -th relative distance in \hat{D} . Recall that for the i -th location in \hat{D} , there are exact i bits of 1 in $\hat{T}[1, j]$. Consequently, we count the number of bit 1s in the first j bits (including the j -th bit) in \hat{T} . We then decode the binary code \hat{D} by traversing the adopted encoding tree (e.g. DDP-tree or PDDP-tree) and return the start position of the i -th code. Since each position of \hat{D} records the relative distance to a vertex in the road network, we could guarantee that the error bound of **TtoD** is the same as the accuracy of positioning η . Let ℓ_z be the longest length of the code in the adopted encoding tree and p be the length of \hat{D} , then this operation requires on average $\mathcal{O}(\frac{p}{2}\ell_z)$ time.

4.1.2 Mappings Operations

For a collection of trajectories with logical format (t, x, y) , they are physically stored as compressed codes. Therefore, we need to provide a few primitive mapping operations between the logical format and their compressed codes.

- **time2T**(t, \hat{T}). Given a time slot t , this operation returns its mapped time positions j and j' in \hat{T} such that their corresponding timestamps t_i and t_{i+1} satisfy $t_i \leq t \leq t_{i+1}$. For the given time slot t , we first binary search t in $T'(Tr)$ and get two pairs (i_x, t_x) and (i_y, t_y) such that $t_x \leq t \leq t_y$. If i_x and i_y are not adjacent numbers (i.e. $i_y > i_x + 1$), it means each timestamp $t_{x'}$ ($t_x < t_{x'} < t_y$) in $T'(Tr)$ follows the same time interval of $t_I = t_x - t_{x-1}$. Therefore, we calculate $i_{x'} = i_x + \lfloor \frac{t - t_x}{t_I} \rfloor$ such that its corresponding timestamp $t_{x'} = t_x + \lfloor \frac{t - t_x}{t_I} \rfloor \leq t$ is the closest timestamp to t . Then we locate the j -th bit in \hat{T} such that its first j bits contain $i_{x'}$ bit 1s. Similar to the function **COUNTBIT** (see Section 3.1.2), let \hat{T} occupy several bytes $c_1 c_2 \dots$ and $C_f(c_i)$ be the summation of 1s in a byte c_i . We devise **LOCATEBIT**($i_{x'}, \hat{T}$) to calculate j in constant time as follows. We start from c_1 and find a byte c_h such that $i' = \sum_{i=1}^{h-1} C_f(c_i) \leq i_{x'} \leq \sum_{i=1}^h C_f(c_i)$. We then scan bits in the h -th byte and locate the a th bit such that the first $j = 8(h-1) + a$ bits in $\hat{T}(Tr)$ contain $i_{x'}$ bits 1. Then, starting from j , we find the next bit 1 at position j' in \hat{T} , and return j and j' . The operation **time2T**(t, \hat{T}) costs $\mathcal{O}(\log l)$, where l is the length of $T'(Tr)$.

- **edge2E**($v_s, en, v, \hat{E}', B, A$). Given an edge represented by $v_s \rightarrow en$ in the road network, and a compressed entry path (\hat{E}', B, A) starting from v , this operation returns -1 if trajectory Tr does not pass the input edge; otherwise it returns an integer j which means $v_s \rightarrow en$ is the j -th edge passed by Tr . To support this mapping, we recover the edges passed by Tr via decompressing (\hat{E}', B, A) . Let v' be the starting vertex, $i = 1$ represent the current iteration, and e' be the i -th decompressed

entry. If $v' \rightarrow e'$ equals $v_s \rightarrow en$, i is returned to complete the mapping. Otherwise, i is increased by 1, v' is reset to the ending vertex of edge $v' \rightarrow e'$, and e' is reset to the next decompressed entry. If none of the edges passed by Tr could match $v_s \rightarrow en$, -1 is returned. The time complexity of this operation is $\mathcal{O}(\frac{m}{2})$ where m is the average length of a trajectory.

4.2 An Index to Support Efficient Primitive Operations

Based on above description, we understand both **edge2E** and **TtoD** have to scan $\hat{E}'(Tr)$ and $\hat{D}(Tr)$ from beginning, respectively. In order to avoid it, we hope to index some vertices in a road network so that we can locate an edge in $\hat{E}'(Tr)$ via starting from an intermediate vertex. In the following, we propose an index structure to tackle this problem. The main idea is to partition the road network into sub-regions so that a trajectory actually crosses one or several sub-regions.

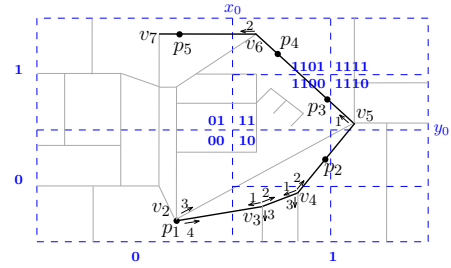


Fig. 9: Partitioning a road network.

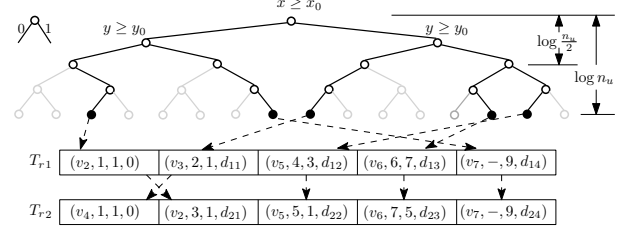


Fig. 10: Partition tree for Tr_1 and Tr_2 .

Road network partition. For ease of presentation, we use grids to partition a road network, while our technique can be applied to other partition approaches. We partition the network into four regions via x_0 and y_0 that correspond to binary codes 00, 01, 10, and 11, respectively, as shown in Fig. 9. For each region, we can further partition it into four sub-regions using the same encoding scheme. Let d be the number of iterative partitioning steps, then there are totally 2^{2d} sub-regions and the code for each sub-region occupies 2^d bits. Given a vertex v of a trajectory Tr , it can be easily located into a sub-region S_i by comparing its 2D location with x_0 and y_0 . For example, vertex v_5 locates in the region 1110. If we pre-process the trajectory Tr such that v_i refers to the first vertex in S_i that is reached by Tr , we can actually scan Tr from v_i instead of the first vertex of Tr when we need to locate v in trajectory Tr . We name the first vertex in sub-region S_i reached by Tr as a *representative vertex* w.r.t. Tr .

Partition tree. We use a partition tree to maintain these representative vertices w.r.t. their trajectories. The partition tree is a binary tree, with left sub-tree standing for 0 (i.e. $x < x_0$ for the first level or $y < y_0$ for the second level) and right sub-tree standing for 1 (i.e. $x \geq x_0$ or $y \geq y_0$), as shown in Fig. 10. The label sequence from the root to a leaf corresponds to a region code U . Each leaf node stores a list of tuples, with each in the form of (v, j, i', d_{ac}) corresponding to a trajectory Tr . Here, v is

TABLE 1: Major LBS Queries.

Query types	Queries	Related primitive operations	Time complexity
Basic	$where(Tr, t)$	$time2T, TtoE$	$\mathcal{O}(\frac{m}{2^{2d+1}})$
	$when(Tr, x, y)$	$TtoD$	$\mathcal{O}(\frac{p}{2^{2d+1}} \ell_z)$
Range	$distance(Tr, t_1, t_2)$	$time2T, TtoE$	$\mathcal{O}(\frac{m}{2^{2d+1}})$
	$howlong(Tr, x_1, y_1, x_2, y_2)$	$TtoD$	$\mathcal{O}(\frac{p}{2^{2d+1}} \ell_z)$
Aggregation	$count(Tr_{set}, x, y, r)$	$TtoD$	$\mathcal{O}(\frac{p}{2^{2d+1}} \ell_z)$
General	$kNN(Tr_{set}, x, y, t_1, t_2)$	$time2T, TtoE, TtoD$	$\mathcal{O}(\frac{m}{2^{2d+1}} + \frac{p}{2^{2d+1}} \ell_z + c \cdot n \cdot \log k)$
	$window(Tr_{set}, x_1, y_1, x_2, y_2, t_1, t_2)$	$time2T, TtoE, TtoD$	$\mathcal{O}(\frac{m}{2^{2d+1}} + \frac{p}{2^{2d+1}} \ell_z + c \cdot n)$

Algorithm 3: Finding previous representative vertex.

Input: An edge (v_s, en) , start vertex v of a trajectory Tr_x , and a partition tree Pt ;
Output: Tuples (v, j, i', d_{ac}) corresponding to v_s if (v_s, en) is an edge in Tr , null otherwise;
1 Traverse Tr to its leaf node n_l using v_s ;
2 Scan the list pointed by n_l and return tuples (v, j, i', d_{ac}) corresponding to the trajectory Tr_x ;

the representative vertex for the region with code U w.r.t. Tr , j indicates that the edge started from v is the j -th entry in Tr , and d_{ac} is the accumulative distance traveled along Tr from the beginning to v . Let $j' < j$ such that $\dot{T}(Tr)[j'] = 1$ and meanwhile $\forall j'' \leq j' \leq j, \dot{T}(Tr)[j''] = 0$. Then, i' refers to matching position of relative distance corresponding to $\dot{T}(Tr)[j']$, which is also stored in the partition tree. i' allows us to scan $\dot{D}(Tr)$ from position i' instead of the beginning when we need to locate the relative distance w.r.t. $\dot{T}(Tr)[j]$. In total, the partition tree stores 2^{2d} vertices and $n \cdot 2^{2d}$ tuples. Notice that in order to balance the overall load performance for a given set of trajectories, we can determine the parameters x_0 and y_0 according to the trajectories. Fig. 10 shows an example partition tree. Let Tr_1 be the trajectory that crosses vertices v_2, v_3, v_4, v_5, v_6 , and v_7 , and Tr_2 be another trajectory that crosses vertices v_4, v_3, v_2, v_5, v_6 , and v_7 . We can see from Fig. 9 that vertices v_3 and v_4 locate in the same sub-region 1000. For sub-region 1000, vertex v_3 is the representative vertex w.r.t. Tr_1 since it is the first vertex in Tr_1 that arrives in this region, whereas v_4 is the representative vertex w.r.t. Tr_2 . We store the set of prefixes for each depth in perfect hash table, e.g. $d_1=(0,1)$, $d_2=(00,01,10,11)$, $d_3=(001,011,100,110,111)$, $d_4=(0010,0111,1000,1101,1110)$.

Lemma 1. Looking up a representative vertex in a partition tree costs $\mathcal{O}(\log \log n_u)$ time, where n_u is the number of nodes in the partition tree.

Based on this partition tree, we can easily find a vertex close to the given edge (v_s, en) to accelerate the location of (v_s, en) w.r.t. a trajectory $\dot{T}(Tr)$. Algorithm 3 shows the detail. Given a vertex v_s , it first traverses the tree to locate v_s to a leaf node (i.e., a sub-region S_j) and the labels from the root to the leaf node correspond to the sub-region code U of S_j (line 1) and then traverses the associated list (line 2). Since we are interested in the case that the input edge (v_s, en) belongs to the trajectory $\dot{T}(Tr)$, we can guarantee that there must exist a representative vertex v' having the same region code as v_s . The vertex v' could be v_s or some other vertex. If (v_s, en) does not belong to $\dot{T}(Tr)$, the algorithm returns null without traversing the trajectory. The time complexity of Algorithm 3 is $\mathcal{O}(\log \log 2^{2d}) = \mathcal{O}(\log 2d)$.

Answering primitive operations efficiently. Now using Algorithm 3 based on the partition tree, we can support the operations

$edge2E$ and $TtoD$ efficiently.

For the operation $edge2E(v_s, en, v, \dot{E}'(Tr), B, A)$, we get the range code of v_s and invoke Algorithm 3 to get the representative vertex v w.r.t. Tr . Instead of traversing the compressed entry path from its start vertex v_s , we start from v . The time complexity of this operation depends on the depth d of iterative partitions in the road network. Since v is a representative vertex of a sub-region and there are totally 2^{2d} sub-regions, the time complexity for the mapping operation $edge2E$ is reduced from $\mathcal{O}(\frac{m}{2})$ to $\mathcal{O}(\frac{m}{2^{2d+1}})$.

For the operation $TtoD(\dot{T}(Tr), j, \dot{D}(Tr))$, in order to avoid traversing $\dot{D}(Tr)$ from its beginning when the j -th bit in $\dot{T}(Tr)$ is 1, we want to locate a start position i'_x that we can decode $\dot{D}(Tr)$ correctly. We binary search the array of tuples w.r.t. Tr pointed by the partition tree using j and get tuple (v, j_x, i'_x, d_{ac}) , where j_x is the closest integer to j ($j_x \leq j$). Since i'_x is the start position of the relative distance corresponding to $\dot{T}(Tr)[j_x]$, we start from i'_x and traverse the adopted encoding tree (e.g. Huffman, DDP, or PDDP tree). Then the operation returns the start position of the $(j - j'' + 1)$ th decoded relative distance. The time complexity for this operation is reduced from $\mathcal{O}(\frac{p}{2}) \ell_z$ to $\mathcal{O}(\frac{p}{2^{2d+1}} \ell_z)$.

4.3 Answering Major LBS Queries

Given a trajectory Tr stored as a TED format $(v, \dot{E}(Tr))$, $\dot{D}(Tr), (t_1, \dot{T}(Tr))$, many of LBS queries can be supported via primitive operations with little decompression. Below we list seven queries which roughly represent four types of typical LBS queries, as summarized in Table 1. We also list in Table 1 the main primitive operations that can be used to support those queries.

• **where(Tr, t) query.** A query $where(\dot{T}(Tr), t)$ returns the 2D location (x, y) where the object locates at time t in the trajectory $\dot{T}(Tr)$.

We first locate the sub-path p_s that the object locates from time stamps t_i to t_{i+1} with $t_i \leq t \leq t_{i+1}$ as follows. We use $time2T(t, \dot{T}(Tr))$ to get the mapping time position $[j, j']$ in $\dot{T}(Tr)$. Then we use $TtoE(\dot{T}(Tr), j, \dot{E}(Tr))$ to get the code w and translate it to the corresponding entry en_j . We binary search the array of representative vertices w.r.t. $\dot{T}(Tr)$ and use $\mathcal{O}(2d)$ time to locate the tuple (v', j_s, i', d_{ac}) with the largest position $j_s (\leq j)$. We then traverse the entry path from the representative vertex v' to en_j and get the start vertex v_j of en_j . Let p be the path from vertex v' to v_j . Similarly, we get the edge $(v_{j'}, en_{j'})$ and the sub-path p_s from v_j to $v_{j'}$. Then, the object should locate at $\frac{t-t_i}{ut} |p_s|$ from the start vertex v_s along the path p_s . We finally locate this position in the road map and get the corresponding location (x, y) as follows. According to data structure EdgeGeometry in the road network, we know that p_s consists of a consecutive linear segments s_1, \dots, s_q . Then we can get a linear segment s_w that the object locates in at time t , i.e. $\sum_{j=1}^{w-1} |s_j| \leq |p_s| \leq \sum_{j=1}^w |s_j|$. Let $v'_s = (x_s, y_s)$ and $v'_e = (x_e, y_e)$ be the start and end points of the linear segment s_w . The 2D location (x, y) can be derived, as expressed in Eq. 2:

Algorithm 4: Calculating *when* query

Input: Adjacent locations rd_1, rd_2 in edges e_1, e_2 at time stamps t_1 and t_2 , respectively; Current location rd in the edge e ;

- 1 **if** $e_1 = e_2 = e$ **then**
- 2 $when \leftarrow t_1 + \frac{r-rd_1}{rd_2-rd_1} \cdot ut$;
- 3 **else if** there is a path p_1 in between e_1 and e && $e = e_2$ **then**
- 4 $when \leftarrow t_2 - \frac{(rd_2-rd_1) \cdot |e|}{(1-rd_1) \cdot |e_1| + |p_1| + rd_2 \cdot |e_2|} \cdot ut$;
- 5 **else if** there is a path p_1 in between e_1 and e && a path p_2 in between e and e_2 **then**
- 6 $when \leftarrow t_1 + \frac{(1-rd_1) \cdot |e_1| + |p_1| + r \cdot |e|}{(1-rd_1) \cdot |e_1| + |p_1| + |p_2| + rd_2 \cdot |e_2|} \cdot ut$;
- 7 **return** $when$;

$$\begin{cases} x = x_s + \frac{|p_s| - \sum_{j=1}^{w-1} |s_j|}{|c|} (x_e - x_s), \\ y = y_s + \frac{|p_s| - \sum_{j=1}^{w-1} |s_j|}{|c|} (y_e - y_s). \end{cases} \quad (2)$$

The time complexity of answering $where(Tr, t)$ query is mainly on traversing path from the representative vertex to the edge, which on average is $\mathcal{O}(\frac{m}{2^{2d+1}})$.

• **when**(Tr, x, y) **query**. It returns the timestamp that the object locates at (x, y) in the trajectory Tr .

For the given location (x, y) , we first utilize the popular map-location operation *mapmatch* to locate (x, y) to its nearest edge (v_s, en) in the road network. Notice that, the position (x, y) might not appear in any edge, so we let $when(Tr, x, y)$ return false if the distance from (x, y) to the edge (v_s, en) is greater than a threshold η . If (x, y) appears in the edge (v_s, en) , we check if trajectory Tr contains the edge (v_s, en) using Algorithm 3. If so, we get the leaf node (v, j, i', d_{ac}) of the representative vertex corresponding to Tr and traverse $\hat{T}(Tr)$ from the j -th entry. We then invoke $TtoD(\hat{T}(Tr), j, \hat{T}(Tr))$ to get positions of two adjacent relative distances rd_1, rd_2 and their edges e_1, e_2 using $\hat{T}(Tr)$. Let the time stamps for these two adjacent locations be t_1 and t_2 , respectively. Then $when(Tr, x, y)$ can be calculated easily by assuming that the object moves with constant velocity between t_1 and t_2 by using Algorithm 4. Answering *howlong*(Tr, x_1, y_1, x_2, y_2) query can be answered using $when(Tr, x_1, y_1) - when(Tr, x_2, y_2)$.

• **distance**(Tr, t_1, t_2) **query**. It returns the distance that an object runs from t_1 to t_2 along the trajectory Tr . It can be easily answered using $where(Tr, t_2) - where(Tr, t_1)$.

• **howlong**(Tr, x_1, y_1, x_2, y_2) **query**. It returns how long it takes the object to travel from location (x_1, y_1) to location (x_2, y_2) along Tr . It can be easily answered using $when(Tr, x_1, y_1) - when(Tr, x_2, y_2)$.

• **count**(Tr_{set}, x, y, r) **query**. It returns the number of time-stamped locations in a set of trajectories Tr_{set} along the path R centered by (x, y) with radius r .

It locates (x, y) to the j -th entry in $\hat{E}(Tr_v)$ w.r.t. a trajectory $\hat{E}(Tr_v)$. Then the algorithm uses j to get the i -th relative distance $r(p_i)$ in $\hat{D}(Tr_v)$. If $r(p_i)$ locates in the path range R , the algorithm adds the count number by 1. The final count number is returned after all trajectories in Tr_{set} are processed.

Recall that $\hat{D}(Tr_v)$ can be encoded using DDP-encoding scheme or PDDP-encoding scheme. Since they preserve distances in their corresponding codes, we could efficiently get the i -th relative distance in $\hat{D}(Tr_v)$ using DDP encoding scheme (the

approach based on PDDP encoding scheme is the same). Given the central location (x, y) and radius r , we can calculate a relative distance range $[a, b]$ along the edge in the road network. According to Lemma 2, relative distances in both DDP and PDDP trees are stored in order, i.e., for any two relative distances r_1 and r_2 , the path for r_1 code is always in the left to the path for r_2 .

Lemma 2. Values of leaf nodes in both DDP-tree and PDDP-tree are sorted in ascending order.

Proof. Given two relative distances r_1 and r_2 with $r_1 < r_2$, let w_1 and w_2 be their encoded codes using DDP-encoding and p be the longest path that they share in the DDP tree. Let w be the corresponding code of path p , and $v = w[1]\alpha[1] + \dots + w[|w|]\alpha[|w|]$. Then it must satisfy $r_1 - v < \alpha[|w| + 1] < r_2 - v$, otherwise p is not their longest sharing path. Then the bit for $w_1[|w| + 1]$ should be 0 and that for $w_2[|w| + 1]$ should be 1. Therefore, the path corresponding to w_1 is to the left of the path w.r.t. w_2 . Similarly, values in a PDDP tree are sorted in ascending order. \square

Therefore, instead of decompressing the binary sequence $\hat{D}(Tr_v)$, we only need to check if there are paths in between paths corresponding to a and b . The time complexity for this checking operation is $\mathcal{O}(\log n')$, where n' is the number of nodes in a DDP or PDDP tree. Since the number of nodes in PDDP tree is less than the number in DDP tree, we could improve the performance by using PDDP tree.

• **kNN**(Tr_{set}, x, y, t_1, t_2) **query**. It returns k nearest trajectories in Tr_{set} that are close to a given point (x, y) and active during a given period of time from t_1 to t_2 .

We first use *time2T* and *TtoE* to locate entry paths in Tr_{set} for the given time interval $[t_1, t_2]$ and get entries in $\hat{E}(Tr_i)[j_1, j_2]$ for each $Tr_i \in Tr_{set}$. Let Can be the candidate set which keeps k trajectory segments $\hat{E}(Tr_i)[j_1, j_2]$ and d_k be the k -th minimum distance between query point (x, y) and a candidate trajectory segment in Can . For each Tr_i , we calculate the maximal distance d' from vertices along the path $\hat{E}(Tr_i)[j_1, j_2]$ to query point (x, y) . If d' is larger than the current k -th best true distance d_k , we can safely prune this trajectory segment in Tr_i since it cannot be closer to (x, y) than any of the k candidates preserved in Can . Otherwise, we include it into Can to replace the candidate with the largest distance (i.e. d_k) to (x, y) and change d_k to the maximal distance in the new candidate set Can . In the refining phase, we locate positions for each candidate trajectory segment by invoking *TtoD*. We then calculate the true distance between each candidate segment and the query point, and get segments with k smallest true distances to (x, y) . The time complexity is $\mathcal{O}(\frac{m}{2^{2d+1}} + \frac{p}{2^{2d+1}} \ell_z + c \cdot n \cdot \log k)$, where c is the number of timestamped positions in between t_1 and t_2 .

• **window**($Tr_{set}, x_1, y_1, x_2, y_2, t_1, t_2$) **query**. It returns all trajectories in Tr_{set} that overlap with window $(x_1, y_1), (x_2, y_2)$ and active during a given period of time from t_1 to t_2 .

Same as *kNN*, we first get entries in $\hat{E}(Tr_i)[j_1, j_2]$ for each $Tr_i \in Tr_{set}$. We then use *TtoD* to get the timestamped position (x_a, y_a) (and (x_b, y_b)) corresponding to $\hat{E}(Tr_i)[j_1]$ (and $\hat{E}(Tr_i)[j_2]$). We estimate the moving area of this trajectory segment using (x_a, y_a) and (x_b, y_b) . If this area overlaps with the query window, we calculate the actual answer of window query. The time complexity is $\mathcal{O}(\frac{m}{2^{2d+1}} + \frac{p}{2^{2d+1}} \ell_z + c \cdot n)$.

TABLE 2: Road network data sets.

Data sets	# of vertices	# of edges	# of entries per vertex	Data distribution of exit entries		
Singapore	20,801	55,892	Average 2.68 (from 1 to 7)	(≤ 3): 84.8%	($= 4$): 14.4%	(> 4): 0.7%
Beijing	171,504	433,391	Average 2.52 (from 1 to 6)	(≤ 3): 90.5%	($= 4$): 9.1%	(> 4): 0.3%

TABLE 3: Trajectory data sets.

Data sets	Storage of raw data	# of trajectories	# of edges per trajectory	# of time-stamped locations per trajectory
Singapore Taxi	34.9GB	553,414	Average 2,985 (from 1 to 31,478)	Average 997 (from 2 to 45,974)
Beijing Taxi	348.4MB	8,911	Average 8,127 (from 1 to 9,015)	Average 1,104 (from 1 to 1,788)

5 EXPERIMENTS

In this section, we conducted extensive experiments on two real data sets to demonstrate our techniques.

Data Set. The experiments were based on two real data sets namely *Beijing data set* and *Singapore data set*.

- Singapore data set. It was from a major taxi company in Singapore, containing trajectories generated by about 15,000 taxis in September 2011. Each taxi reported its location every 30 to 180 seconds.
- Beijing data set. It was downloaded from Microsoft research website, which contains a one-week trajectories of 10,357 taxis.² Taxis reported their locations from every 1 second to about 4 minutes.

Table 3 shows the detailed information of these trajectories. Such trajectories are mainly generated by motor vehicles (e.g. cars, taxis, buses, and etc.) that report their timestamped locations regularly based on the installed hardware, like GPS. We found that in both Singapore and Beijing data sets, large percentage of data have fixed sampling rates (time intervals). For example, Singapore and Beijing trajectories contain **41.58%** and **57.76%** timestamped positions with fixed time intervals, respectively.

We also did synthetic analysis by changing the timestamps of each position in both Singapore and Beijing datasets. We randomly change the timestamps and set the ratio of data with fixed time interval from 0 to 100%.

Comparison algorithms. We compared our work with following state-of-the-art algorithms.

- PRESS [20]. PRESS is the state-of-the-art approach for supporting queries on compressed trajectories in road networks and the closest work to ours.³
- Ext.SDM [17]. Ext.SDM is the latest representative trajectory compression that supports a generalized in-network trajectory data model, however it does not support queries.
- WinZip. It is a widely used compression tool.
- SPNET [11]. SPNET is an efficient shortest-path compression of trajectories to achieve a compact index structure, which is similar to PRESS.
- SharkDB [23]. SharkDB uses the delta encoding scheme to encode the P-frame points by calculating the difference between each P-frame and its closest prior I-frame point, which supports compressing trajectory data effectively.

All the algorithms were implemented using g++ on linux. The experiments were run on a 64-bit PC with an Intel 2.10GHz Quad Core i3 CPU and 5.8GB memory with a 485.8GB disk, running on ubuntu 15.04.

2. <http://research.microsoft.com/apps/pubs/?id=152883>

3. Thanks the authors of PRESS for sharing their source code.

5.1 Compression

(i) Compression ratios. Table 4 shows the comparison of TED compression with PRESS compression, Ext.SDM, WinZip, SPNET, and SharkDB. Both WinZip and SPNET are lossless compressions, whereas the others are error-bounded trajectory compressions. PRESS achieved a low compression ratio even under large error thresholds (i.e. Time Syn. Network Dis. (TSND)=100 meters and Network Syn. Time Dif. (NSTD) = 100 seconds), and it was not comparable with WinZip. SPNET provided the lowest compression ratio, whereas Ext.SDM and SharkDB achieved compression ratios that are better than PRESS but worse than TED. TED compression achieved a high compression ratio with an error bound of 1 meter on both data sets. When the error bound increased (e.g. η from 1m to 4m), the compression ratios of TED compression also increased. Please note that such increasing is not significant, so we can use a small error bound to achieve a high compression ratio.

(ii) Compression ratios of T, E, and D. Table 5 shows that the compression ratios of E were greater than 10 on both data sets. When entry paths were partitioned to 6 groups (i.e., $q = 6$ when partitioning matrix M), on average 70.02% bits in each base B were 0s, so the compression ratio was improved more than 1.21 times on Singapore data set. Compression ratio of D was higher than 5 for Singapore data set and 4.6 for Beijing data set even for a small η ($= 1$ meter). In addition, compression of T achieved very good compression ratios since most of real trajectories were reported in regular time interval, and many pairs of time flag and their corresponding timestamps could be reduced.

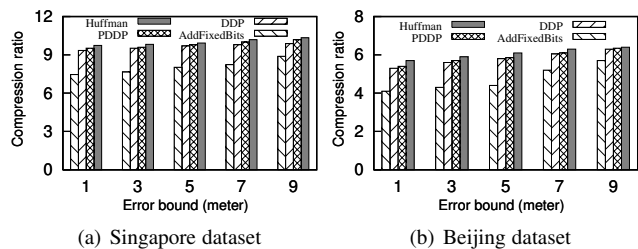


Fig. 11: Effect of error bounds using TED compression.

We also evaluated the compression ratio of D using different error bounds. Fig. 11 shows that increasing error bound will not affect the compression ratio too much. Therefore, our approach can achieve a high compression ratio as well as good accuracy.

Recall that the compression ratio of time is highly dependent on the assumption of fixed time intervals. Therefore, we also performed synthetic analysis by changing the timestamps of both data sets. Fig. 12 shows when increasing the ratios of fixed time intervals in both Singapore and Beijing data sets, the compression ratios increased. When there is no fixed time intervals, only the

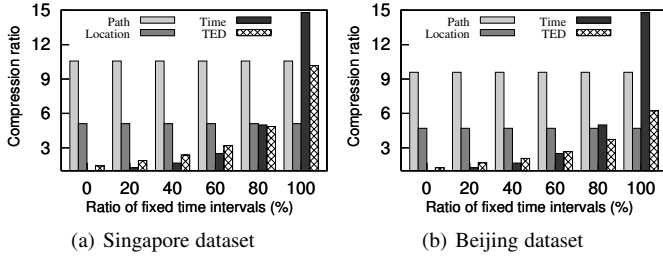


Fig. 12: Effect of fixed time intervals on TED Compression.

temporal compression ratio was zero, the compression ratios for entry path and location were not affected.

(iii) **Effect of encoding schemes.** Fig. 11 and Table 6 report the compression ratios under four different encoding schemes, including Huffman encoding, adding fixed bits, DDP encoding, and PDDP encoding for distances D when $\eta = 1m$. We used Huffman encoding as a ground truth since it theoretically guarantees the smallest space consumption and can achieve the highest compression ratio. The compression ratio of PDDP encoding was very close to that of Huffman encoding. For Singapore data set, compression ratios for D under Huffman, AddFixedBits, DDP, and PDDP were 5.68, 4.19, 5.20, and 5.44, respectively. The results for Beijing data set were similar.

(iv) **Efficiency of Compression and Decompression.** Table 7 shows the results. Both TED compression and depression ran faster than PRESS compression and Ext.SDM. This is because our representation maintains the mapping relationship among T, E, and D, and the whole process is based on bit operations.

5.2 Performance of Answering LBS Queries on Compressed Trajectories

We compared processing of LBS queries under TED compression with that of PRESS. We only report partial query results on Singapore data set as results on Beijing data set were similar.

Fig. 13 shows the performance of using TED-based query processing. We can see that TED can answer different types of queries efficiently. Figs. 13(a) and 13(b) show the comparison of using TED and PRESS for *when* and *where* queries only since PRESS does not support the other five queries. TED-based query processing was much faster than PRESS by over two orders of magnitude for *when*. When increasing the number of trajectories, both TED-based approach and PRESS required longer time for query processing. PRESS also adopts R-tree to process queries. Since PRESS needs to load the Shortest-Path table, which was huge for Beijing dataset, TED-based query processing ran faster than PRESS by three orders of magnitude.

5.3 Index Size

Table 8 shows index sizes for 15,000 Singapore taxi trajectories and 1,368 Beijing taxi trajectories. TED-based index was only 15.292MB on Singapore dataset, and the major space cost for TED-based index is the partition tree (the depth of partition tree was 8 for this test). The PRESS-based index was 2.842GB. PRESS uses three types of indices to support their query processing, which are Aho-Corasick automaton, Huffman tree, and SP-table (Shortest path table). We can see any of them was larger than the whole TED index size. The majority space cost for PRESS index is SP-table, which was 194.1146GB for Beijing

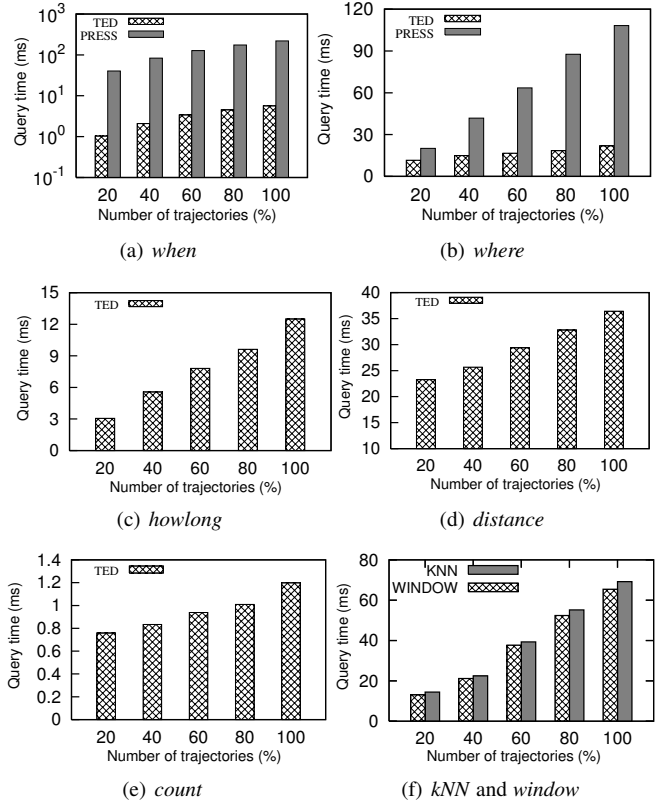


Fig. 13: Performance of partial LBS queries.

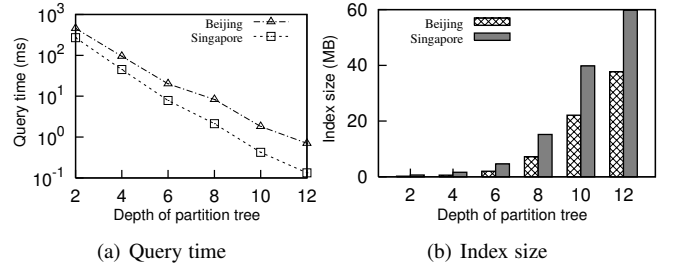


Fig. 14: Depth of partition tree.

road network. The size of SP-table depends on the size of road network but trajectories. Notice that, TED based approach does not need shortest path to support query processing.

We also evaluated the effect of partition granularity on query performance. Fig. 14 shows the query time by adopting different depths of partition trees. As the depth of partition tree increased, the query time decreases exponentially since the road network can be partitioned into smaller regions. It is not surprising to see that with the depth of partition tree increases, the size of index would get larger accordingly.

6 RELATED WORK

Trajectory compression can be roughly classified into the following categories.

Simplification-based compression. There are a bunch of compression techniques for reducing the number of raw time-stamped points. Such approaches [6], [10], [15], [21] are also called path and line simplification, that approximate a polyline with a subset of the vertices from the raw time-stamped points using a simplification error to bound the accuracy.

TABLE 4: Comparison of total compression ratios.

Data sets	TED		PRESS		Ext.SDM Error bound = 4m	WinZip	SPNET	SharkDB
	Error bound		TSND=30m, NSTD=30 Sec.	TSND=100m, NSTD=100 Sec.				
	$\eta = 1m$	$\eta = 4m$						
Singapore	10.1819	10.62143	3.18491	3.18897	4.545	4.21232	2.58518	4.46
Beijing	6.23195	7.15463	2.64732	2.65045	2.2727	3.69836	2.35793	4.46

TABLE 5: Compression ratio using TED compression.

Data sets	Entry path E (lossless)		Distance D ($\eta = 1m$)	Time T (lossless)
	One base	Multi bases		
Singapore	10.5684	12.8543	5.10769	37.626
Beijing	10.5886	14.2471	4.69909	16.9946

TABLE 6: Compression ratios of distance compression.

Data sets	Huffman	AddFixedBits	DDP	PDDP
Singapore	5.67953	4.18699	5.20357	5.43879
Beijing	4.07673	3.09538	3.86785	4.04217

Douglas-Peucker Algorithm [6] constructs a compressed trajectory T' by repeatedly removing points from T until the maximum spatial error becomes smaller than τ . Based on Douglas-Peucker Algorithm, Top-Down Time Ratio (TD-TR) [15] also considers the temporal error. Opening Window algorithms [10] slides a window over the points in the original trajectory to approximate each trajectory, so that the resulting spatial error is smaller than a bound τ . Dead Reckoning [21] stores the location and velocity of the point location to compress the trajectory within a threshold τ . In a nutshell, these algorithms and their variants use a subset of the trajectory points to compress the trajectory within a given error bound. Recently, DPTS [14] introduces the notion of direction preserving trajectory simplification to support a broader range of applications than traditional position-preserving trajectory simplification. SharkDB [23] proposes a frame-based structure to store and compress trajectories. It allocates one sample point in each frame by adding/removing sample points and stores each frame into column-oriented data structure. Furthermore, it uses delta encoding to compress trajectory data. Both DPTS and SharkDB use a simplification error to bound the accuracy.

Geographical-embedded compression. Studies [9], [19], [20], [24], [25] on geographical embedded compression increase significantly based on the observation that most trajectories are associated with geographical networks. Techniques mainly focus on two directions: one is to link GPS raw data to a network requiring matching a time-stamped point with a map (i.e. map matching), and the other is to compress trajectories in geographical network once the raw data is mapped to the network.

Both indoor [25] and outdoor matching algorithms have been proposed to link GPS raw data to a network in recent years. In a road network scenario, point-to-point and point-to-curve algorithms are presented to match a time-stamped point against a vertex and an edge in the road network, respectively. Since a high accuracy can be achieved in road network [24], recent techniques [9], [19], [20] are developed to compress trajectories that are embedded in a road network.

The closest work to ours is PRESS [20], which separates the spatial representation of a trajectory from the temporal representation. It adopts Huffman encoding to compress spatial path based on the assumption that moving objects tend to take shortest paths. The representation of temporal information is storage consuming, therefore, it provides an error-bounded temporal compression.

Semantic-based Compression. STC [19] presents a semantic trajectory compression for compressing trajectories in transporta-

tion networks, where only semantically crucial points are kept to achieve the minimal storage requirement. The algorithm is bounded by a given threshold and is mainly developed for relatively high sampling rate, low speed, and short trajectories generated by human. While we focus on the massive long-lasting vehicle trajectories. Moreover, it only uses 18 real trajectories with less than 200 sample points to evaluate the compression ratio. The data size is too small to demonstrate the scalability of compression algorithm. It is bounded by a given threshold.

Dictionary-based Compression. [13] treats a time-series as the string, “segments” are analogous to “words,” and timestamped points are similar to “characters.” It adopts Lempel-Ziv compression to compress time-series. [18] proposes an adaptive trajectory (lossy) compression algorithm based on learnt dictionary matrix.

In addition, [4] proposes an adaptive storage system for large trajectory data sets even it is not based on a road network. Our TED representation and compression on trajectories embedded in a road network can achieve a high compression ratio. The maximum difference between a real relative distance and its encoded code could be derived, so our distance compression is bounded, which is much smaller and can be fixed; while all the other compression algorithms, to the best of our knowledge, take this bound as an input (with much larger values).

7 CONCLUSIONS

We proposed our TED-representation for trajectories embedded in road networks. This representation has the inherent advantage that contains large number of duplicate information, which makes it achieve a lower entropy compared with existing representations, thereby drastically cutting the storage cost. Based on the presentation, we devised compression approaches to achieve high compression ratios. In addition we showed that LBS queries can be directly performed on compressed trajectories. Experimental study on real datasets shows the effectiveness and efficiency of the TED compression. We obtained around 10 compression ratio with an error bound of 1m only, i.e. the compressed trajectory is around 10% of the original trajectories. We also test the efficiency of direct queries on compressed trajectories. Note that trajectories are usually added in an append manner (i.e. appending new time-stamped data in a trajectory or adding a new trajectory) and our TED-representation can easily support such incremental compression since all compressed codes are organized in bit strings, which are stored in bytes. We can use our current technique on the appended bytes. Due to space reason, we do not discuss the detail. In addition, we would like to consider another kind of incremental compression for new trajectories in future work.

REFERENCES

- [1] S. E. Anderson. Bit twiddling hacks, May 2005.
- [2] A. Colantonio and R. D. Pietro. Concise: Compressed ‘n’ composable integer set. *Inf. Process. Lett.*, 110(16):644–650, 2010.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Reading, Massachusetts, 2nd edition, 2001. (book).

TABLE 7: Compression and Decompression time (Min.).

Data sets	TED (bound: $\eta = 1m$)		PRESS (bound: 30m, 30 Sec.)		Ext.SDM (bound: 4m)		WinZip	
	Compression	Decompression	Compression	Decompression	Compression	Decompression	Compression	Decompression
Singapore	367.40	244.61	700.47	512.23	571.23	257.31	72.33	30
Beijing	22.95	15.28	462.61	330.24	36.18	15.33	3.95	2.04

TABLE 8: Index size to support queries.

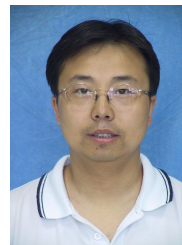
Singapore data set	TED (15.292MB)		
	EntryTable	EncodingTree	PartitionTree
	112KB	689B	15.18MB
	PRESS (2.842GB)		
	Aho-Corasick automaton	Huffman	SP-table
17.2MB	25.1MB	2.8GB	
Beijing data set	TED (8.066MB)		
	EntryTable	EncodingTree	PartitionTree
	866KB	461B	7.2MB
	PRESS (194.1146GB)		
	Aho-Corasick automaton	Huffman	SP-table
22.7MB	36.9MB	194.055GB	

- [4] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, 2010.
- [5] F. Delière and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *EDBT*, pages 228–239, 2010.
- [6] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The Intl. J. for Geographic Info. and Geovisualization*, 10(20):112–122, 1973.
- [7] F. Fusco, M. P. Stoecklin, and M. Vlachos. Net-flt: On-the-fly compression, archiving and indexing of streaming network traffic. *PVLDB*, 3(2):1382–1393, 2010.
- [8] D. Hankerson, G. A. Harris, and P. D. Johnson. *Introduction to Information Theory and Data Compression*. Chapman and Hall/CRC, Reading, Massachusetts, 2nd edition, 2003. (book).
- [9] G. Kellaris, N. Pelekis, and Y. Theodoridis. Map-matched trajectory compression. *Journal of Systems and Software*, 86(6):1566–1579, 2013.
- [10] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In *ICDM*, pages 289–296, 2001.
- [11] B. B. Krogh, C. S. Jensen, and K. Torp. Efficient in-memory indexing of network-constrained trajectories. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 17:1–17:10, 2016.
- [12] J. Krumm. Trajectory analysis for driving. In *Computing with Spatial Trajectories*, pages 213–241. 2011.
- [13] W. Lang, M. D. Morse, and J. M. Patel. Dictionary-based compression for long time-series similarity. *IEEE Trans. Knowl. Data Eng.*, 22(11):1609–1622, 2010.
- [14] C. Long, R. C. Wong, and H. V. Jagadish. Direction preserving trajectory simplification. *PVLDB*, 6(10):949–960, 2013.
- [15] N. Meratnia and R. A. de By. Spatiotemporal compression techniques for moving point objects. In *EDBT*, pages 765–782, 2004.
- [16] J. Niedermayer, A. Zfle, T. Emrich, M. Renz, N. Mamoulis, L. Chen, and H.-P. Kriegel. Probabilistic nearest neighbor queries on uncertain moving object trajectories. *PVLDB*, 7(3):205–216, 2013.
- [17] I. S. Popa, K. Zeitouni, V. Oria, and A. Kharrat. Spatio-temporal compression of trajectories in road networks. *Geoinformatica*, 19(1), 2015.
- [18] R. K. Rana, M. Yang, T. Wark, C. T. Chou, and W. Hu. Simpletrack: Adaptive trajectory compression with deterministic projection matrix for mobile sensor networks. *CoRR*, abs/1404.6151, 2014.
- [19] K.-F. Richter, F. Schmid, and P. Laube. Semantic trajectory compression: Representing urban movement in a nutshell. *Journal of Spatial Information Science*, pages 3–30, 2012.
- [20] R. Song, W. Sun, B. Zheng, and Y. Zheng. PRESS: A novel framework of trajectory compression in road networks. *PVLDB*, 7(9), 2014.
- [21] G. Trajcevski, H. Cao, P. Scheuermann, O. Wolfson, and D. Vaccaro. On-line data reduction and the quality of history in moving objects databases. In *MobiDE*, pages 19–26, 2006.
- [22] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, 2011.
- [23] H. Wang, K. Zheng, X. Zhou, and S. W. Sadiq. Sharkdb: An in-memory storage system for massive trajectory data. In *SIGMOD*, 2015.

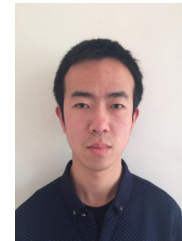
- [24] C. E. White, D. Bernstein, and A. L. Kornhauser. Some map matching algorithms for personal navigation assistants. *Transportation Research Part C Emerging Technologies*, 1(6):91–108, 2000.
- [25] X. Xie, H. Lu, and T. B. Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *ICDE*, pages 434–445, 2013.
- [26] X. Xie, M. L. Yiu, R. Cheng, and H. Lu. Scalable evaluation of trajectory queries over imprecise location data. *TKDE*, 26(8):2029–2044, 2014.
- [27] B. Yang, C. Guo, Y. Ma, and C. S. Jensen. Toward personalized, context-aware routing. *VLDB J.*, 24(2):297–318, 2015.
- [28] J. Yuan, Y. Zheng, X. Xie, and G. Sun. T-drive: Enhancing driving directions with taxi drivers’ intelligence. *TKDE*, 2012.



Xiaochun Yang is a professor in the School of Computer Science and Engineering, Northeastern University, China. She received her PhD degree in computer science from Northeastern University, China, in 2001. Her research interests include data management, string processing, data quality, and data privacy. She is a member of the ACM, the IEEE, and a senior member of the CCF.



Bin Wang is an associate professor in the School of Computer Science and Engineering, Northeastern University, China. He received his PhD degree in computer science from Northeastern University, China, in 2008. His research interests include design and analysis of algorithms, queries processing over streaming data, and distributed systems.



Kai Yang is a master student in the School of Computer Science and Engineering, Northeastern University, China. He received his Bachelor degree in computer science from Northeastern University, China, in 2015. His research interests include trajectory processing and similarity search.



Chengfei Liu received his PhD degrees in computer science from Nanjing University, China, in 1988. Currently, he is a professor in the Swinburne University of Technology, Australia. His research interests include keywords search on structured data, query processing, and refinement for advanced database applications, query processing on uncertain data and big data, and data-centric workflows. He is a member of the IEEE and the ACM.



Baihua Zheng is an associate professor in the School of Information Systems, Singapore Management University. She received her PhD degree in computer science from Hong Kong University of Science and Technology. Her research interests include data management and mobile/pervasive computing. She has published more than 100 technical papers in these areas. She is a member of the IEEE.

APPENDIX A

PROOF OF THEOREMS

A.1 Proof of Theorem 1

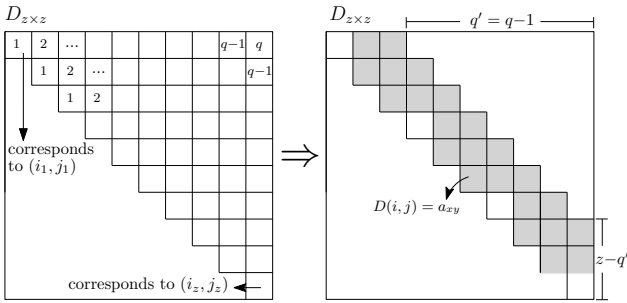
Proof. We prove Theorem 1 by giving a reduction from the Traveling Salesman Problem to our matrix transformation problem in polynomial time. Given a directed graph $G(V, E)$ with m nodes and $m(m-1)$ edges, we construct a matrix $M_{n \times m}$. Each column vector \vec{v}_i in $M_{n \times m}$ corresponds to a vertex v_i in the graph. Let all the outcome edges from each node v_i have the same weights. For each edge e from vertex v_i to vertex v_j in the graph, we construct the i -th column vector \vec{v}_i that contains the same number of 0s as the weight value of the edge. A path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_m$ corresponds to a matrix $M = (\vec{v}_1 \vec{v}_2 \dots \vec{v}_m)$.

Then there is a minimal travel path with k edges that cover all vertices in G if and only if there is an alignment with k columns. We call such matrix M'' . Thus the problem of transforming M to M'' is NP-hard. In M'' there could be more than one duplicate columns in the alignment, we keep the most left column and remove the others (e.g. we change $\dots \vec{v}_i \vec{v}_j \vec{v}_i \dots$ to $\dots \vec{v}_i \vec{v}_j \dots$). We can do it in $O(m)$ time. For each row in the remaining matrix, we can find a boundary such that cells in the right part to the boundary contain only 0s and cells in the left part to the boundary contain either 0 or 1. Let value c be the number of cells in the right part. We then rank each row in M'' in ascending of c values of each row in $O(n)$ time to generate the final matrix M' .

In summary, the main time cost of our matrix transformation problem is the problem of choosing an alignment to generate M'' , which is equivalent to the problem of choosing a shortest path to cover all the vertices in the original graph. Thus our problem is NP-hard. \square

A.2 Proof of Theorem 2

Proof. For ease of representation, we use a matrix $D_{z \times z}$ to represent these z boundary points, and each cell $D(i, j)$ represents the corresponding area a_{xy} in M' (see Figure 15). Then if we choose $q-1$ boundary points to partition entry paths into q groups, the summation of their gains will repeatedly count the area $D(1, z)$ for q times and the areas $D(1, q-1)$ and $D(2, q)$ for $q-1$ times. So we do not need to consider these areas. Then gain for each boundary point is changed to the one shown in Fig. 15(b).



(a) $Gain(i_x, j_x) = \sum_{i=1}^x \sum_{j=x}^z D(i, j)$ (b) $Gain(i_x, j_x) = \sum_{i=1}^x \sum_{j=x}^{z-q+1} D(i, j)$

Fig. 15: Maximizing partition.

Then we prove Theorem 2 by giving a reduction from the Knapsack Problem to our partition problem in polynomial time. Given a set of z items, we construct a matrix $D_{z \times z}$. We first rank items in ascendant order of their weights as w_1, \dots, w_z . We then assign values of different cells iteratively as follows. For the first

item with weight w_1 , let cells from $D(1, 2)$ to $D(1, z - q')$ be either 0 or 1 such that $\sum_{j=1}^{z-q'} D(1, j) < w_1$ and let $D(1, 1) = w_1 - \sum_{j=1}^{z-q'} D(1, j)$. For the second item with weight w_2 , let cells from $D(2, 3)$ to $D(2, z - q' + 1)$ be either 0 or 1 such that $\sum_{j=1}^{z-q'} D(1, j) + \sum_{j=2}^{z-q'+1} D(1, j) < w_2$ and let $D(2, 2) = w_2 - \sum_{j=1}^{z-q'} D(1, j) - \sum_{j=2}^{z-q'+1} D(1, j)$.

Similarly, for the i -th item ($2 \leq i \leq z$) with weight w_i , let cells from $D(i, i+1)$ to $D(i, z - q' + i - 1)$ be either 0 or 1 such that

$$\sum_{l=\max(1, i-(z-q'-1))}^i \sum_{j=l}^{\min(z, z-q'+l-1)} D(i, j) < w_i$$

and let

$$D(i, i) = w_i - \sum_{l=\max(1, i-(z-q'-1))}^i \sum_{j=l}^{\min(z, z-q'+l-1)} D(i, j).$$

Then there are q items with value k if and only if there are q chosen boundary points in M' with gain k . Thus our problem is NP-hard. \square

A.3 Proof of Theorem 3

Proof. Let $q' = q-1$ be the number of chosen boundary points in M' that can partition D into q groups.

(i) When PARTITION chooses only one boundary point (i.e. $q' = 1$), $\frac{OPT}{OPT^*} = 1$.

(ii) When $q' = z-1$, $\frac{OPT}{OPT^*} = 1$.

(iii) When $1 < q' < z-1$, the OPT solution will try to make the covered area as large as possible by choosing scattered boundary points, and the OPT solution could cover at most $z \cdot (z - q') - \frac{z}{q'+1}$ areas in M' (see Fig. 15). Whereas PARTITION can cover at least $\frac{q'}{q'+1} \cdot z \cdot (z - q')$ areas by choosing consecutive q' boundary points. Then

$$\frac{OPT}{OPT^*} \leq \frac{z \cdot (z - q') - \frac{z}{q'+1}}{\frac{q'}{q'+1} \cdot z \cdot (z - q')} < \frac{q' + 1}{q'}.$$

Since $2 \leq q' < z-1$, and $\frac{q'+1}{q'}$ is monotonically decreasing, we get $\frac{OPT}{OPT^*} < \frac{3}{2}$. \square

A.4 Proof of Theorem 4

Proof. Let $W = \{w_1, \dots, w_g\}$ be a set of binary codes that satisfy the prefix condition and Tr be its corresponding DP-tree. Let w be a prefix of code w_i , so $W \cup \{w\}$ does not satisfy the prefix condition. Now we expand w to w' via algorithm EXPANSION.

Assume there is another code $w'' \notin W$ ($w'' \neq w'$, $|w''| < |w'|$), such that w'' and w represent the same relative distance and $W \cup \{w''\}$ satisfies the prefix condition. Then, we add a path in Tr to represent w'' , and let $d'' = |w''| - |w|$. Since w is a prefix of w_i , and w and w'' represent the same relative distance, w'' and w_i must share a common ancestor node n_a such that the path from root to n_a represents code w and the path from n_a to the leaf node n'' corresponding to w'' must consist of d'' edges labeled as 0s.

Let $d' = |w'| - |w|$ and n' be the node corresponding to w' . According to the algorithm EXPANSION, we know there must exist a code $w_j \in W$ such that w' and w_j share a common ancestor node n'_a and n'_a is the parent node of n' . According to

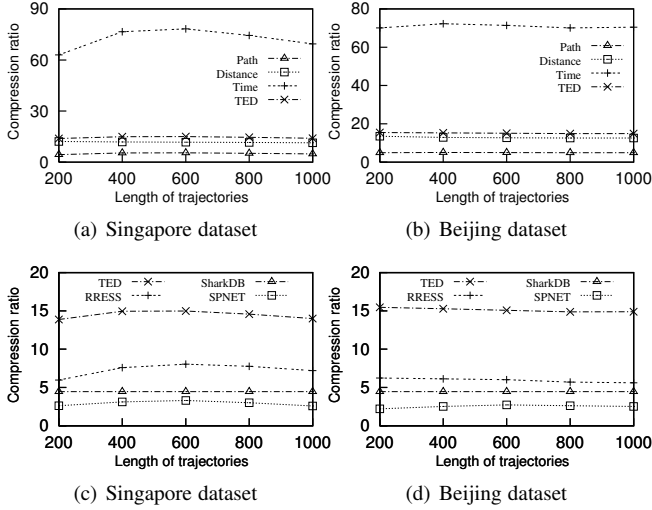


Fig. 16: Effect of trajectory length.

the assumption that $|w''| < |w'|$, the path representing w' must belong to the path corresponding to w_j , then $W \cup \{w''\}$ does not satisfy prefix condition, which contradicts the assumption. Therefore, the assumption is not valid, and the proof completes. \square

APPENDIX B MORE EXPERIMENTAL RESULTS

B.1 Effect of Trajectory Length on Compression Ratios

We also test the compression ratio under different lengths of trajectories, with the results reported in Fig. 16. The experiments show that with the increase of length of trajectories, the compression ratio does not change obviously. The compression ratios for entry path and distance did not change when increasing the trajectory length, while the ratio for time varied a little bit since some of time slots were not reported regularly. The comparisons of our TED compression with other approaches are shown in Figs. 16(c) and 16(d). Again, TED achieves the best performance in terms of compression ratio.

B.2 Effect of Error Bounds

Then we investigate the accuracy of compression with regard to the error bounds. We set the error bounds to 1m, 3m, . . . , 9m as mentioned in Section 5.1, and report the accuracy performance for three different types of queries (i.e., count queries, window queries, and k NN queries) in Fig. 17. As reported in Figs. 17(a) and 17(b), the error rates corresponding to different types of queries are always below 0.3% for Singapore dataset and below 0.2% for Beijing dataset. In Figs. 17(c) and 17(d), we evaluate the average difference of time for *howlong* and *when* queries, and results show that the average difference is less than 1.5 seconds for Singapore and 2 seconds for Beijing datasets. Then we test the average difference of distance for *distance* and *where* queries. As we can see, the difference is less than 6 meters and 8 meters for Singapore and Beijing datasets. Obviously, the higher the error bound is, the lower the accuracy is. This is because increasing the error bounds will result in a more significant loss of the information of datasets, which will decrease the accuracy when decompressing datasets for queries.

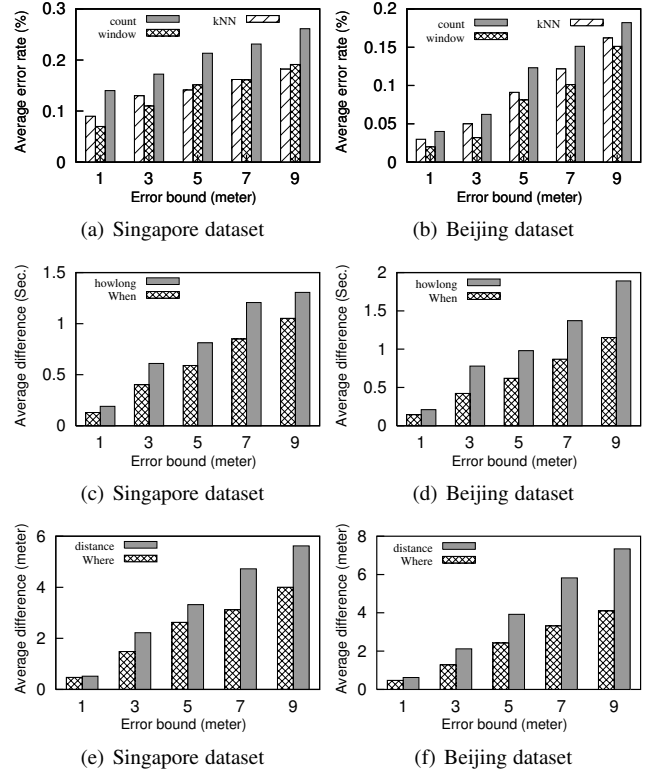


Fig. 17: Effect of error bounds.

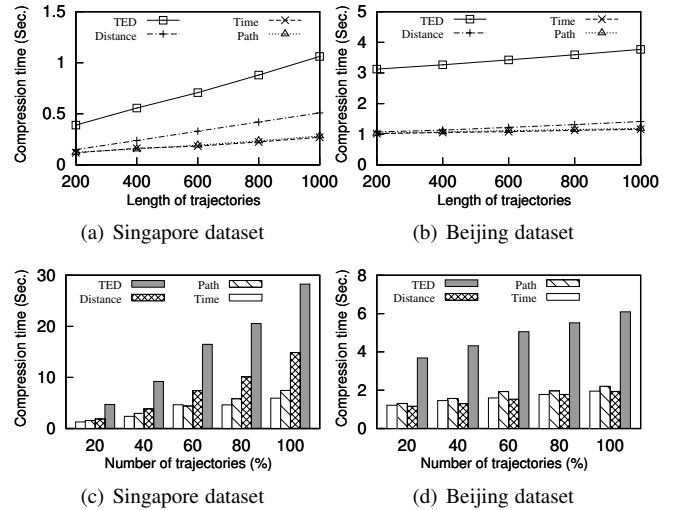


Fig. 18: Compression time.

B.3 Compression Performance

Last but not the least, we evaluate the compression time cost and space cost as follows.

We evaluated the compression time with the increase of length and number of trajectories in Fig. 18. As we can see from Figs. 18(a) and 18(b), the compression time is almost linear with the length of trajectories (varied from 200m to 1000m). Figs. 18(c) and 18(d) show that compression time is nearly linear with the number of trajectories (from 20% to 100%).

The space cost is reported in Fig. 19. We record the memory footprint for T, E and D respectively when compressing. As expected, increasing the length or number of trajectories will

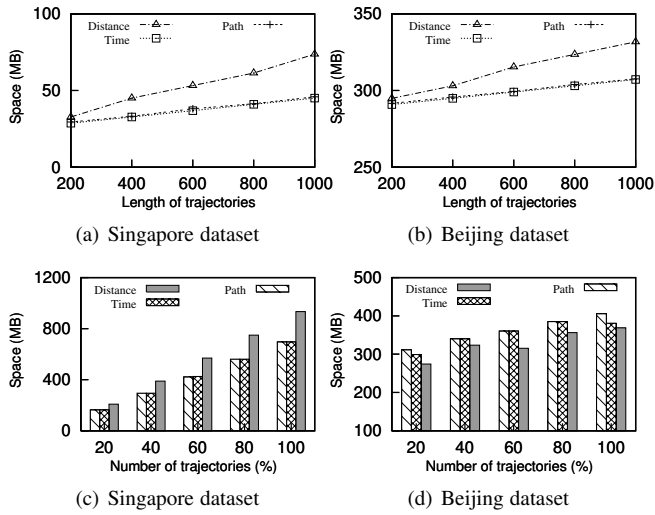


Fig. 19: Space cost.

increase the space cost in compression stage. Similar to the time cost, the space cost is almost linear with the length and nearly linear with the number of trajectories. An interesting observation is that there is not much difference between the space costs when compressing time and path.