# COMPRESS: A Comprehensive Framework of Trajectory Compression in Road Networks

YUNHENG HAN and WEIWEI SUN, Fudan University & Shanghai Key Laboratory of Data Science
BAIHUA ZHENG, Singapore Management University

More and more advanced technologies have become available to collect and integrate an unprecedented amount of data from multiple sources, including GPS trajectories about the traces of moving objects. Given the fact that GPS trajectories are vast in size while the information carried by the trajectories could be redundant, we focus on trajectory compression in this article. As a systematic solution, we propose a comprehensive framework, namely, COMPRESS (*Comprehensive Paralleled Road-Network-Based Trajectory Compression*), to compress GPS trajectory data in an urban road network. In the preprocessing step, COMPRESS decomposes trajectories into spatial paths and temporal sequences, with a thorough justification for trajectory decomposition. In the compression step, COMPRESS performs spatial compression on spatial paths, and temporal compression on temporal sequences in parallel. It introduces two alternative algorithms with different strengths for lossless spatial compression and designs lossy but error-bounded algorithms for temporal compression. It also presents query processing algorithms to support error-bounded location-based queries on compressed trajectories without full decompression. All algorithms under COMPRESS are efficient and have the time complexity of $O(|T|)$, where $|T|$ is the size of the input trajectory $T$. We have also conducted a comprehensive experimental study to demonstrate the effectiveness of COMPRESS, whose compression ratio is significantly better than related approaches.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: GPS trajectory, road network, trajectory compression, map matching, information entropy, trajectory representation, entropy encoding, dictionary coder, stabbing polyline

**ACM Reference Format:**
Yunheng Han, Weiwei Sun, and Baihua Zheng. 2017. COMPRESS: A comprehensive framework of trajectory compression in road networks. ACM Trans. Database Syst. 42, 2, Article 11 (May 2017), 49 pages.
DOI: http://dx.doi.org/10.1145/3015457



## 1. INTRODUCTION

In the era of big data, quantities of data reach almost incomprehensible proportions. As we move forward, we are going to have more and more huge data collections. Take the

This research is supported partially by the National Natural Science Foundation of China (61073001), Natural Science Foundation of Shanghai (14ZR1403100), and National Research Foundation, Prime Ministers Office, Singapore, under its International Research Centres in Singapore Funding Initiative. Y. Han also received additional support from the National University Student Innovation Program (201410246020) and Fudan's Undergraduate Research Opportunities Program (14021).
Authors' addresses: Y. Han and W. Sun (corresponding author), School of Computer Science, Fudan University, 825 Zhangheng Road, Shanghai, China, 201203; emails: {yhhan16, wwsun}@fudan.edu.cn; B. Zheng, School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore, 178902; email: bhzheng@smu.edu.sg.

DOI: http://dx.doi.org/10.1145/3015457

GPS trajectories generated by moving objects in urban spaces as an example. According to the statistics released by the U.S. Department of Transportation, motor vehicles in the United States travelled $2.95 \times 10^{12}$ miles in total in 2011 [OHPI 2014]. If we assume the speed limit is 60 miles per hour and a low GPS sampling rate of 1/60 Hz, motor vehicles in the United States generated at least 53TB of GPS trajectory data in 2011. Consequently, it is critical to reduce the size of trajectory data in order to alleviate the storage overhead and communication cost. On the other hand, many trajectories could be similar as their movements are strictly bounded by the given road network and hence the information carried by trajectories could be redundant. For example, the total length of the road network in the United States is about $4.08 \times 10^6$ miles, as reported by the Central Intelligence Agency (CIA) in 2012, while motor vehicles in the United States traveled in total $2.95 \times 10^{12}$ miles in 2011. This means on average each road is traveled over 723,000 times within a year, which further justifies the necessity of trajectory data compression.

In this work, we focus on the trajectory compression issue. To be more specific, we model a road network as a directed graph $G = (V, E)$, where $V$ is the vertex set and $E$ is the edge set. The weight on an edge $e$, denoted as $w(e)$, can be physical distance, travel time, or other costs according to different application contexts. A trajectory is the path that a moving object follows through space as a function of time. Consequently, it contains both spatial information and temporal information. Traditional approaches represent a trajectory $T$ via a sequence of $n$ triples in the form of $(\langle x_1, y_1, t_1\rangle, \langle x_2, y_2, t_2\rangle, \ldots, \langle x_n, y_n, t_n\rangle)$, where $(x_i, y_i)$ records the longitude and latitude of a moving object at timestamp $t_i$.[1]

As a solution, we propose a novel and efficient framework, namely, COMPRESS (*Comprehensive Paralleled Road-Network-Based Trajectory Compression*), as a systematic solution to compress the trajectory data. The main objective of COMPRESS is to achieve a high compression ratio. In addition, we also consider the requirements of different applications and want to retain the utility of trajectory data even in the compressed form. COMPRESS proposes a suite of compression algorithms with different strengths to cater for the requirements of various applications. For example, the *Hybrid Dictionary Compression* algorithm can achieve a good compression ratio with reasonably high utility, the *Labeling and Coding* compression algorithm can guarantee a much higher compression ratio with low utility, and the *Bounded Temporal Compression* algorithm can adjust its compression ratio via setting tolerant errors to different values with certain data utility.

In the rest of the article, we first present the architecture of the COMPRESS framework in Section 2 and also list the main contributions of the COMPRESS framework as compared with existing works. We next present the trajectory decomposition, spatial compression, and temporal compression, the three key tasks performed by COMPRESS in Section 3, Section 4, and Section 5, respectively. We then list some applications that COMPRESS can support in Section 6. We report our experimental study in Section 7 and review existing work in Section 8. Finally, we conclude this article with some directions for future work in Section 9.

## 2. OVERVIEW OF COMPRESS

Now we present the main components and basic operating principles of the COMPRESS framework. As shown in Figure 1, the COMPRESS framework takes GPS

[1]Note that $x_i$ and $y_i$ are spherical coordinates instead of Cartesian coordinates, and the distance between sample points is calculated by the spherical law of cosines. In our experiments, both $x_i$ and $y_i$ occupy 8 bytes, in the format of double-decision decimal points, and $t_i$ occupies 4 bytes, in the format of a single-decision decimal point.

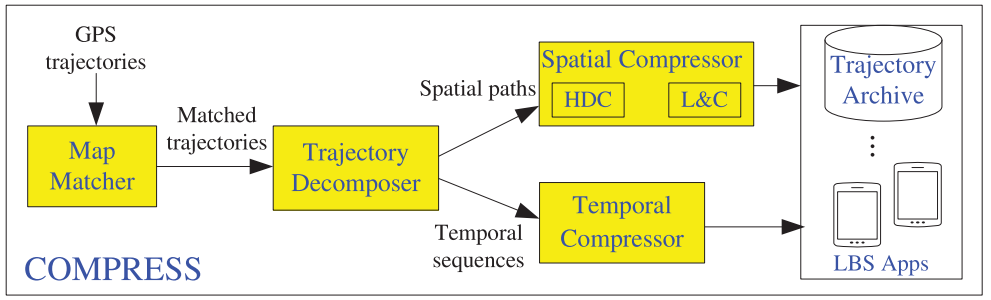



Fig. 1. COMPRESS framework.

trajectories as input and invokes *Map Matcher* to map the raw trajectories to a given road network [Brakatsoulas et al. 2005; Lou et al. 2009; Newson and Krumm 2009; Song et al. 2012]. Map Matcher then clusters the GPS trajectories into two sets, one set of trajectories, denoted as $\Gamma m$, that can be matched successfully to the given road network and the other set of trajectories, denoted as $\Gamma u$, that fails to be matched. Our COMPRESS framework only manages matched trajectories preserved in $\Gamma m$, while trajectories in $\Gamma u$ have their own application base such as auto-recovering the road segments that are missing in the road network [Wu et al. 2015; Shan et al. 2015]. We also want to highlight that how to match the trajectory to maps is not the focus of this work, and we adopt an existing map-matching algorithm in our implementation. Note that map-matching algorithms will be reviewed in Section 8.

Next, the COMPRESS framework invokes the *Trajectory Decomposer* to decompose each trajectory into a *spatial path* and a *temporal sequence*. In view of the fact that trajectory data is a series of space locations as a function of time, some data reduction algorithms apply space-time decomposition to trajectories, including not only GPS data [Song et al. 2014; Cao and Wolfson 2005] but also data generated by other devices, for example, data from particle accelerators [Arpaia et al. 2010]. Although we are not the first to decompose the GPS trajectories, we want to highlight that this is the *first* work to explain the inefficiency of representing a trajectory in the format of the $\langle x_i, y_i, t_i \rangle$ sequence theoretically from the perspective of trajectory compression, which provides a thorough justification for trajectory decomposition.

After decomposing the trajectory data into spatial paths and temporal sequences, the COMPRESS framework invokes two separate compressors, that is, the *spatial compressor* and *temporal compressor*, to compress them, respectively. Note that the spatial compressor implements two core algorithms, namely, *Hybrid Dictionary Compression (HDC)* and *Labeling and Coding (L&C)*. Both algorithms can perform *lossless* compression on spatial paths with different strengths. HDC can preserve certain key spatial properties of the original spatial paths even after compression. In other words, spatial paths compressed by HDC can still support searches, and hence it is suitable for applications that need to not only compress trajectory data but also perform queries, for example, location-based service applications. On the other hand, L&C can achieve a better compression rate but the compressed data are not able to support any search unless the compressed data are decompressed. Consequently, L&C is ideal for data archiving. The temporal compressor implements *lossy* compression algorithms based on Computational Geometry, including *Rigid Stabbing polyLine Compression (RSLC)* and *Tube Stabbing polyLine Compression (TSLC)*. Both lossy algorithms ignore *unnecessary* points instead of encoding them, so searches on compressed temporal sequences are exactly the same as those on the original ones.

As a summary, our COMPRESS framework is still based on the trajectory decomposition, which was initially proposed in Song et al. [2014] as a new approach to represent a trajectory. The COMPRESS framework extends the initial study via not only proposing new spatial compression algorithms and new temporal compression algorithms with better performance but also justifying the advantages of our trajectory decomposition framework via thorough theoretical analysis. To be more specific, the major value-added extension over Song et al. [2014] in the COMPRESS framework is summarized as follows:

—Although Song et al. [2014] have already implemented trajectory decomposition, their paper lacks the theoretical support from information theory. In this article, we analyze the inefficiency of traditional GPS trajectory representation in terms of encoding in Section 3.1. It justifies the necessity of representation transformation before compression and provides a theoretical basis for all the algorithms developed.
—We propose two new algorithms to compress the spatial paths in Section 4, that is, HDC based on dictionary coding and L&C based on entropy coding. Both dictionary coding and entropy coding are optimal algorithms that achieve the entropy limit, while the original algorithm proposed in Song et al. [2014] is heuristic but not optimal. HDC constructs a dictionary for not only compression but also serving as an index for efficient query processing. L&C implements an optimal labeling algorithm to achieve the highest compression ratio.
—Two error metrics, namely, TSND and NSTD, were introduced in Song et al. [2014] to bound errors in time and distance dimensions, respectively. However, the time compression algorithm proposed in Song et al. [2014] is a greedy algorithm but not optimal. Accordingly, we introduce a geometric model for the problem of temporal compression and design a new temporal compression algorithm in Section 5.3, namely, TSLC, which is optimal in terms of compression ratio.

Besides the three extensions just mentioned, the error-bounded query processing on partially decompressed data discussed in Song et al. [2014] is still available after COMPRESS implements new compression algorithms. Query processing on partially decompressed data is efficient in terms of both storage cost and time cost, as compared with conventional methods. Finally, we conduct extensive experiments to compare the performance of COMPRESS and other existing methods and to demonstrate the efficiency and effectiveness of the COMPRESS framework.

## 3. TRAJECTORY REPRESENTATION

A trajectory is the path that a moving object follows through space as a function of time, and most, if not all, existing works present a trajectory via a sequence of $n$ triples in the form of $\langle x_1, y_1, t_1\rangle, \langle x_2, y_2, t_2\rangle, \ldots, \langle x_n, y_n, t_n\rangle$. In the following, we first analyze the limitation of this conventional representation in terms of compression ratio and then present the proposed decomposition approach.

### 3.1. Analysis on Conventional Representation

*Compression ratio* is a common metric to evaluate the effectiveness of compression algorithms, which is defined in Definition 3.1. Here, compression ratio is the ratio of the size of original data to the size of the compressed data. For example, if we compress a trajectory of 4KB data into a compressed form of 2KB, the compression ratio is $\frac{4}{2} = 2$.

*Definition* 3.1 (*Compression Ratio*). Given a trajectory $T$ of size $|T|$ and a compressed trajectory $T^c$ of $T$ with size $|T^c|$, the compression ratio is $\frac{|T|}{|T^c|}$.

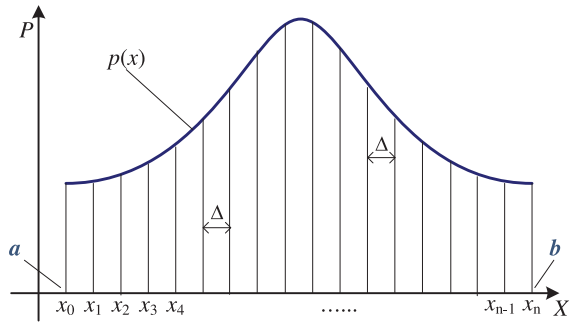



Fig. 2. Division of the interval.

Data compression can be either lossless or lossy. Let's first consider *lossless* compression on the original trajectory. If we use conventional algorithms such as Huffman coding [Huffman 1952] or Lempel-Ziv coding [Lempel and Ziv 1977, 1978] to compress the trajectory in the original form, we are only able to achieve a very low compression ratio. As the theoretical background of data compression is provided by information theory, we analyze the problem from the perspective of information entropy [Shannon 1948]. We represent the trajectory as continuous random variables and calculate its entropy, which shows the lower bound of the compressed trajectories' size. As stated in Theorem 3.2, both entropy coding and dictionary coding on original trajectories are *ineffective*.

An intuitive but informal explanation of Theorem 3.2 is that the frequency differences among symbols become smaller as the precision increases, which makes Huffman coding perform similarly as the sequential coding with a low compression ratio. Furthermore, Lempel-Ziv coding can hardly find repeated patterns in trajectories of a high precision, and thus its compression ratio is also low. In fact, the compression ratio drops to nearly 1 on floating-point numbers with accuracy of nine significant decimal digits,[2] which will be shown in Section 7.

THEOREM 3.2. *Both entropy coding and dictionary coding can only compress GPS trajectory data with a compression ratio close to 1 as the precision of floating-point numbers tends to be infinity.*

PROOF. First we prove that entropy coding is inefficient on compressing trajectory data. Let $X$ be a continuous random variable and $p(x)$ be the probability density function (PDF) of $X$. To calculate entropy of $X$, we divide the sample space $\Omega = [a, b)$ of $X$ into $n$ equal segments of size $\Delta$, as shown in Figure 2. Assume the interval $[a, b)$ is divided into $\{[a = x_0, x_1), [x_1, x_2), \ldots, [x_{n-1}, x_n = b)\}$. Then, we have probability of $x_i \le x < x_{i+1}$ stated in Equation (1):

$$P(x_i \le x < x_{i+1}) = \int_{x_i}^{x_{i+1}} p(x)dx. \tag{1}$$

Based on the mean-value theorem, there exists a value $x_{m_i}$ in each part that

$$p(x_{m_i})\Delta = \int_{x_i}^{x_{i+1}} p(x)dx.$$

[2]In our experimental studies, we represent both the longitude and latitude using a real number with accuracy of nine significant decimal digits.

In other words, we can calculate entropy of the discrete random variable as

$$\begin{aligned} H^{\Delta}(X) &= -\sum_{i=0}^{n} p(x_{m_i})\Delta \log(p(x_{m_i})\Delta) \\ &= -\sum_{i=0}^{n} (p(x_{m_i}) \log p(x_{m_i}))\Delta - \log \Delta = h^{\Delta}(X) - \log \Delta. \end{aligned}$$

If $p(x) \log p(x)$ is Riemannian integrable,

$$\begin{aligned} H(X) &= \lim_{\substack{n\to\infty \\ (\Delta\to 0)}} H^{\Delta}(X) = \lim_{\substack{n\to\infty \\ (\Delta\to 0)}} h^{\Delta}(X) - \lim_{\substack{n\to\infty \\ (\Delta\to 0)}} \log \Delta \\ &= -\int_a^b p(x) \log p(x) dx - \lim_{\substack{n\to\infty \\ (\Delta\to 0)}} \log \Delta = h(X) - \lim_{\substack{n\to\infty \\ (\Delta\to 0)}} \log \Delta, \end{aligned}$$

where $h(X)$ is a differential entropy of $X$.

Since $n$ here is the precision of real numbers in practice, we have to use $n$ unique symbols to represent all $n$ parts in the interval. In other words, we can represent an original real number with $\lceil \log n \rceil$ bits by a sequential coding. According to Shannon's source coding theorem [Shannon 1948] and the optimality of entropy encoding (e.g., Huffman coding and arithmetic coding), when $n \to \infty$, we have

$$\begin{aligned} \lim_{n\to\infty} \frac{1}{r} &= \lim_{\Delta\to 0} \frac{H^{\Delta}(X)}{\lceil \log n \rceil} = \lim_{n\to\infty} \frac{h^{\Delta}(X) - \log \Delta}{\lceil \log n \rceil} \\ &= \lim_{n\to\infty} \frac{h^{\Delta}(X) - \log(b-a) + \log n}{\lceil \log n \rceil} \\ &= (h(X) - \log(b-a)) \lim_{n\to\infty} \frac{1}{\lceil \log n \rceil} + \lim_{n\to\infty} \frac{\log n}{\lceil \log n \rceil} = 1. \end{aligned}$$

Consequently, the compression ratio tends to be 1 as the precision tends to be infinity.

The performance of other coding algorithms like LZW is bounded by joint entropy, so the proof is similar. Given any $k$, let $p(x_1, x_2, \ldots, x_k)$ be the joint PDF of $X_1, X_2, \ldots, X_k$. The average coding length $L_k$ is bounded according to Equation (2):

$$\frac{H(X_1, X_2, \ldots, X_k)}{k} \le L_k < \frac{H(X_1, X_2, \ldots, X_k)}{k} + \frac{1}{k}. \tag{2}$$

In other words, we have to use at least $H(X_1, X_2, \ldots, X_k)$ bits to encode $k$ symbols.

Then we divide the sample space $\Omega = [a_1, b_1) \times [a_2, b_2) \times \cdots \times [a_k, b_k)$ into $n^k$ equal parts of size $\prod_{i=1}^{k} \Delta_i$. Assume the interval $[a_i, b_i)$ is divided into $\{[a_i = x_{i,0}, x_{i,1}), [x_{i,1}, x_{i,2}), \ldots, [x_{i,n-1}, x_{i,n} = b_i)\}$; then we have

$$\begin{aligned} &P(x_{1,j_1} \le x_1 < x_{1,j_1+1}, x_{1,j_2} \le x_2 < x_{1,j_2+1}, \ldots, x_{k,j_k} \le x_k < x_{k,j_k+1}) \\ &= \int_{x_{1,j_1}}^{x_{1,j_1+1}} \int_{x_{2,j_2}}^{x_{2,j_2+1}} \cdots \int_{x_{k,j_k}}^{x_{k,j_k+1}} p(x_1, x_2, \ldots, x_k) dx_1 dx_2 \cdots dx_k. \\ &= p(x_{m_{1,j_1}}, x_{m_{2,j_2}}, \ldots, x_{m_{k,j_k}}) \prod_{i=1}^{k} \Delta_i. \end{aligned}$$

The entropy is calculated as

$$
\begin{aligned}
&H^{\Delta}(X_1, X_2, \ldots, X_k)\\
&= -\sum_{j_1=0}^{n}\sum_{j_2=0}^{n}\cdots\sum_{j_k=0}^{n} p(x_{m_{1,j_1}}, x_{m_{2,j_2}}, \ldots, x_{m_{k,j_k}})\prod_{i=1}^{k}\Delta_i \log\left(p(x_{m_{1,j_1}}, x_{m_{2,j_2}}, \ldots, x_{m_{k,j_k}})\prod_{i=1}^{k}\Delta_i\right)\\
&= -\log\prod_{i=1}^{k}\Delta_i - \sum_{j_1=0}^{n}\sum_{j_2=0}^{n}\cdots\sum_{j_k=0}^{n} p(x_{m_{1,j_1}}, x_{m_{2,j_2}}, \ldots, x_{m_{k,j_k}})\\
&\quad\times\log(p(x_{m_{1,j_1}}, x_{m_{2,j_2}}, \ldots, x_{m_{k,j_k}})\prod_{i=1}^{k}\Delta_i\\
&= -\sum_{i=0}^{k}\log\Delta_i + h^{\Delta}(X_1, X_2, \ldots, X_k).
\end{aligned}
$$

Assuming there are $k$ real numbers and their precision is $n$, then at least $H^{\Delta}(X_1, X_2, \ldots, X_k)$ bits are needed to encode them. If $p(x_1, x_2, \ldots, x_k)\log p(x_1, x_2, \ldots, x_k)$ is Riemannian integrable,

$$
\begin{aligned}
&H(X_1, X_2, \ldots, X_k)\\
&= \lim_{\substack{n\to\infty\\(\Delta\to 0)}} H^{\Delta}(X_1, X_2, \ldots, X_k) = \lim_{\substack{n\to\infty\\(\Delta\to 0)}} h^{\Delta}(X_1, X_2, \ldots, X_n) - \lim_{\substack{n\to\infty\\(\Delta\to 0)}}\sum_{i=1}^{k}\log\Delta_i\\
&= -\int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_k}^{b_k} p(x_1, x_2, \ldots, x_k)\log p(x_1, x_2, \ldots, x_k)dx_1dx_2\cdots dx_k - \lim_{\substack{n\to\infty\\(\Delta\to 0)}}\sum_{i=1}^{k}\log\Delta_i\\
&= h(X_1, X_2, \ldots, X_k) - \lim_{\substack{n\to\infty\\(\Delta\to 0)}}\sum_{i=1}^{k}\log\Delta_i,
\end{aligned}
$$

where $h(X_1, X_2, \ldots, X_k)$ is the joint differential entropy. Finally, the compression ratio $r$ of dictionary coding is calculated as

$$
\begin{aligned}
\lim_{n\to\infty}\frac{1}{r} &= \lim_{\Delta\to 0}\frac{H^{\Delta}(X_1, X_2, \ldots, X_k)}{k\times\lceil\log n\rceil} = \lim_{n\to\infty}\frac{-\sum_{i=0}^{k}\log\Delta_i + h^{\Delta}(X_1, X_2, \ldots, X_k)}{k\times\lceil\log n\rceil}\\
&= \lim_{n\to\infty}\frac{-\sum_{i=1}^{k}\log(b_i - a_i) + \sum_{i=1}^{k}\log n + h^{\Delta}(X_1, X_2, \ldots, X_k)}{k\times\lceil\log n\rceil}\\
&= \lim_{n\to\infty}\frac{1}{\lceil\log n\rceil}\times\lim_{n\to\infty}\frac{h^{\Delta}(X_1, X_2, \ldots, X_k) - \sum_{i=1}^{k}\log(b_i - a_i)}{k} + \lim_{n\to\infty}\frac{\log n}{\lceil\log n\rceil}\\
&= 0\times\frac{h(X_1, X_2, \ldots, X_k) - \sum_{i=1}^{k}\log(b_i - a_i)}{k} + 1 = 1. \quad \square
\end{aligned}
$$

Lossy compression is able to achieve a very high compression ratio, but it can hardly guarantee the utility of data after compression. As will be discussed in Section 8, many lossy trajectory compression algorithms actually remove certain sampled points from trajectories and hence compressed trajectory data suffer from high deviation from original data. Consequently, if the error caused by the compression is strictly bounded (e.g., tolerant error is small/zero and lossy compression is approximate/equivalent to lossless compression), the compression ratio of those lossy compression algorithms could

be very limited. Moreover, critical points like road junctions in trajectories cannot be discarded in order to preserve the geometric shape, which also leads to a low compression ratio. The limitations of the existing lossy 2D compression algorithms have been reported in Popa et al. [2015]. More details about lossless and lossy compression algorithms will be discussed in Section 8.

After careful analysis, we want to highlight that the ineffectiveness of trajectory compression is caused by data representation, regardless of algorithms used. In fact, both Huffman coding and Lempel-Ziv coding are optimal because their expected coding length achieves entropy (or entropy rate) of information sources, where an information source is a sequence of random variables, or a stochastic process. Information entropy that bounds the performance of data compression algorithms is a measure of uncertainty involved in the outcome of a stochastic process. More precisely, the more uncertainty the stochastic process involves, the more information the data contains, and the more bits are needed to encode the data. Conventional representation is suitable for arbitrary 2D trajectories. However, trajectories of moving objects in a road network capture the movements that are strictly restricted by the underlying road network, and the uncertainty of a trajectory in the road network is much smaller than that of an arbitrary one in a 2D plane. In other words, conventional representation involves unnecessary uncertainty (i.e., unnecessary information) in the data. This explains the reason that the compression ratio of trajectories in conventional representations is low though the compression algorithm applied is optimal.

### 3.2. Trajectory Decomposition

We apply dimensionality reduction to remove those unnecessary uncertainties in the data. Suppose we reduce three-dimensional data $\langle x_i, y_i, t_i \rangle$ to a lower-dimensional form of $(d_i, t_i)$ or even a one-dimensional form of $z_i$; then it is possible to raise the basic compression ratio up to $\frac{3}{2} = 1.5$ or $\frac{3}{1} = 3$. As we must construct a bijective mapping from the original form to the reduced form to make compression lossless, it is very challenging to achieve one-dimensional reduction. Consequently, we propose two-dimensional reduction to save some additional data.

The original triple $\langle x_i, y_i, t_i \rangle$ tells that the moving object is located at position $(x_i, y_i)$ at the timestamp $t_i$. Let $(x_1, y_1)$ be the starting point of the trajectory; the distance $d_i$ from the first point to the $i^{th}$ point is definite if the position $(x_i, y_i)$ of the $i^{th}$ point is known. If we know the distance $d_i$ a moving object has traveled from $t_1$ to $t_i$, the position $(x_i, y_i)$ corresponding to the timestamp $t_i$ is uncertain. Consequently, we record the spatial paths $(e_1, e_2, \ldots, e_m)$ passed by a trajectory $T$ to translate $d_i$ back to $(x_i, y_i)$, where $(e_1, e_2, \ldots, e_m)$ is a sequence of $m$ edges in $E$ that the moving object passes by via trajectory $T$. It is much easier to compress a sequence of edges and indeed we are able to compress $(e_1, e_2, \ldots, e_m)$ into a very small size. To be more specific, we decompose a trajectory $T$ into a *spatial path* and a *temporal sequence*, as defined in Definition 3.3 and Definition 3.4, respectively.

*Definition* 3.3 (*Spatial Path*). A spatial path of a trajectory $T$ is the edge sequence in the road network $G(V, E)$ that the trajectory passes by sequentially, denoted as $\mathcal{SP}_T = (e_1, e_2, \ldots, e_m)$.

*Definition* 3.4 (*Temporal Sequence*). A temporal sequence of a trajectory $T$ is a sequence of two-tuples $(d_i, t_i)$, where $d_i$ is the distance traveled by the moving object from the start point until the timestamp $t_i$ along trajectory $T$. $\mathcal{TS}_T = ((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$.

To illustrate the concept of trajectory decomposition, we plot a sample road network in Figure 3. In the sample road network, we have $V = \{v_1, v_2, \ldots, v_{12}\}$ and

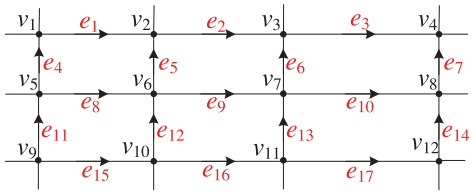



Fig. 3. Sample road network $G(V, E)$.

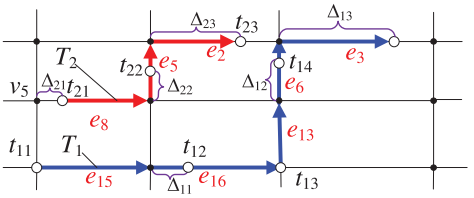



Fig. 4. Decomposition of trajectory $T$.

$E = \{e_1, e_2, \ldots, e_{17}\}$. We also show two sample trajectories $T_1$ and $T_2$ in Figure 4, with hollow dots along the trajectories representing the sampled GPS points. Take $T_1$ as an example. According to original representation, $T_1$ is represented as $(\langle x_{11}, y_{11}, t_{11}\rangle, \langle x_{12}, y_{12}, t_{12}\rangle, \langle x_{13}, y_{13}, t_{13}\rangle, \langle x_{14}, y_{14}, t_{14}\rangle, \langle x_{15}, y_{15}, t_{15}\rangle)$. Here, $x_i$ and $y_i$ capture the $x$-coordinator and $y$-coordinator of a point in the road network, respectively. According to trajectory decomposition, $T_1$ is decomposed into a spatial path $\mathcal{SP}_{T_1}$ and a temporal sequence $\mathcal{TS}_{T_1}$, with $\mathcal{SP}_{T_1} = (e_{15}, e_{16}, e_{13}, e_6, e_3)$, and $\mathcal{TS}_{T_1} = ((0, t_{11}),(w(e_{15})+\Delta_{11}, t_{12}),(w(e_{15})+w(e_{16}), t_{13}),(w(e_{15})+w(e_{16})+w(e_{13})+\Delta_{12}, t_{14}),(w(e_{15})+w(e_{16})+w(e_{13})+w(e_6)+\Delta_{13}, t_{15}))$. Note that if a trajectory does not start from the starting vertex of an edge, the distance $d_1$ associated with starting timestamp $t_1$ will be the distance from the starting point of the trajectory to the starting vertex of the edge. Take trajectory $T_2$ shown in Figure 4 as another example. Its starting point lies on the edge $e_8$, having $\Delta_{21}$ distance from the starting vertex of $e_8$ (i.e., $v_5$). Its spatial path $\mathcal{SP}_{T_2} = (e_8, e_5, e_2)$, and its temporal sequence $\mathcal{TS}_{T_1} = ((\Delta_{21}, t_{21}), (w(e_8) + \Delta_{22}, t_{22}), (w(e_8) + w(e_5) + \Delta_{23}, t_{23}))$.

After decomposing trajectories, a trajectory set is converted into a set of spatial paths and a set of temporal sequences. We then use different algorithms to compress them separately, according to their unique features. In the COMPRESS framework, we apply lossless data compression algorithms to spatial paths since they are sequences of integers that have relatively low entropy; meanwhile, we use an error-bounded lossy data compression algorithm to compress temporal sequences because compression of temporal sequences is more challenging. For a given trajectory $T$, as we can transform it between the original form and the decomposed form with time complexity of $O(|T|)$, it is easy to recover original trajectories after decompression of spatial paths and temporal sequences.

## 4. SPATIAL COMPRESSION

In this section, we focus on spatial path compression that takes $\mathcal{SP}_T$ in the form of $(e_1, e_2, \ldots, e_m)$ as input. Obviously, we can treat each edge in $\mathcal{SP}_T$ as an alphabet and regard $\mathcal{SP}_T$ as a string. In other words, we can employ existing lossless data compression algorithms to perform the compression. However, the alphabet size $|\Sigma|$ of $\mathcal{SP}_T$ equals the number of edges $|E|$ that is much larger than that of a normal alphabet set (e.g., English alphabet). For example, the California Road Network released by the Stanford Network Analysis Platform (SNAP) has over 2.7 millions edges. If we treat $\mathcal{SP}_T$ as a string and simply invoke coding algorithms to compress it, according to Theorem 3.2, the compression ratio will be very low when trajectories are in a road network with a large edge set and frequency differences between edges are not very notable. Furthermore, edges in a spatial path are not mutually independent. Consequently, entropy coding like Huffman coding is ineffective when symbols are not independent and identically distributed, while dictionary coding has a better compression capability because it can compress frequent substrings in the data. Since Huffman coding is not suitable for spatial compression, the original spatial compression algorithm proposed

Table I. Compression of $S_{in} = ababababa$ via LZW

| Input $S_{in}$ | Current String $W$ | Next Symbol | Output | Binary Output | New Entry in $\Sigma$ | $\lvert\Sigma\rvert$ |
|---|---|---|---|---|---|---|
| $\underline{a}babababa$ | $a$ | $b$ | $a$: 1 | 01 | $ab$: 3 | 4 |
| $\underline{b}abababa$ | $b$ | $a$ | $b$: 2 | 10 | $ba$: 4 | 5 |
| $\underline{ab}ababa$ | $ab$ | $a$ | $ab$: 3 | 011 | $aba$: 5 | 6 |
| $\underline{aba}ba$ | $aba$ | $b$ | $aba$: 5 | 101 | $abab$: 6 | 7 |
| $\underline{ba}$ | $ba$ | \0 | $ba$: 4 | 100 | | 7 |
| | \0 | NULL | \0: 0 | 000 | | 7 |

in Song et al. [2014] that is based on heuristic Huffman coding and takes $\theta$ symbols ($1 \leq \theta \leq 3$) as a supersymbol could be further improved.

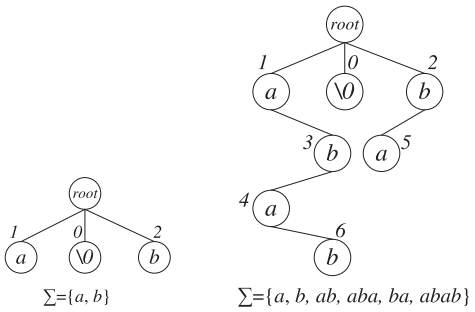
This observation inspires us to think of other ways to compress spatial paths. In the COMPRESS framework, we propose two new spatial compression algorithms catered for spatial path compression, namely, *Hybrid Dictionary Compression* and *Labeling and Coding*. HDC performs priming algorithms and then applies dictionary coding; L&C transforms the representation of spatial paths and then applies entropy coding. HDC preserves certain key spatial properties of spatial paths and supports efficient query operations on compressed trajectories; L&C can achieve a higher compression ratio and it is useful for archiving data or building backup. In the following, we explain these two algorithms in detail.

### 4.1. Hybrid Dictionary Compression

HDC takes a spatial path $\mathcal{SP}_T = (e_1, e_2, \ldots, e_m)$ as an input and performs lossless compression. In the following, we first review the LZW algorithm, one of the most commonly used dictionary coding algorithms. The LZW algorithm is ineffective on compressing spatial paths because the alphabet size is large while the length of a typical spatial path is short. Therefore, we introduce two priming methods, *Frequent Path Priming (FPP)* and *Shortest Path Priming (SPP)*, to build priming dictionaries. Afterward, HDC can compress spatial paths effectively via priming dictionaries.

*4.1.1. Review of Dictionary Coding.* Lempel and Ziv proposed two different versions of Lempel-Ziv coding, namely, the *LZ77 algorithm* [Lempel and Ziv 1977] and *LZ78 algorithm* [Lempel and Ziv 1978], in 1977 and 1978, respectively. The LZ77 algorithm maintains a sliding window as its dictionary, and the LZ78 algorithm builds a dictionary containing bijection between references and strings, usually in the form of a tree. Then, Welch enhanced LZ78 in 1984, which is known as the *Lempel-Ziv-Welch Algorithm* (LZW) [Welch 1984]. Given an input string $S_{in}$, LZW performs compression in four steps. In Step 1, it initializes the dictionary to contain all strings of length 1 via a *trie* [Knuth 1997]. In Step 2, it finds the *longest* string $W$ in the dictionary that matches the current input $S_{in}$. In Step 3, it outputs the index corresponding to $W$ and removes $W$ from $S_{in}$. In Step 4, it adds $W$ followed by the next symbol in the input $S_{in}$ to the dictionary, that is, adding a new node to Trie. It goes to Step 2 again to repeat this process until the ending of the input is reached.

To facilitate the understanding of LZW compression, we illustrate the main steps of compressing a given string $S_{in} = ababababa$ via LZW in Table I. If we assume the initial dictionary $\Sigma = \{a, b\}$, LZW will initialize a trie containing three nodes, for three strings of length 1 (i.e., $a$, $b$, \0), as shown in Figure 5(a). Note that the symbol \0 represents the end of strings. The number next to each node, as depicted in Figure 5(a), represents the code (in decimal format) corresponding to each symbol in $\Sigma$. We list the codes, in both decimal format and binary format, in Figure 6. For example, for symbol $a \in \Sigma$, its code is 1 in decimal and 01 in binary format; for symbol $b \in \Sigma$, its code is 2 in decimal and 10 in binary format. We want to highlight that the number




(a) Initial Trie (b) Final Trie

Fig. 5. Example Trie.

| $S$ | $code_d$ | $code_b$ |
|---|---|---|
| $\backslash 0$ | 0 | 00 |
| $a$ | 1 | 01 |
| $b$ | 2 | 10 |
| $ab$ | 3 | 011 |
| $ba$ | 4 | 100 |
| $aba$ | 5 | 101 |
| $abab$ | 6 | 110 |

Fig. 6. Symbols $S \in \Sigma$ and their codes.

of binary bits used to represent a decimal code is dependent on the size of current $\Sigma$. For example, as $|\Sigma| = 3$ initially, two binary bits are sufficient to code all the symbols currently captured by $\Sigma$. When the fifth symbol is introduced to $\Sigma$, the algorithm will have to switch at that point from two binary bits to three binary bits to represent all the codes, including those that were previously represented by only two binary bits.

After the initialization, LZW scans the input string $S_{in}$ letter by letter. First, it finds $a$ in Trie, which matches the prefix of $S_{in}$. It enrolls the binary code of $a$ (i.e., 01) to the output. In addition, it forms a new symbol $ab$ based on $a$ and its next symbol in $S_{in}$, inserts a new node corresponding to $ab$ to Trie, and removes $a$ from the input string. Accordingly, a new code (i.e., 3 in decimal) is assigned to $ab$. Next, it finds $b$ in Trie, which matches the prefix of current $S_{in}$. It enrolls the binary code of $b$ (i.e., 10) to the output. Similarly, it also forms a new symbol $ba$ based on $b$ and its next symbol in $S_{in}$, inserts a new node corresponding to $ba$ to Trie, and removes $b$ from the input string. Accordingly, a new code (i.e., 4 in decimal) is assigned to $ba$. Note that, as $\Sigma = \{a, b, ab, ba\}$ (and $\backslash 0$) reaches the size of five, binary codes in 2-bit forms are not sufficient. That's why the next output (and following outputs as well) will use 3 bits. Then, it finds $ab$ in Trie that matches the prefix of current $S_{in}$. It enrolls the binary code of $ab$ (i.e., 011) to the output. Again, it also forms a new symbol $aba$ based on $ab$ and its next symbol $a$ in $S_{in}$, inserts a new node corresponding to $aba$ to Trie, and removes $ab$ from the input string. Accordingly, a new code (i.e., 5 in decimal) is assigned to $aba$. The process continues until the input string becomes empty. The final output is 01 10 011 101 100 000. It is obvious that LZW compresses the input string by scanning the string once, so it has a linear time complexity $O(|S_{in}|)$.

Each row in Table I corresponds to one step described previously. The input string in the original form contains 10 symbols, including the end symbol $\backslash 0$, with each taking 2 bits, while the output string in the compressed form occupies 16 bits. In other words, using LZW has saved $(2 \times 10 - 16)$ out of 20, achieving a compression ratio of 1.25.

The decompression algorithm works by reading a value from the encoded input and outputting the corresponding symbol from the initialized dictionary $\Sigma$. Note we only need to know the initial dictionary (e.g., $\Sigma = \{a, b\}$ in our example) but not the extended dictionary as additional symbols can be reconstructed based on initial entries in $\Sigma$. We continue previous example and list the main steps of decompressing 0110011101100000 via LZW in Table II. Initially, $|\Sigma| = 3$ and the binary code is in the form of 2 bits. That explains why the first two rows in Table II read input as 2-bit code. After the first two steps listed in the first two rows, $\Sigma = \{a, b, ab, ba\}$ and hence the algorithm starts reading input as 3-bit codes. As the decompression algorithm is very

Table II. Decompression of 0110011101100000 via LZW

| Input | Current Code | Output | New Entry in $\Sigma$ |
|---|---|---|---|
| 0110011101100000 | 01 | $a$ | $ab$: 3 |
| 10011101100000 | 10 | $b$ | $ba$: 4 |
| 011101100000 | 011 | $ab$ | $aba$: 5 |
| 101100000 | 101 | $aba$ | $abab$: 6 |
| 100000 | 100 | $ba$ | |
| 000 | 000 | \0 | |

similar to the compression algorithm, its details are skipped here to avoid redundancy. Again, the decompression algorithm has linear time complexity.

*4.1.2. Frequent Path Priming (FPP).* Trajectories are not evenly distributed within the road network, and edges in a road network are not accessed uniformly. In other words, certain edge sequences are much more popular than others in terms of frequency. If we are able to locate popular subspatial paths, named *frequent subspatial paths* (FSPs), we can use a certain coding scheme to compress them and to replace them in the trajectories with the corresponding codes.

Given a large set of trajectory data, the concept of FSP makes sense. Consequently, the compression based on FSP is not effective if the underlying dataset is small. In addition, we also assume given the trajectories of all the moving objects (e.g., cars, buses) moving within a city for a duration of several months, the dataset of 1 day should be similar to the dataset of another day. Under this assumption, we can locate FSPs based on a subset of the complete trajectory dataset, which corresponds to the training process in data mining. In the following, we explain how to use dictionary coding like LZW to mine FSPs and to perform spatial path compression based on FSPs.

There are almost no repeated patterns in a single spatial path, so using LZW to compress a single spatial path is ineffective. However, different spatial paths could share the same patterns, and hence we can compress a large number of spatial paths together to improve the compression ratio of LZW. If we simply adopt LZW, we have to recompress all spatial paths every time a new trajectory is added to our collection because trajectories are always sampled separately in practice. A simple adoption of LZW might result in very expensive time complexity that we cannot afford, and we need an effective algorithm to compress each single spatial path independently, though compressing spatial paths together improves the compression ratio. As a solution, we use the *Priming Lempel-Ziv-Welch Algorithm (PLZW)* to mine FSPs and then to compress spatial paths based on mined FSPs.

Consider the process of the LZW algorithm; given an entry $S$ in the dictionary, all prefixes of $S$ must have appeared at least once in the previous part of the input string. Furthermore, given a node $n_i$ in the trie, the entry $S$ represented by the node $n_i$ must have appeared at least $(k + 1)$ times, with $k$ being the number of $n_i$'s descendant nodes. During the process of compression, the LZW algorithm gradually adds new entries into the dictionary and we can regard all those entries in the trie as FSPs. In order to mine FSPs, we start with an initial dictionary and use the LZW algorithm to compress spatial paths in the training dataset one by one. Different from the original LZW algorithm, we do not abandon the dictionary created by the former spatial paths but extend the dictionary by compressing the latter spatial paths. Finally, we will create a large priming dictionary that contains enough FSPs. Thereafter, we use the compression algorithm based on the priming dictionary to compress the spatial paths of new trajectories.

Different from LZW, algorithm PLZW constructs a priming dictionary from the training dataset and stores the dictionary for further compression and decompression. It

| | |
|---|---|
| $\mathcal{SP}_{T_1}$ | $(e_8, e_5, e_2, e_3)$ |
| $\mathcal{SP}_{T_2}$ | $(e_{15}, e_{12}, e_9, e_{10})$ |
| $\mathcal{SP}_{T_3}$ | $(e_{12}, e_5, e_2, e_3)$ |

Fig. 7. Training spatial path set $S_{train}$.

| | |
|---|---|
| before training | $\Sigma = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}\}$ |
| after training | $\Sigma = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_2e_3, e_5e_2, e_5e_2e_3, e_8e_5, e_9e_{10}, e_{12}e_5, e_{12}e_9, e_{15}e_{12}\}$ |

Fig. 8. Content of dictionary.

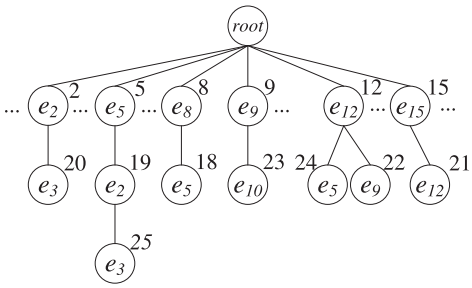



Fig. 9. Training dictionary of $S_{train}$.

shares the same compression and decompression process as the LZW algorithm as we only replace the initial dictionary with a dictionary that contains more entries. Whereas the training data is sufficiently large to generate a priming dictionary with enough FSPs, we will *not* append any new entries to the dictionary during the compression of real trajectories. In addition, we maintain an index (e.g., a balanced binary search tree [Bayer 1972]) in each node of the trie to facilitate the search for the specific child nodes efficiently during compression because the dictionary is large and so is the number of child nodes of each node. The time complexity of algorithm PLZW is $O(\sum |T_i| \log |E|)$ for the training stage and $O(|T| \log |E|)$ for the compression stage, where $|T|$ is the length of the new trajectory to compress and $\sum |T_i|$ is the total length of trajectories in the training dataset. Each trie node has at most $|E|$ child nodes, so the balanced search tree contains at most $|E|$ elements. As compared with LZW, the complexity of compression is multiplied by $O(\log |E|)$ as we need to search in indexes. The complexity of decompression with PLZW is $O(|T|)$ since we just follow the parent nodes until reaching the root to recover the FSP from the reference to a trie node. The space complexity is also guaranteed because the size of the training dictionary will not exceed the size of the training set.

Take spatial paths listed in Figure 7 as an example training spatial path set, based on the road network depicted in Figure 3. Before the start of the training stage, the dictionary contains 18 entries of length 1, including 17 edges in the sample road network and \0, as listed in the first row of Figure 8. In the training stage, we compress three spatial paths one by one using the PLZW algorithm. First, we compress $\mathcal{SP}_{T_1}$ and append three new entries $\{e_8e_5\}$, $\{e_5e_2\}$, and $\{e_2e_3\}$ to the dictionary. Next, we compress $\mathcal{SP}_{T_2}$ and append another three new entries $\{e_{15}e_{12}\}$, $\{e_{12}e_9\}$, and $\{e_9e_{10}\}$ to the dictionary. Finally, we compress $\mathcal{SP}_{T_3}$ and introduce two new entries. The dictionary after the training stage contains $18 + 8$ $(= 26)$ entries including \0, as listed in the second row of Figure 8.

In the compression stage, PLZW compresses a given spatial path based on the dictionary constructed during the training stage without changing the content of the dictionary. Take $\mathcal{SP}_{T_4} = (e_{15}, e_{12}, e_5, e_2, e_3)$ as an example. PLZW will compress it by dividing it into three subpaths with each corresponding to an entry in the dictionary, that is, $e_{15}e_{12}$ (decimal code: 21, binary code: 10101), $e_5e_2e_3$ (decimal code: 25, binary code: 11001), and \0 (decimal code: 0, binary code: 00000). $\mathcal{SP}_{T_4}$ is encoded into 10101

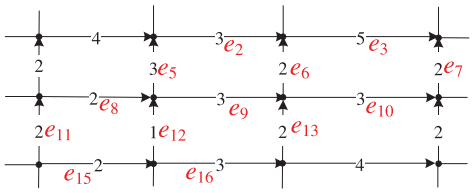



Fig. 10. Sample of shortest-path priming.

| $T$ | $\mathcal{SP}_T$ | $\mathcal{SP}_T^c$ |
|---|---|---|
| $T_1$ | $(e_{15}, e_{12}, e_9, e_{10})$ | $(e_{15}, e_{10})$ |
| $T_2$ | $(e_{15}, e_{16}, e_{13}, e_6, e_3)$ | $(e_{15}, e_{13}, e_3)$ |
| $T_3$ | $(e_{11}, e_8, e_5, e_2, e_3)$ | $(e_{11}, e_5, e_3)$ |
| $T_4$ | $(e_{11}, e_8, e_9, e_6, e_3)$ | $(e_{11}, e_3)$ |

Fig. 11. SPC of sample trajectories.

11001 00000. $\mathcal{SP}_{T_4}$ in the original form is a string of length 6 including \0 and takes the space of $6 \times 5$ bits, and $\mathcal{SP}_{T_4}$ in the compressed form takes 15 bits. In other words, algorithm PLZW achieves a compression ratio of 2 for $\mathcal{SP}_{T_4}$.

The larger the training dataset we use, the more the FSPs PLZW can mine, which leads to a better compression ratio, with additional storage cost to store the larger dictionary. We will study the balance between the dictionary size and the compression ratio in our experiments.

*4.1.3. Shortest-Path Priming (SPP).* Given a source *start* and a destination *end*, most of the time we will take the shortest path from *start* to *end* if all the edges share a similar traffic condition. Shortest paths hence are patterns that frequently appear in spatial paths. In Song et al. [2014], we already proposed *Shortest-Path Compression (SPC)*, which compresses the spatial paths by skipping certain subpaths.

We assume that all-pair shortest-path information is available via a preprocessing of the road network. This can be achieved by any of the well-known shortest-path algorithms, for example, Dijkstra's algorithm [Dijkstra 1959]. If there are several shortest paths between a pair of edges, we only record one of them to eliminate any ambiguity during compression. Assume $SP(e_i, e_j)$ donates the shortest path from edge $e_i$ to edge $e_j$, and a dictionary $Prev(e_i, e_j)$ is maintained for each pair of edges that records the last edge (the edge right before $e_j$) of $SP(e_i, e_j)$. Take the partial road network shown in the Figure 10 as an example. Assume the number in the middle of each edge indicates the network distance of the edge, and then $SP(e_{15}, e_7) = (e_{15}, e_{12}, e_9, e_{10}, e_7)$, $Prev(e_{15}, e_7) = e_{10}$, $Prev(e_{15}, e_{10}) = e_9$, $Prev(e_{15}, e_9) = e_{12}$, and so on.

Different from frequent path priming, we do not have to represent shortest paths with certain codes because the source and the destination of a shortest path themselves are enough to locate it in the dictionary. The main idea is to skip the detailed subtrajectory $(e_i, e_{i+1}, \ldots, e_j)$ if it matches exactly the shortest path from $e_i$ to $e_j$, that is, replacing $SP(e_i, e_j)$ with $(e_i, e_j)$. We list the spatial paths of four trajectories in Figure 11. For $\mathcal{SP}_{T_1}$, $(e_{15}, e_{12}, e_9, e_{10})$ is exactly the shortest path from $e_{15}$ to $e_{10}$, according to the partial road network depicted in Figure 10. Consequently, we can use two-edge tuple $(e_{15}, e_{10})$ to represent the original $\mathcal{SP}_{T_1}$. Similarly, for $\mathcal{SP}_{T_2}$, its subspatial paths $(e_{13}, e_6, e_3)$ and $(e_{15}, e_{16}, e_{13})$ are the shortest path from $e_{13}$ to $e_3$ and that from $e_{15}$ to $e_{13}$, respectively. Consequently, we can represent $(e_{13}, e_6, e_3)$ using a two-edge tuple $(e_{13}, e_3)$ and represent $(e_{15}, e_{16}, e_{13})$ using $(e_{15}, e_{13})$. This explains why $\mathcal{SP}_{T_2}$ is represented by $(e_{15}, e_{13}, e_3)$.

Obviously, there are multiple ways to implement SPC. The detailed implementation of SPC is presented in Song et al. [2014], which is a linear greedy algorithm that generates the path that contains the fewest edges. Although in practice people may not take the shortest path all the time, many subsequences of a long trajectory are actually shortest paths. Consequently, SPC will always achieve a medium compression ratio. Spatial compression in Song et al. [2014] takes SPC as a necessary part because it contributes a lot in terms of compression ratio, but we show that SPC is only an alternative priming stage in HDC in the following.

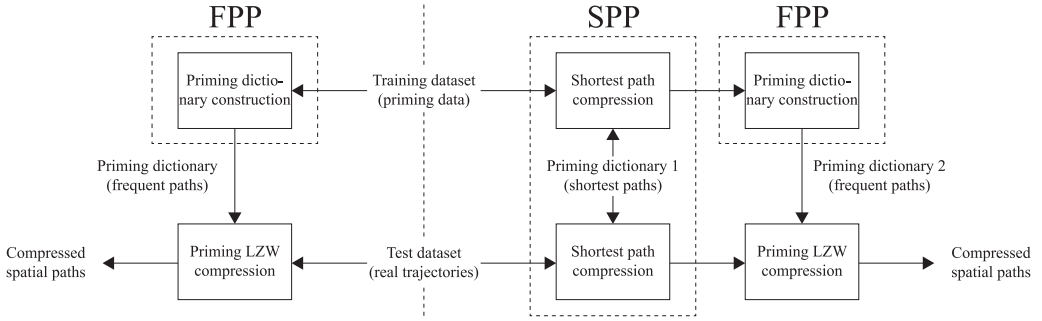



Fig. 12. Two strategies of HDC.

*4.1.4. Combination of SPP and FPP.* SPP and FPP are not mutually exclusive and they can be combined to achieve the advantages of both SPP and FPP. In other words, we can process the spatial paths via SPC first and then by PLZW. To be more specific, we divide the original trajectory set into two disjoint subsets, including a training set and a test set, where both subsets are compressed via SPC first. Then, we mine FSPs from the training set and use those mined FSPs to compress trajectories in the test set via PLZW. We perform SPC for each trajectory in the training dataset and then invoke PLZW in the second stage to mine FSPs. For the test set, we perform SPC and then PLZW.

Given the fact that FPP is able to mine all FSPs including shortest paths via a sufficiently large training set, SPP becomes optional in HDC. This finding suggests that there are two ways to implement HDC in practice, depending on whether SPP is used, as shown in Figure 12. As will be shown in Section 7, the performance difference between two strategies (i.e., the one without SPP and the other with SPP) becomes smaller when the training set size grows. Considering the relatively expensive time and space costs of SPP (i.e., $O(V^2)$), we suggest adopting the strategy without SPP with a large training set if the road network is extremely big and the preprocessing cost of SPP becomes unaffordable. On the contrary, if a road network is small (e.g., $|V| = 60{,}456$ and $|E| = 132{,}207$ in the real dataset we use in the experimental study) and the size of the training data is not sufficiently big, the strategy with SPP should be employed since the preprocessing is one-off.

Last but not least, we want to highlight that the dictionaries generated by SPP and FPP store not only entries but also some useful additional information to support online LBS applications. In other words, the dictionaries can serve as an index. More details will be discussed in applications on compressed trajectories in Section 6.

### 4.2. Labeling and Coding

In addition to HDC, which can effectively compress spatial paths, we propose the *Labeling and Coding* algorithm as an alternative. HDC is based on dictionary coding because low relevance among edges results in ineffectiveness of entropy coding. Different from HDC, L&C first labels every edge in the road network $G$ in the *labeling* stage, after which each spatial path can be translated into a label sequence. The translation summarizes the relevance among edges and thereafter labels are almost independent of each other. Then, in the *coding* stage, it invokes an entropy coding algorithm to compress label sequences. Compared with HDC, L&C can achieve an even higher compression ratio. However, the further reduced storage overhead comes with a loss of spatial properties so that the compressed spatial paths cannot support any LBS application unless fully decompressed.

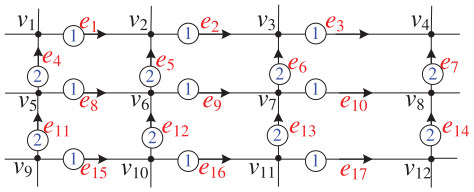



Fig. 13. Example of edge labeling.

| $T$ | $\mathcal{SP}_T$ | label sequence |
|---|---|---|
| $T_1$ | $(e_8, e_5, e_2, e_3)$ | $(e_8, 2, 1, 1)$ |
| $T_2$ | $(e_{15}, e_{12}, e_9, e_{10})$ | $(e_{15}, 2, 1, 1)$ |
| $T_3$ | $(e_{12}, e_5, e_2, e_3)$ | $(e_{12}, 2, 1, 1)$ |

Fig. 14. Training spatial path set.

*4.2.1. Labeling Stage.* In a road network $G = (V, E)$, the max out-degree $D = \max_{v_i \in V}(d_{out}(v_i))$ is small, where $d_{out}(v)$ denotes the out-degree of a vertex $v$. Given an edge $e_{ij}$ from $v_i$ to $v_j$, the number of potential subsequent edges $e_{jl}$ from $v_j$ to $v_l$ in a spatial path is equivalent to $d_{out}(v_j)$ that is bounded by $D$. If we assign different labels to immediate successors of every edge, we can simply locate the next edge in the spatial path if both the label and the previous edge are known.

Take our sample road network as an example, with labels for each edge shown in Figure 13. Vertex $v_6$ has its out-degree as 2, which means it has only two subsequent edges from $v_6$, that is, $e_9$ labeled by 1 and $e_5$ labeled by 2. In other words, the sequence of $(e_8, 2)$ represents the path from $e_8$ to $e_5$. Similarly, vertex $v_{10}$ has its out-degree as 2, with one subsequent edge $e_{16}$ labeled by 1 and the other subsequent edge $e_{12}$ labeled by 2. Consequently, the sequence of $(e_{15}, 2)$ represents the path from $e_{16}$ to $e_{12}$. We also list the label sequence of the spatial paths corresponding to three sample trajectories in Figure 14 as examples.

Let $|\mathcal{SP}|$ denote the length of spatial path $\mathcal{SP}$. The bit length of $\mathcal{SP}$, denoted as $|\mathcal{SP}|_{(2)}$, can be derived based on Equation (3), and the bit length of the labeled spatial path, denoted as $|\mathcal{SP}_L|_{(2)}$, can be derived based on Equation (4). Given the fact that the max out-degree of $G$ is much smaller than the size of the edge set (i.e., $D \ll |E|$), the storage cost is reduced significantly, as we can see from Equation (3) and Equation (4):

$$|\mathcal{SP}|_{(2)} = |\mathcal{SP}|\lceil \log_2(|E|)\rceil \tag{3}$$

$$|\mathcal{SP}_L|_{(2)} = \lceil \log_2(|E|)\rceil + (|\mathcal{SP}|-1)\lceil \log_2(D)\rceil. \tag{4}$$

Even though the labeled spatial path has a much smaller size than the spatial path, we can further compress the label sequence. Because labeling helps to reduce the size of the label sequence's alphabet to $D$, lossless data compression algorithms become *effective*, as stated in Theorem 3.2. In addition, we can further improve the compression ratio of existing compression algorithms by optimizing the label of the graph. Note the only requirement of our labeling algorithm is that out-edges of any vertex should have *different* labels. In other words, there are lots of ways to label the graph. In the following, we propose a labeling algorithm to construct optimal labels, which allow Huffman coding on the label sequence to achieve the *highest* compression ratio. Before we present the labeling algorithm and prove its optimality, we first introduce some important definitions.

*Definition* 4.1 (*Label of an Edge*). A label of an edge $e$ is an integer $L(e)$ that distinguishes the edge $e$ from other edges that share the same starting vertex as $e$.

*Definition* 4.2 (*Label of a Graph*). A label of a graph $G$ is a mapping $L_G$ from the edge set $E$ to an integer set $\mathbb{L}$, where $\mathbb{L} = \{1, 2, \ldots, D\}$ and $D = \max_{v \in V}(d_{out}(v))$. All out-edges of a vertex $v$, denoted as $E_v \subset E$, must have different labels, that is, $\forall v \in V$, $\forall e_i, e_j (\neq e_i) \in E_v$, $L(e_i) \neq L(e_j)$.

*Definition* 4.3 (*Label of a Vertex*). A label of a vertex $v$ is a bijective mapping $L_v$ from the out-edge set $E_v$ of $v$ to an integer set $\mathbb{L}$, where $\mathbb{L} = \{1, 2, \ldots, d_{out}(v)\}$.

Next, we present the *Minimal Entropy Labeling (MEL) Algorithm* to improve the label of the graph. MEL takes the road network $G$ and frequency of edges as an input and labels out-edges of every vertex according to their frequency, for example, label 1 for the most frequent out-edge of $e$, label 2 for the second most frequent out-edge of $e$, and so on. Similar as FPP, we use a large set of trajectory data as a training set to count the frequency of edges. For example, vertex $v_{10}$ has four out-edges $e_{16}$, $e_{12}$, $e_{19}$, and $e_{23}$. Among the spatial paths of trajectories in our training set, 100 spatial paths pass $e_{16}$, 700 spatial paths pass $e_{12}$, 400 spatial paths pass $e_{19}$, and 200 spatial paths pass $e_{23}$. Consequently, we have $f(e_{16}) = 100$, $f(e_{12}) = 700$, $f(e_{19}) = 400$, and $f(e_{23}) = 200$. MEL then labels these four edges based on their frequencies, that is, $L(e_{12}) = 1$, $L(e_{19}) = 2$, $L(e_{23}) = 3$, and $L(e_{16}) = 4$.

**ALGORITHM 1:** Minimal Entropy Labeling (MEL) Algorithm

**Input:** A road network $G = (V, E)$ and frequency $f(e)$ of every edge in $E$
**Output:** A label of the road network $G$

```
1: for each vertex v in G do
2:     Sort out-edges e of v by frequency f(e) in descending order;
3:     currLabel ← 1;
4:     for each out-edge e ∈ E_v in sorted order do
5:         L(e) ← currLabel, currLabel ← currLabel + 1;
```

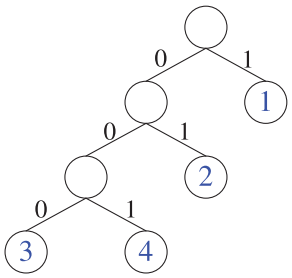
Algorithm 1 lists the pseudo-code for MEL. It scans every vertex and sorts out-edges of each vertex with the time complexity of $O(|V|D \log D)$. Though MEL itself is simple, it constructs a label of the graph that is optimal for Huffman coding to compress label sequences. We will prove the optimality of MEL in next subsection.

*4.2.2. Coding Algorithm.* After labeling the graph, we can generate labeled spatial paths and then use a lossless data compression algorithm to compress them. Different compression algorithms may require different labeling methods in order to achieve an optimal compression ratio. In our L&C algorithm, MEL is designed for entropy encoding, so we use entropy encoding algorithms to compress label sequences in order to achieve optimality. Note that all entropy encoding algorithms are acceptable and here we illustrate the coding algorithm by Huffman coding just for convenience. In experiments, we implement another entropy encoding algorithm, arithmetic coding [Rissanen 1976], for comparison.

To compress the labeled spatial paths, we apply the Huffman coding algorithm. First, the algorithm enrolls the head edge in the labeled spatial path into the result. It is obvious that the head edge occupies $\log_2 |E|$ bits in the result code. Then, the coding algorithm encodes the following labels by Huffman coding. We define the frequency of the label $l$ as the number of edges in $E$ that are labeled as $l$ in Equation (5). For example, we have a labeled graph $G$ of 100 edges, among which 40 are labeled by 1, 30 labeled by 2, 20 labeled by 3, and 10 labeled by 4. Then, we have $F(1) = 40$, $F(2) = 30$, $F(3) = 20$, and $F(4) = 10$.

$$F(l) = |\{e \in E | L(e) = l\}| \tag{5}$$

Based on the frequencies $F(l)$ of different labels, we build a Huffman tree by $F(l)$ to replace labels with Huffman codes. As depicted in Figure 15, we have $code_H(1) = 0$, $code_H(2) = 01$, $code_H(3) = 000$, and $code_H(4) = 001$. Given a labeled spatial path $\mathcal{SP}_L = (e_{15}, 1, 2, 2, 1)$, the coding algorithm first enrolls head edge $e_{15}$ into the result.




| label | frequency | code |
|---|---|---|
| 1 | 40 | 1 |
| 2 | 30 | 01 |
| 3 | 20 | 000 |
| 4 | 10 | 001 |

Fig. 15. Sample Huffman tree for labels.

As there are in total 17 edges in our sample road network, each edge is represented by a 5-bit binary code. In other words, the result code becomes 01111. By adding Huffman codes of labels into the result, we get 01111 0 01 01 0.

Since we choose Huffman coding to compress $\mathcal{SP}_L$, we can assume that the data source is a memoryless process. Let $X$ be a discrete random variable to represent the data source, of which the sample space $\Omega = \mathbb{L} = \{1, 2, \ldots, D\}$, and we can calculate probability mass function $p(x)$ as

$$p(x) = \frac{F(x)}{\sum_{i=1}^{D} F(i)}.$$

We define the entropy $H(X)$ as the entropy $H(L_G)$ of the label of the graph. In the end of this section, we prove that MEL constructs a label with the *lowest* entropy.

THEOREM 4.4 (OPTIMALITY OF MEL). *MEL constructs a label of the graph $L_G^M$ with the* lowest *entropy.*

PROOF. To facilitate the proof of the optimality of MEL, we define some symbols first. We represent the mapping $L_v$ as a permutation $\sigma_v$ to show details of $L_v$. For example, the label of $v$ will be represented as

$$\sigma_v = \begin{pmatrix} e_1 & e_2 & \cdots & e_d \\ l_1 & l_2 & \cdots & l_d \end{pmatrix}.$$

The frequency of edges labeled by $x$ and meanwhile not started from vertex $v$ is defined as

$$g_v(l) = \sum_{L_e = l \wedge st(e) \neq v} f(e),$$

where $st(e)$ represents the starting vertex of $e$. Thus, if the label of $e$ is $l$ with $st(e) = v$, $g_v(l) + f(e) = F(l)$.

In the first part of our proof, we define a property of $L_v$ named *adjustability*. Given any label $L_G$ of the graph $G$, we can sort the labels of $v$ according to Inequation (6) and sort the out-edges of $v$ according to Inequation (7). Then, we can give labels to out-edges of $v$ according to these two inequations:

$$g_v(l_1) \geq g_v(l_2) \geq \cdots \geq g_v(l_d) \tag{6}$$

$$f(e_1) \geq f(e_2) \geq \cdots \geq f(e_d) \tag{7}$$

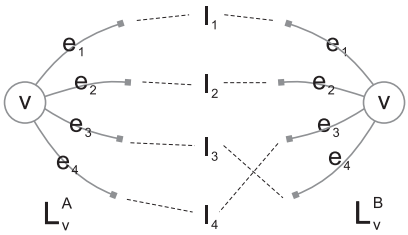



Fig. 16. Sample label $L_v^A$ and $L_v^B$.

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $g_v(l_i)$ | 30 | 50 | 20 | 35 |
| $f(e_i)$ | 4 | 12 | 10 | 1 |

Fig. 17. Sample $g_v(l_i)$ and $f(e_i)$.

$$\sigma_v^N = \begin{pmatrix} e_1 & e_2 & \cdots & e_d \\ l_1 & l_2 & \cdots & l_d \end{pmatrix}$$

$$\sigma_v' = \begin{pmatrix} e_1 & e_2 & \cdots & e_d \\ m_1 & m_2 & \cdots & m_d \end{pmatrix}.$$

Furthermore, we call labels of vertexes like $\sigma_v^N$ *nonadjustable* labels. On the contrary, if a label $\sigma_v'$ cannot satisfy either Inequation (6) or Inequation (7), it is *adjustable*. Take Figure 16 and Figure 17 as an example. $\sigma_v^A$ is adjustable, but $\sigma_v^B$ is nonadjustable:

$$\sigma_v^A = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 \\ l_1 & l_2 & l_3 & l_4 \end{pmatrix} = \begin{pmatrix} e_2 & e_3 & e_1 & e_4 \\ l_2 & l_3 & l_1 & l_4 \end{pmatrix} \sim \begin{pmatrix} f(e_2):12 & f(e_3):10 & f(e_1):4 & f(e_4):1 \\ g_v(l_2):50 & g_v(l_3):20 & g_v(l_1):30 & g_v(l_4):35 \end{pmatrix};$$

$$\sigma_v^B = \begin{pmatrix} e_4 & e_3 & e_2 & e_1 \\ l_3 & l_4 & l_2 & l_1 \end{pmatrix} = \begin{pmatrix} e_2 & e_3 & e_1 & e_4 \\ l_2 & l_4 & l_1 & l_3 \end{pmatrix} \sim \begin{pmatrix} f(e_2):12 & f(e_3):10 & f(e_1):4 & f(e_4):1 \\ g_v(l_2):50 & g_v(l_4):35 & g_v(l_1):30 & g_v(l_3):20 \end{pmatrix}.$$

We will construct a series of permutations from an arbitrary label $\sigma_v'$ to a nonadjustable label $\sigma_v^N$ in order to show a decrease of the entropy $H(L_G)$. To do this, initially we set $\sigma_v'$ as $\sigma_v^0$. Next, we swap $l_1$ with the first element in $\sigma_v^0$ and we get $\sigma_v^1$. Then, we swap $l_2$ with the second element in $\sigma_v^1$ to get $\sigma_v^2$. Keep doing this and finally we get $\sigma_v^k = \sigma_v^N$, which is nonadjustable. For example, given a label of $v$, if

$$g_v(l_1) \geq g_v(l_2) \geq \cdots \geq g_v(l_d)$$

$$f(e_1) \geq f(e_2) \geq \cdots \geq f(e_d),$$

then we have

$$\sigma_v' = \sigma_v^0 = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ \boldsymbol{l_6} & \boldsymbol{l_1} & l_5 & l_2 & l_4 & l_3 \end{pmatrix}$$

$$\sigma_v^1 = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ l_1 & \boldsymbol{l_6} & l_5 & \boldsymbol{l_2} & l_4 & l_3 \end{pmatrix}$$

$$\sigma_v^2 = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ l_1 & l_2 & \boldsymbol{l_5} & l_6 & l_4 & \boldsymbol{l_3} \end{pmatrix}$$

$$\sigma_v^3 = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ l_1 & l_2 & l_3 & \boldsymbol{l_6} & \boldsymbol{l_4} & l_5 \end{pmatrix}$$
$$\sigma_v^4 = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ l_1 & l_2 & l_3 & l_4 & \boldsymbol{l_6} & \boldsymbol{l_5} \end{pmatrix}$$
$$\sigma_v^N = \sigma_v^5 = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{pmatrix}.$$

Consider the $i^{th}$ step to estimate the change of entropy:

$$\sigma_v^i = \begin{pmatrix} e_1 & e_2 & \cdots & e_i & e_{i+1} & \cdots & e_j & \cdots \\ l_1 & l_2 & \cdots & l_i & x & \cdots & l_{i+1} & \cdots \end{pmatrix}$$
$$\sigma_v^{i+1} = \begin{pmatrix} e_1 & e_2 & \cdots & e_i & e_{i+1} & \cdots & e_j & \cdots \\ l_1 & l_2 & \cdots & l_i & l_{i+1} & \cdots & x & \cdots \end{pmatrix}.$$

Then, we can calculate the frequency change of $l_{i+1}$ and $x$:

$$F^{i+1}(l_{i+1}) = g_v(l_{i+1}) + f(e_{i+1})$$
$$F^i(l_{i+1}) = g_v(l_{i+1}) + f(e_j)$$
$$F^{i+1}(x) = g_v(x) + f(e_j)$$
$$F^i(x) = g_v(x) + f(e_{i+1}).$$

Since $x \geq i+1$, it is clear that $g_v(l_{i+1}) \geq g_v(x)$ and $f(e_{i+1}) \geq f(e_j)$ according to Inequation (6) and Inequation (7). Consequently, we have

$$\begin{aligned} F^{i+1}(x) &= g_v(x) + f(e_j) \leq g_v(l_{i+1}) + f(e_j) = F^i(l_{i+1}) \\ F^i(l_{i+1}) &= g_v(l_{i+1}) + f(e_j) \leq g_v(l_{i+1}) + f(e_{i+1}) = F^{i+1}(l_{i+1}) \\ F^{i+1}(x) &\leq F^i(l_{i+1}) \leq F^{i+1}(l_{i+1}). \end{aligned} \tag{8}$$

Meanwhile, the sum of $F(x)$ and $F(l_{i+1})$ is fixed:

$$F^i(l_{i+1}) + F^i(x) = g_v(l_{i+1}) + f(e_{i+1}) + g_v(x) + f(e_j) = F^{i+1}(x) + F^{i+1}(l_{i+1}). \tag{9}$$

Now we are able to estimate the change of entropy $H(L_G)$. Let $X$ be a discrete random variable of which the sample space $\Omega = \{a, b\}$. If $p(a) = x$, we have $H(X) = h(x) = x \log x + (1-x)\log(1-x)$. We plot the functional image in Figure 18 to show the property of $h(x)$. As is shown in the figure, if the sum of $p(a)$ and $p(b)$ is fixed, then the greater the difference between $p(a)$ and $p(b)$ is, the lower the entropy will be.

Note that $p(x) = \frac{F(x)}{\sum_{i=1}^{D} F(i)}$ and $\sum_{i=1}^{D} F(i)$ is a constant, so we can replace $F$ with $p$ in Inequation (8) and Equation (9) to get Inequation (10) and Equation (11) and then infer Inequation (12) and Inequation (13), as shown in Figure 19:

$$p^{i+1}(x) \leq p^i(l_{i+1}) \leq p^{i+1}(l_{i+1}) \tag{10}$$

$$p^i(l_{i+1}) + p^i(x) = p^{i+1}(x) + p^{i+1}(l_{i+1}) \tag{11}$$

$$|p^i(l_{i+1}) - p^i(x)| \leq |p^{i+1}(l_{i+1}) - p^{i+1}(x)| \tag{12}$$

$$p^i(x)\log p^i(x) + p^i(l_{i+1})\log p^i(l_{i+1}) \geq p^{i+1}(x)\log p^{i+1}(x) + p^{i+1}(l_{i+1})\log p^{i+1}(l_{i+1}). \tag{13}$$

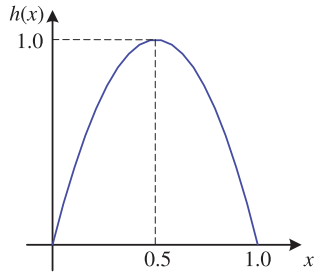



Fig. 18. Image of $h(x)$.

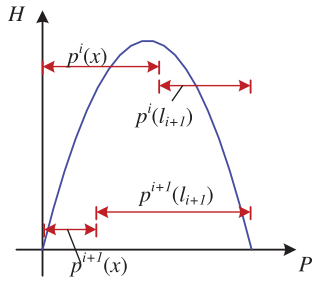



Fig. 19. Change of $p(l_{i+1})$ and $p(x)$.

The entropy of the label of the graph is defined as

$$H(L_G^i) = -\sum_{j=1}^{d} p^i(j) \log p^i(j). \tag{14}$$

Since only terms $p(x)\log p(x)$ and $p(l_{i+1})\log p(l_{i+1})$ in Equation (14) change, we can infer that $H(L_G^i) \geq H(L_G^{i+1})$. Consequently, we have

$$H(L'_G) = H(L_G^0) \geq H(L_G^1) \geq \cdots \geq H(L_G^k) = H(L_G^N). \tag{15}$$

To sum up the first part, we prove that any adjustable label of a vertex $L'_v$ can be adjusted to a nonadjustable label of the vertex $L_v^N$ and the entropy of $L_G^N$ does not exceed the entropy of $L'_G$.

The second part of our proof will exclude the equal case in Inequation (15). According to Inequation (15), if $H(L'_G) = H(L_G^N)$, we can infer that

$$H(L'_G) = H(L_G^0) = H(L_G^1) = \cdots = H(L_G^k) = H(L_G^N).$$

In addition, at any step in permutations, there must be

$$p^i(x)\log p^i(x) + p^i(l_{i+1})\log p^i(l_{i+1}) = p^{i+1}(x)\log p^{i+1}(x) + p^{i+1}(l_{i+1})\log p^{i+1}(l_{i+1}),$$
$$|p^i(l_{i+1}) - p^i(x)| = |p^{i+1}(l_{i+1}) - p^{i+1}(x)|.$$

Replacing $p$ with $F$, we get

$$|F^i(l_{i+1}) - F^i(x)| = |F^{i+1}(l_{i+1}) - F^{i+1}(x)|,$$
$$F^{i+1}(x) = F^i(l_{i+1}) \; or \; F^i(l_{i+1}) = F^{i+1}(l_{i+1}).$$

Then, we can infer that

$$g_v(l_{i+1}) = g_v(x) \; or \; f(e_{i+1}) = f(e_j).$$

Next, we use mathematical induction to prove $\sigma'_v$ is nonadjustable. We assume that after some swaps from $\sigma_v^N = \sigma_v^k$, we get

$$\sigma_v^{i+1} = \begin{pmatrix} e_1 & e_2 & \cdots & e_i & e_{i+1} & \cdots & e_j & \cdots \\ l_1 & l_2 & \cdots & l_i & l_{i+1} & \cdots & x & \cdots \end{pmatrix}$$

and it is nonadjustable:

$$g_v(l_1) \geq g_v(l_2) \geq \cdots \geq g_v(l_i) \geq g_v(l_{i+1}) \geq \cdots \geq g_v(x) \cdots,$$
$$f(e_1) \geq f(e_2) \geq \cdots \geq f(e_i) \geq f(e_{i+1}) \geq \cdots \geq f(e_j) \cdots.$$

Note that the position of the mapping pair $(e_j, x)$ in the permutations may differ from that in the first part of the proof because of the previous swaps. However, it still appears after the pair $(e_{i+1}, l_{i+1})$. If $g_v(l_{i+1}) = g_v(x)$, we swap $x$ and $l_{i+1}$ and get

$$\begin{pmatrix} e_1 & e_2 & \cdots & e_i & e_{i+1} & \cdots & e_j & \cdots \\ l_1 & l_2 & \cdots & l_i & x & \cdots & l_{i+1} & \cdots \end{pmatrix} = \sigma_v^i,$$

which satisfies

$$\begin{aligned} g_v(l_1) \geq g_v(l_2) \geq \cdots \geq g_v(l_i) \geq g_v(x) = \cdots = g_v(l_{i+1}) \cdots, \\ f(e_1) \geq f(e_2) \geq \cdots \geq f(e_i) \geq f(e_{i+1}) \geq \cdots \geq f(e_j) \cdots. \end{aligned}$$

In similar manner, if $f(e_{i+1}) = f(e_j)$, we swap $e_j$ and $e_{i+1}$ and get

$$\begin{pmatrix} e_1 & e_2 & \cdots & e_i & e_j & \cdots & e_{i+1} & \cdots \\ l_1 & l_2 & \cdots & l_i & l_{i+1} & \cdots & x & \cdots \end{pmatrix} = \begin{pmatrix} e_1 & e_2 & \cdots & e_i & e_{i+1} & \cdots & e_j & \cdots \\ l_1 & l_2 & \cdots & l_i & x & \cdots & l_{i+1} & \cdots \end{pmatrix} = \sigma_v^i,$$

which satisfies

$$\begin{aligned} g_v(l_1) \geq g_v(l_2) \geq \cdots \geq g_v(l_i) \geq g_v(l_{i+1}) \geq \cdots \geq g_v(x) \cdots, \\ f(e_1) \geq f(e_2) \geq \cdots \geq f(e_i) \geq f(e_{i+1}) = \cdots = f(e_j) \cdots. \end{aligned}$$

We keep doing this and finally have

$$\begin{pmatrix} e_{n_1} & e_{n_2} \cdots & e_{n_d} \\ l_{m_1} & l_{m_2} \cdots & l_{m_d} \end{pmatrix} = \begin{pmatrix} e_1 & e_2 & \cdots & e_d \\ m_1 & m_2 & \cdots & m_d \end{pmatrix} = \sigma_v^0 = \sigma_v',$$

which is nonadjustable:

$$\begin{aligned} g_v(l_{m_1}) \geq g_v(l_{m_2}) \geq \cdots \geq g_v(l_{m_d}), \\ f(e_{n_1}) \geq f(e_{n_2}) \geq \cdots \geq f(e_{n_d}). \end{aligned}$$

Consequently, if there is a vertex whose label is adjustable, we can adjust it and strictly decrease the entropy of the label of the whole graph.

In the third part, we prove that if there is no adjustable $L_v$ in $L_G$, $L_G$ is equivalent to the label $L_G^M$ constructed by MEL. Let $L_G$ be a label of the graph with frequency of labels sorted as

$$F(l_{m_1}) \geq F(l_{m_2}) \geq \cdots \geq F(l_{m_D}).$$

Given a nonadjustable label $L_v$,

$$\sigma_v = \begin{pmatrix} e_1 & e_2 \cdots & e_d \\ l_1 & l_2 \cdots & l_d \end{pmatrix},$$

and since $g_v(l) + f(e) = F(l)$, it is clear that

$$g_v(l_{m_1}) + f(e_{m_1}) \geq g_v(l_{m_2}) + f(e_{m_2}) \geq \cdots \geq g_v(l_{m_d}) + f(e_{m_d}).$$

If $g_v(l_{m_x}) + f(e_{m_x}) \geq g_v(l_{m_y}) + f(e_{m_y})$ and $g_v(l_{m_x}) < g_v(l_{m_y})$, we can infer that $f(e_{m_x}) \geq f(e_{m_y})$. On the other hand, if $g_v(l_{m_x}) + f(e_{m_x}) \geq g_v(l_{m_y}) + f(e_{m_y})$ but $g_v(l_{m_x}) \geq g_v(l_{m_y})$, because $L_v$ is nonadjustable, we can also infer that $f(e_{m_x}) \geq f(e_{m_y})$. Therefore, in $L_v$, there must be

$$\begin{aligned} g_v(l_{m_1}) \geq g_v(l_{m_2}) \geq \cdots \geq g_v(l_{m_d}), \\ f(e_{m_1}) \geq f(e_{m_2}) \geq \cdots \geq f(e_{m_d}). \end{aligned}$$

In other words, in each $L_v$, we give $l_{m_i}$ to the edge with the $i^{th}$ greatest frequency. Because labels of different orders in $G$ are essentially equivalent, we infer that the nonadjustable label $L_G$ is equivalent to $L_G^M$.

To draw a conclusion, given any adjustable label $L_G$, we are able to adjust it and decrease the entropy strictly until it is equivalent to $L_G^M$. Consequently, MEL constructs a label with the lowest entropy. □

Because of the optimality of MEL and Shannon's source coding theorem, the coding algorithm based on Huffman coding will achieve the minimal average code length. The coding algorithm builds a Huffman tree on labels, so the time complexity of initialization is $O(D \log D)$. Both compression and decompression have a time complexity of $O(|T|)$, which is exactly the time complexity of Huffman coding. Different from HDC, the L&C algorithm incurs a much smaller storage overhead to store a Huffman tree and a label of the graph, and hence it is ideal for archiving or transferring data. Our L&C algorithm doesn't take advantage of FSPs, but it involves no major principle, and we will explain this problem in our experiments.

## 5. TEMPORAL COMPRESSION

Temporal sequences are the other part of decomposed trajectories. In this section, we introduce a lossy compression algorithm to compress temporal sequences. First, we present some preliminary definitions. Next, we define the error metrics we use to bound the potential difference between the information captured by the real temporal sequence and that captured by the compressed temporal sequence. Then, we present the error-bounded temporal compression algorithm.

### 5.1. Preliminary

A temporal sequence is in the form of $((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$, where $d_i$ refers to the distance the moving object has traveled at the timestamp $t_i$. If we strictly follow this statement, $d_1$ corresponding to $t_1$ will be always zero as $t_1$ is the timestamp when the object is about to start the journey. However, we want to highlight that $d_1$ is not always zero as we use $d_1$ to indicate the distance from an object's initial position to the starting point of the first edge $e_1$ in the corresponding spatial path. This arrangement is to guarantee that even the spatial path only captures the edges that a moving object passes sequentially within one journey; our new representation is still very general and flexible, which allows an object to start its journey from any point in the road network, not necessarily from a node of the road network. Take trajectory $T_2$ depicted in Figure 4 as an example. The object is initially located on edge $e_8$, having distance $\Delta_{21}$ from the start node of $e_8$. That explains why $d_1 = \Delta_{21}$. To be more precise, $d_i$ in our temporal sequence refers to the distance an object has traveled so far, plus $d_1$. In the cases when $d_1 = 0$, $d_i$ refers to the real travel distance. Otherwise (i.e., $d_1 > 0$), it refers to the summation of travel distance and $d_1$.

As mentioned before, the number of $(d_i, t_i)$ tuples in a temporal sequence $\mathcal{TS}_T$ is the same as the number of $\langle x_i, y_i, t_i \rangle$ tuples in the original GPS trajectory $T$ (i.e., $|\mathcal{TS}_T| = |T|$) and we can transform easily between $\mathcal{TS}_T$ and $T$ based on the corresponding spatial path $\mathcal{SP}_T$. However, both the GPS trajectory $T$ and the temporal sequence $\mathcal{TS}_T$ only capture certain information of a real trajectory that is *incomplete*. More precisely, the sampled trajectory only records accurate positions of the moving object at some specific timestamps and the trajectory between two continuous timestamps is unknown. As a solution, we make an assumption that the moving object moves *uniformly* in a short period (e.g., from $t_i$ to $t_{i+1}$) and the trajectory between two timestamps is approximated by linear interpolation. Formally, given a timestamp $t_x \in [t_i, t_{i+1})$, its corresponding $d_x$ can be calculated as $d_i + \frac{(t_x - t_i) \times (d_{i+1} - d_i)}{t_{i+1} - t_i}$, which means we can define distance as a function of time as stated in Definition 5.1.

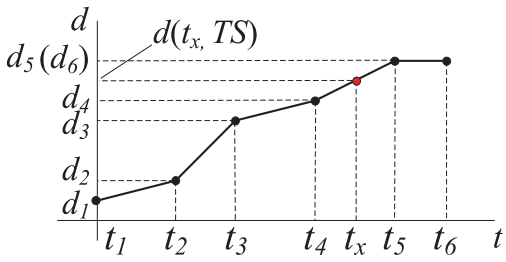



Fig. 20. Sample of a time-distance graph.

*Definition* 5.1 (*Distance Function*). The distance function $d(t_x, \mathcal{TS}_T)$ corresponding to a temporal sequence $\mathcal{TS}_T$ is a piecewise linear function, which is defined as

$$d(t_x, \mathcal{TS}_T) = \begin{cases} d_i + \dfrac{(t_x - t_i) \times (d_{i+1} - d_i)}{t_{i+1} - t_i} & (t_i < t_x < t_{i+1}) \\ d_i & (t_x = t_i), \end{cases}$$

where $(d_i, t_i)$s and $(1 \le i \le n)$ are tuples in the temporal sequence $\mathcal{TS}_T$.

To visualize the change of distance over time in a temporal sequence, we can plot all sampled tuples $(d_i, t_i)$ in the time-distance graph and join contiguous tuples $(d_i, t_i)$ and $(d_{i+1}, t_{i+1})$ by line segments, as depicted in Figure 20. Distance function $d(t_x, \mathcal{TS}_T)$ actually returns the corresponding position along distance dimension for a given timestamp.

Similarly, we can also introduce the function $t(d_x, \mathcal{TS}_T)$, which returns all the timestamps when the distance traveled is equivalent to a given input $d_x$, as formally defined in Definition 5.2. As the distance is nondecreasing and an object might remain in a position for some time (e.g., cars stuck in a traffic jam, buses waiting in front of a traffic light), the return of $t(d_x, \mathcal{TS}_T)$ might be a duration rather than a single timestamp. Back to the time-distance graph depicted in Figure 20, $t(d_5, \mathcal{TS}_T)$ will return all the timestamps within the duration of $[t_5, t_6]$. Both function $d(t_x, \mathcal{TS}_T)$ and function $t(d_x, \mathcal{TS}_T)$ will be useful for applications that need to query trajectory sets, such as locating all the moving objects that pass a position $p$ between 16:50 and 17:00 on April 1, 2014. Please refer to Section 6 for the set of essential queries that can be supported by COMPRESS.

*Definition* 5.2 (*Time Function*). The time function $t(d_x, \mathcal{TS}_T)$ corresponding to a temporal sequence $\mathcal{TS}_T$ is a function that returns a set of time values $t_x$ such that $t(d_x, \mathcal{TS}_T) = d_x$, that is, $t(d_x, \mathcal{TS}_T) = \{t_x \in [t_0, t_n] | d(t_x, \mathcal{TS}_T) = d_x\}$.

### 5.2. Error Metrics

Since the compression algorithm for temporal sequences is lossy, we must bound the inaccuracy that could be caused during the compression. Many existing works [Cao and Wolfson 2005; Cao et al. 2006; Kellaris et al. 2013; Muckell et al. 2013] use the metric *Time Synchronized Euclidean Distance* (TSED) to bound the error between the original trajectory and the compressed trajectory. Given two trajectories $T$ and $T'$, TSED returns the maximum Euclidean distance between point $\langle x_i, y_i, t_i \rangle$ in the original trajectory $T$ and point $\langle x'_j, y'_j, t'_j \rangle$ with $t_i = t'_j$ in another trajectory $T'$. Different from existing works, we propose two new error metrics, namely, *Time Synchronized Network Distance* (TSND) and *Network Synchronized Time Difference* (NSTD), as formally defined in Definition 5.3 and Definition 5.4, respectively.[3]

[3]Definition 5.1 and Definition 5.3 are the same as the ones defined in Song et al. [2014], and we still present them here for readability. However, Definition 5.2 and Definition 5.4 are different from the ones in Song

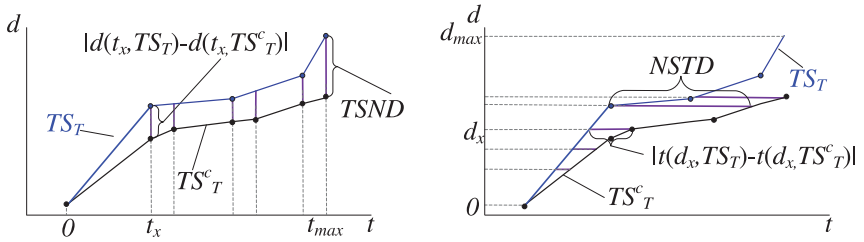



Fig. 21. Sample TSND and NSTD.

*Definition* 5.3 (*Time Synchronized Network Distance (TSND)*). Given a temporal sequence $\mathcal{TS}_T$ and its compressed one $\mathcal{TS}^c_T$, TSND measures the maximum difference between the distance that the moving object travels via $\mathcal{TS}_T$ and $\mathcal{TS}^c_T$ at any same timestamp:

$$TSND = max_{t_x \in [0, t_{max}]}(|d(t_x, \mathcal{TS}_T) - d(t_x, \mathcal{TS}^c_T)|). \tag{16}$$

*Definition* 5.4 (*Network Synchronized Time Difference (NSTD)*). Given a temporal sequence $\mathcal{TS}_T$ and its compressed one $\mathcal{TS}^c_T$, NSTD measures the maximum difference between the time that the moving object travels via $\mathcal{TS}_T$ and $\mathcal{TS}^c_T$ through the same distance:

$$NSTD = max_{d_x \in [0, d_{max}]}(|t^{min}(d_x, \mathcal{TS}_T) - t^{min}(d_x, \mathcal{TS}^c_T)|, |t^{max}(d_x, \mathcal{TS}_T) - t^{max}(d_x, \mathcal{TS}^c_T)|), \tag{17}$$

where $t^{min}(d_x, \mathcal{TS}_T)$ returns the minimum value of time in $t(d_x, \mathcal{TS}_T)$, and $t^{max}(d_x, \mathcal{TS}_T)$ returns the maximum value of time in $t(d_x, \mathcal{TS}_T)$.

To be more specific, TSND measures the maximum difference between two temporal sequences along the time dimension, and NSTD measures the maximum difference between two temporal sequences along the distance dimension, as visualized in Figure 21. As a temporal sequence is a polyline in time-distance dimensions, the maximum distance between two temporal sequences along time or distance dimension only appears at vertices of polylines. In other words, we can derive TSND and NSTD in $O(|\mathcal{TS}_T|)$ time.

In most cases, function $t(d_x, \mathcal{TS}_T)$ only returns a single value and Equation (17) could be simplified as $NSTD = max_{d_x \in [0, d_{max}]}(|t(d_x, \mathcal{TS}_T) - t(d_x, \mathcal{TS}^c_T)|)$, which is the definition we presented in Song et al. [2014]. However, for an object that does not move within a certain period of time, $t(d_x, \mathcal{TS}_T)$ actually returns a period instead of a single timestamp; for example, $t(d_5, \mathcal{TS}_T)$ for the temporal sequence depicted in Figure 20 returns a duration $[t_5, t_6]$. For a given $d_x$, if $t(d_x, \mathcal{TS}_T)$ and/or $t(d_x, \mathcal{TS}^c_T)$ return a duration, we separate the comparison of the distance at the start time from that of the distance at the end time, as visualized in Figure 22. In brief, Definition 5.4 and Definition 5.2 are more rigorous than the ones defined in Song et al. [2014].

We want to highlight that, given two temporal sequences, their TSND and NSTD are meaningful only when those two temporal sequences are associated with the same spatial path, such as $\mathcal{TS}_T$ and $\mathcal{TS}^c_T$. Given a spatial path, it is easy to prove that TSND between two temporal sequences is always an upper bound of TSED between their GPS trajectories, as proved in Song et al. [2014]. That means if the error measured by TSND is less than $\tau$, then the error measured by TSED must be less than $\tau$. On the other hand, NSTD is as important as TSND because they measure the errors

et al. [2014] since we have revised the definition to deal with the case that several timestamps share the same distance.

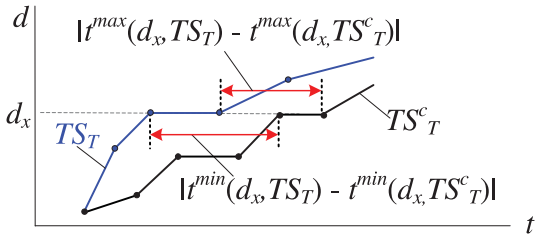



Fig. 22. Special case of NSTD.

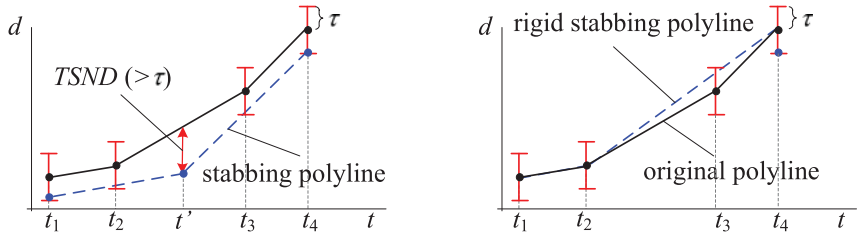



Fig. 23. Stabbing polyline and rigid stabbing polyline.

caused by compression in different dimensions. A small NSTD does not guarantee a small TSND, and vice versa. Later in Section 6, we will introduce several queries that can be efficiently processed on compressed data, and the precision of query results is also bounded by TSND and NSTD. Consequently, we need to consider both TSND and NSTD when designing our temporal sequence compression algorithms to guarantee the query precision on the compressed trajectory. Both TSND and NSTD are special cases of Fréchet distance, which measures the similarity between two curves, and Fréchet distance in road networks is introduced in Fan et al. [2010].

### 5.3. Bounded Temporal Compression

Having introduced metrics TSND and NSTD, we are now ready to present the lossy compression algorithm for temporal sequence compression, with the following two main targets. First, both TSND and NSTD between the compressed temporal sequence and the original temporal sequence must be bounded by the tolerant values specified by users/applications. Second, the compression ratio should be as high as possible.

*5.3.1. Rigid Stabbing Polyline Compression.* When we plot the original temporal sequence on the time-distance graph, it is exactly a polyline stabbing all points from $(d_1, t_1)$ to $(d_n, t_n)$. For a given TSND value $\tau$, we can draw a series of $n$ vertical line segments with length of $2\tau$ centered at each point, as depicted in Figure 23. If we consider those $n$ vertical line segments as objects, we can reduce the temporal compression problem to a computational geometry problem, namely, *stabbing polyline problem* [Guibas et al. 1991] in the time-distance plane, which tries to search for a polyline containing a minimum number of vertices passing through several objects in order. By reducing the number of vertices used to represent the polyline, the stabbing polyline problem can help to save the storage overhead. As shown in Figure 23, the original temporal sequence contains four points, while its stabbing polyline contains only three points with a 1.33 compression ratio. If we consider both TSND and NSTD, we formulate the stabbing polyline problem in our context in Definition 5.5.

*Definition* 5.5 (*Stabbing Polyline*). A stabbing polyline is a polyline that intersects with all $n$ vertical line segments and $n$ horizontal line segments centered at $n$ vertices of the original polyline on the time-distance graph.

However, we want to highlight that the stabbing polyline defined in Definition 5.5 can only guarantee the maximum difference between the original polyline and a stabbing polyline along a time dimension at *original sampled timestamps* (e.g., $t_1$, $t_2$, $t_3$, and $t_4$ in Figure 23) bounded by a given TSND (i.e., $\tau$ in the graph), but the real TSND might be larger than the specified $\tau$. For example, as shown in Figure 23, the difference between the original polyline and the stabbing polyline at time $t'$ is larger than $\tau$. If we choose the resulting vertices from the vertices of original polyline, the time and distance differences at *all timestamps* (e.g., $[t_1, t_4]$ in Figure 23) are guaranteed. In order to differentiate the polyline we look for from normal polylines, we introduce the concept of the *rigid stabbing polyline* in Definition 5.6.

*Definition* 5.6 (*Rigid Stabbing Polyline*). Given a temporal sequence $\mathcal{TS}_T$ represented by a polyline of $n$ points in time-distance space and two error bounds $\tau$ and $\eta$, set $S$ refers to $n$ vertical line segments of length $2\tau$ and $n$ horizontal line segments of length $2\eta$, both centered at $n$ points of $\mathcal{TS}_T$. A stabbing polyline corresponding to $2n$ line segments of $S$ is a *rigid stabbing polyline* if all its vertices belong to the original $n$ points of $\mathcal{TS}_T$.

in Song et al. [2014], we proposed a greedy algorithm searching for a rigid stabbing polyline to compress a temporal sequence. Though the greedy algorithm only takes $O(n)$ time, it cannot generate an optimal rigid polyline containing the fewest vertices. The optimal rigid stabbing polyline can be computed via dynamic programming, which takes $O(n^2)$ time. Furthermore, the dynamic programming algorithm is not optimal, because the vertices of the optimal stabbing polyline may be new vertices instead of input vertices. In the following, we are going to introduce *Tube Stabbing polyLine Compression (TSLC)*, a linear algorithm that computes the optimal stabbing polyline. We want to highlight that RSLC is not used for temporal compression in COMPRESS, but rather for comparison purposes in experiments to illustrate the performance advantage of TSLC. In Section 7, we compare TSLC with the greedy RSLC to illustrate the efficiency of TSLC and compare TSLC with RSLC based on dynamic programming to show its effectiveness. More details of rigid stabbing polyline compression are discussed in the online appendix [Han et al. 2016].

*5.3.2. Tube Stabbing Polyline Compression.* As mentioned before, a rigid stabbing polyline is a variance of the original stabbing polyline, which applies the restriction *turn in objects* to the original stabbing polyline problem. The stabbing polyline with no restriction does not satisfy the requirements of TSND and NSTD, as "no restriction" is too weak to bound the error. On the other hand, "turn in objects" is such a strong restriction that the algorithm may miss the optimal stabbing polyline, as the optimal stabbing polyline that satisfies the requirements of TSND and NSTD and meanwhile contains the minimum number of vertices does not necessarily pass the original vertices (e.g., Figure 24). In the following, we introduce another variance of stabbing polyline, which proposes the restriction "*tune in tube*" that requires each vertex of the polyline to be in a region bounded by two consecutive objects and their outer common tangents.

Again, we plot the original temporal sequence in the form of a polyline in time-distance space. We also plot vertical line segments with length of $2\tau$ centered at each vertex. Then, by joining all the top endpoints of consecutive vertical line segments and joining all the bottom endpoints of consecutive vertical line segments, we can form a simple polygon, namely, TSND tube $P_d$, that is centered at the original polyline,

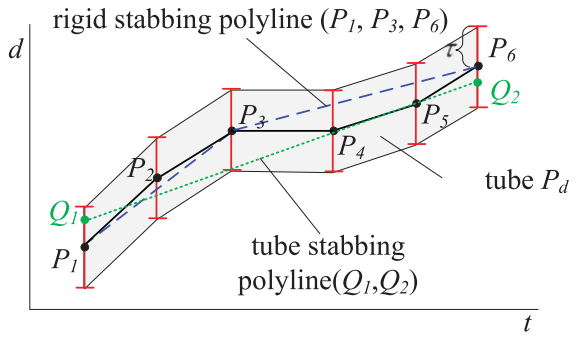



Fig. 24. Tube stabbing polyline and rigid stabbing polyline.

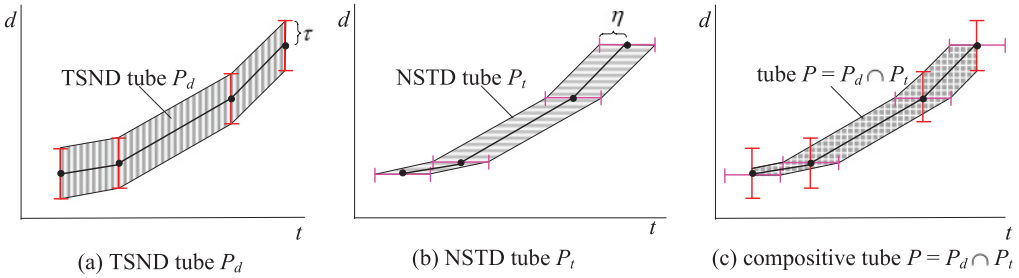



Fig. 25. Sample tube.

as formally defined in Definition 5.9 and visualized in Figure 25. Obviously, TSND between the tube stabbing polyline and the original polyline is bounded by $\tau$ if and only if the tube stabbing polyline falls *completely* inside the TSND tube $P_d$. In this example, we also observe that the rigid stabbing polyline contains three vertices, while the tube stabbing polyline only contains two vertices.

Given a tolerant time error $\eta$, we can repeat the previous process to obtain a simple polygon called NSTD tube $P_t$, as defined in Definition 5.8. Given a polyline with corresponding error bounds $\tau$ and $\eta$, we can form TSND tube $P_d$ and NSTD tube $P_t$, and then a compositive tube $P = P_t \cap P_d$. We name a stabbing polyline that falls completely within the tube $P$ *tube stabbing polyline*, as formally defined in Definition 5.10. It is confirmed that given a polyline and a corresponding tube stabbing polyline, their TSND and NSTD are bounded by $\tau$ and $\eta$ respectively.

*Definition* 5.7 (*TSND Tube*). Given a temporal sequence $((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$ and a tolerant TSND error $\tau$, the corresponding *TSND tube* $P_d$ is a polygon with $2n$ points, that is, $(d_1+\tau, t_1), (d_2+\tau, t_2), \ldots, (d_n+\tau, t_n), (d_n-\tau, t_n), (d_{n-1}-\tau, t_{n-1}), \ldots, (d_1-\tau, t_1)$.

*Definition* 5.8 (*NSTD Tube*). Given a temporal sequence $((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$ and a tolerant NSTD error $\eta$, the corresponding *NSTD tube* $P_t$ is a polygon with $2n$ points, that is, $(d_1, t_1 - \eta), (d_2, t_2 - \eta), \ldots, (d_n, t_n - \eta), (d_n, t_n + \eta), (d_{n-1}, t_{n-1} + \eta), \ldots, (d_1, t_1 + \eta)$.

*Definition* 5.9 (*Compositive Tube*). Given a temporal sequence $((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$, a tolerant TSND error $\tau$, and a tolerant NSTD error $\eta$, let $P_d$ be the corresponding TSND tube and $P_t$ be the corresponding NSTD tube. Then, their intersection forms the *compositive tube* $P$, that is, $P = P_d \cap P_t$.

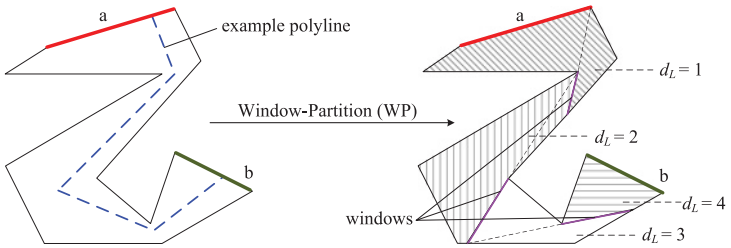



Fig. 26. Window-Partition (WP) algorithm.

*Definition* 5.10 (*Tube Stabbing Polyline*). Given a temporal sequence $((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$, and its corresponding compositive tube $P$, a *tube stabbing polyline* is a polyline completely falling within the compositive tube, which links $(d_1, t_1)$ to $(d_n, t_n)$.

Both the TSND tube and NSTD tube can be constructed within $O(n)$ time, where $n$ is the length of the temporal sequence $\mathcal{TS}$. Because both the TSND tube and NSTD tube are monotone polygons, it takes $O(n)$ time to compute the intersection by a sweep line algorithm, which is introduced in Chapter 2.4 of de Berg et al. [2000]. After constructing the compositive tube, we connect $(d_1, t_1)$ to $(d_n, t_n)$ with a polyline $P_{\mathcal{TS}}$ completely falling inside the tube, to compress the temporal sequence $\mathcal{TS}$. As the polyline contains the fewest vertices, the compressed temporal sequence is optimal. In the following, we explain how to locate the polyline $P_{\mathcal{TS}}$.

In the computational geometry literature, a distance metric called *link distance* refers to the minimum number of vertices of a polyline that connects two objects inside a region without crossing the boundary. *Minimum-link path problems* are problems of finding polylines with minimum link distance, which have been studied in Suri [1990], Mitchell et al. [1992], Esther M. Arkin and Suri [1995], and Chiang and Tamassia [1997]. Algorithm WP [Suri 1990] can find a polyline with minimum link distance between two edges of a simple polygon with $n'$ vertices in $O(n')$ time when the polygon is triangulated. Polygon triangulation is to decompose a polygon into a set of nonintersecting triangles, which can be done in $O(n')$ time [Bernard 1991], though the algorithm is very complex. In the following, we will prove that the compositive tube is a monotone polygon, which is easy to triangulate in linear time.

Algorithm WP partitions the simple polygon into several regions in which the points have a constant link distance from the source, and the partition lines between regions are called *windows*. Take Figure 26 as an example. Given a source edge $a$, the polygon is partitioned into four smaller polygons by three windows, within which each of the points shares a constant link distance $d_L$ from the source. The algorithm first computes a region called *visibility polygon* $P_V(a)$ containing all points inside the polygon that are visible from $a$. Each edge of the visibility polygon $P_V(a)$ is either on the boundary or a window. Then, we discard $P_V(a)$ and recursively compute visibility polygon $P_V(e)$ for those window edges $e$ of $P_V(a)$ until the destination edge $b$ is reached. The computation of $P_V(e)$ costs $O(k_e)$ time, where $k_e$ is the number of triangles intersected by $P_V(e)$, since we need to visit at most $k_e$ triangles to determine the visible region. Because a triangle in the polygon intersects at most three regions in the polygon while there are $O(n')$ triangles, as introduced in Chapter 3 of de Berg et al. [2000], the total time complexity is linear. The link distance between edge $a$ and edge $b$ is 4; that is, at least four edges are needed to connect $a$ and $b$, and an example polyline with four edges is plotted in the figure for illustration purposes. Algorithm WP can also find the minimum-link path between vertices by regarding them as edges (with two overlapping vertices each).

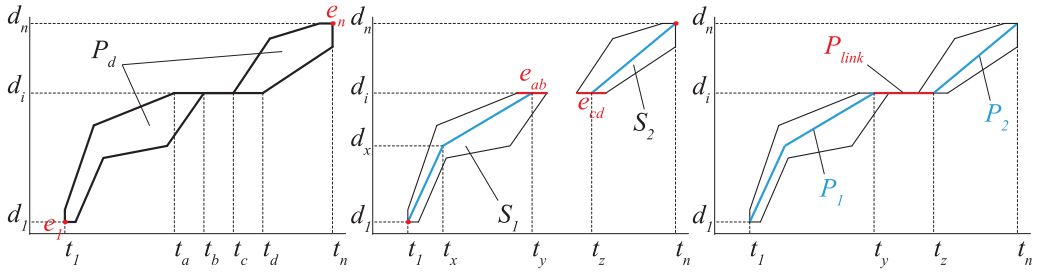



Fig. 27. Division of compositive tube.

Though the minimum-link path problem in a simple polygon has been solved, the algorithm WP cannot directly compute the minimum tube stabbing polyline because there may be two edges intersecting with each other in the compositive tube, when some timestamps share the same value of distance (e.g., $(t_i, d_i)$ and $(t_{i+1}, d_{i+1})$ with $d_i = d_{i+1}$) or one of the tolerant errors $\eta$ and $\tau$ is zero. The following theorem proves that the two cases mentioned earlier are the *only* scenarios in which edge overlapping actually occurs.

THEOREM 5.11. *The compositive tube is a simple polygon—moreover, a strictly monotone polygon—if both tolerant errors are not zero and distance is a strictly increasing function of time.*

PROOF. Please refer to the online appendix [Han et al. 2016]. □

It is easy to deal with the case that $\tau = 0$ or $\eta = 0$ since the stabbing polyline and the original one have exactly the same shape. In the following, we only focus on the case where $\tau \cdot \eta > 0$ and the edges' overlapping is the result of the same values of distance at some timestamps, and thus the overlapping edges are horizontal line segments. As shown in Figure 27, a three-stage algorithm can be performed to construct a minimum stabbing polyline in a compositive tube. First, a compositive tube $P_d$ is divided into several simple polygons according to the overlapping edges. Take the tube in the figure as an example; $\langle (d_i, t_a), (d_i, t_c) \rangle$ and $\langle (d_i, t_b), (d_i, t_d) \rangle$ are two overlapping edges of $P_d$, and $P_d$ is divided into $S_1$ and $S_2$. Second, we compute the minimum-link path in each simple polygon. In Figure 27, the path in $S_1$ links $e_1 = (d_1, t_1)$ to $e_{ab} = \langle (d_i, t_a), (d_i, t_b) \rangle$, while the path in $S_2$ links $e_{cd} = \langle (d_i, t_c), (d_i, t_d) \rangle$ to $e_n = (d_n, t_n)$. Finally, the horizontal line $\langle (d_i, t_a), (d_i, t_d) \rangle$ forces the path to go through it, so $(d_i, t_x)$ is linked to $(d_i, t_z)$ and $\langle (d_1, t_1), (d_x, t_x), (d_i, t_y), (d_i, t_z), (d_n, t_n) \rangle$ forms the minimum stabbing polyline.

It is easy to triangulate monotone polygons in $O(n')$ time, so the compositive tube is favorable for the algorithm WP to process. To draw a conclusion, TSLC compresses the temporal sequence in $O(n)$ time and generates the minimum stabbing polyline, that is, a compressed temporal sequence with the smallest size. However, algorithm WP is much more complicated than greedy algorithms. We will compare the efficiency and effectiveness of RSLC and TSLC in our experimental study.

## 6. APPLICATIONS ON COMPRESSED TRAJECTORIES

The main purpose of trajectory compression is to reduce the storage overhead of trajectory data. Consequently, whether the compressed trajectories can support efficient queries is not our main focus. However, it is still desirable that some queries can be processed on trajectories that are not fully decompressed. The effectivity of queries on compressed trajectories is ensured by error-boundness of COMPRESS. More precisely, given the tolerant error of TSND $\tau$ and that of NSTD $\eta$, the error of every point in

Table III. Four Query Processing Approaches

| | Original | Indexed | Full | Partial |
|---|---|---|---|---|
| Space cost | High | Highest | Low | Low |
| Time cost | High | Lowest | Highest | Low |

the decompressed trajectory is less than $\tau$ and $\eta$, so the time and distance errors of query results are bounded too. There are mainly four query processing approaches based on different data forms, and they are different in space and time performance, as summarized in Table III.

—*Query processing on original data, denoted as Original*. This naive approach is simply searching in original data to process queries. It is ordinary and does not incur any additional space or time cost. We consider the naive approach as a benchmark when evaluating the performance of other approaches.
—*Query processing on original data with indexes, denoted as Indexed*. The conventional method of query processing is creating indexes (e.g., a B-tree [Bayer and McCreight 1972]) on the original trajectories to speed up the query processing. However, this approach stores both original data and additional indexes so it runs counter to the purpose of reducing the storage space.
—*Query processing on fully decompressed data, denoted as Full*. This approach is to store all the trajectories in their compressed form and to process queries on the fully decompressed trajectories. It saves space but costs more processing time.
—*Query processing on partially decompressed data, denoted as Partial*. The last approach stores compressed trajectories so the storage space is reduced. When processing queries, we take the dictionary generated by the compression algorithm as an index and partially decompress the trajectory data. The approach saves space and costs less time than that on fully decompressed data.

Because query processing on original data, indexed data, and fully decompressed data is straightforward, we skip the explanation of these algorithms but report their performance via experimental studies in Section 7. In this section, we introduce two basic queries, position query $where(T, t_i)$ and time query $when(T, x_i, y_i)$, that are building blocks of many common queries used by LBS applications [Cao et al. 2006]. Afterward, we discuss *data utility* and how it affects the performance of compression and query algorithms.

### 6.1. Position Query

The position query $where(T, t_i)$ returns the location $(x_i, y_i)$ of the moving object at a given timestamp $t_i \in [t_1, t_n]$ along the trajectory $T$. Given an original trajectory $T$ that is represented by its spatial path $\mathcal{SP}_T = (e_1, e_2, \ldots, e_m)$ and temporal sequence $\mathcal{TS}_T = ((d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n))$, the query $where(T, t_i)$ on original data first searches $\mathcal{TS}_T$ for $d_i$ corresponding to $t_i$ based on distance function $d(t_i, \mathcal{TS}_T)$, and then searches $\mathcal{SP}_T$ to locate $(x_i, y_i)$ on a certain edge $e_j$ based on travel distance $d_i$. Consequently, query $where(T, t_i)$ on the original trajectory on average scans $\frac{m}{2}$ edges in the spatial path and $\frac{n}{2}$ tuples in the temporal sequence.

Now we explain how to perform query $where(T, t_i)$ on compressed trajectory $T^c$. It is still a two-stage process, first locating $d_i'$ in $\mathcal{TS}_T^c$ and then locating $(x_i', y_i')$ in $\mathcal{SP}_T^c$. As compressed temporal sequence $\mathcal{TS}_T^c$ shares the same format as original temporal sequence $\mathcal{TS}_T$, the first stage also scans $\frac{|\mathcal{TS}_T^c|}{2}$ tuples in temporal sequence on average. Here, $|\mathcal{TS}_T^c|$ refers to the total number of tuples in $\mathcal{TS}_T^c$. If we assume our temporal

compression algorithm can reach a compression ratio of $\beta$ (i.e., $\frac{|\mathcal{TS}_T|}{|\mathcal{TS}_T^c|} = \beta$), the first stage scans $\frac{n}{2\beta}$ tuples of the compressed temporal sequence on average.

As for the second-stage search on compressed spatial paths, it will not be supported automatically as our compressed spatial paths are in a very different format from their original form. Recall that our spatial compression HDC invokes SPC and PLZW[4] to perform compression, with the help of two data structures $Prev(e_i, e_j)$ and Trie. In the following, we explain how to embed distance information in these two data structures in order to support the search on compressed spatial paths based on a given $d_i$.

Given a compressed spatial path $\mathcal{TS}_T^c$ that is in the format of binary code, we first need to decompress $\mathcal{TS}_T^c$ into frequent paths captured by the trie. To avoid fully decompressing $\mathcal{TS}_T^c$, we store additional distance information, denoted as $dist(n_i)$, corresponding to each node $n_i$ in the trie. To be more specific, $dist(n_i)$ captures the shortest distance from the edge captured by the root node to the edge captured by node $n_i$, bypassing all the edges captured by the nodes in between node $n_i$ and the root node. In other words, let $n_i'$ be the parent node of node $n_i$ in the trie; $dist(n_i)$ is the summation of $dist(n_i')$ and the shortest distance from the edge corresponding to node $n_i$ and the edge corresponding to node $n_i'$, that is, $dist(n_i) = dist(n_i') + |SP(edge(n_i), edge(n_i'))|$.[5] Take the trie depicted in Figure 9 as an example. $dist(20)$ will be set to the shortest distance from $e_2$ to $e_3$ and $dist(25)$ will be set to the shortest distance from $e_5$ to $e_2$ and then to $e_3$. Then, we can start the decompression (w.r.t. PLZW) process, which is incremental. Whenever we recover a symbol corresponding to a node $n$ from $\mathcal{TS}_T^c$, we check whether the accumulative distance $\sum dist(n_i)$ of all the recovered symbols has already exceeded $d_i$. Assume we recover nodes $n_1, n_2, \ldots, n_x$, with $\sum_{1 \le j < x} dist(n_j) < d_i$ and $\sum_{1 \le j \le x} dist(n_j) \ge d_i$; we are certain that the travel distance $d_i$ must correspond to some point along the edges captured by FSP $n_x$.

Given one FSP corresponding to a trie node $n_x$, we can recover the edge sequence (i.e., the sequence of edges corresponding to the path from the root node to node $n_x$). However, we want to highlight that our PLZW takes the output of SPC as inputs; for example, two consecutive edges corresponding to one FSP might not be adjacent to each other. We need to recover the shortest path corresponding to each pair of consecutive edges corresponding to $n_x$, with the help of $Prev(e_i, e_j)$. Consequently, we need to store the distance information corresponding to $Prev(e_i, e_j)$ as well. Again, the decompression w.r.t. SPC is also incremental and it can be terminated when the accumulative travel distance (i.e., $\sum_{1 \le j \le x} dist(n_j)$ and the total distance of recovered shortest paths corresponding to $n_x$) exceeds $d_i$.

Suppose that the compression ratio of HDC is $\alpha = \alpha_1 \alpha_2$, where $\alpha_1$ and $\alpha_2$ are the compression ratio of SPC and PLZW, respectively; $K$ is the total number of nodes in the trie; and $|E|$ is the number of edges in the road network. As shown in Figure 28, the final length $L_2$ of the compressed trajectory is

$$L_2 = x \cdot \log_2 K = \frac{m}{\alpha} \cdot \log_2 |E|,$$

so the average number $x$ of FSPs in a spatial path that needs to be decompressed can be estimated as

$$x = \frac{\frac{m}{\alpha_1} \cdot \log_2 |E|}{\alpha_2 \cdot \log_2 K} = \frac{cm}{\alpha},$$

[4]HDC without SPC can support the second-stage search in a similar way, which has been skipped for space saving.

[5]$edge(n_i)$ refers to the edge captured by the trie node $n_i$.

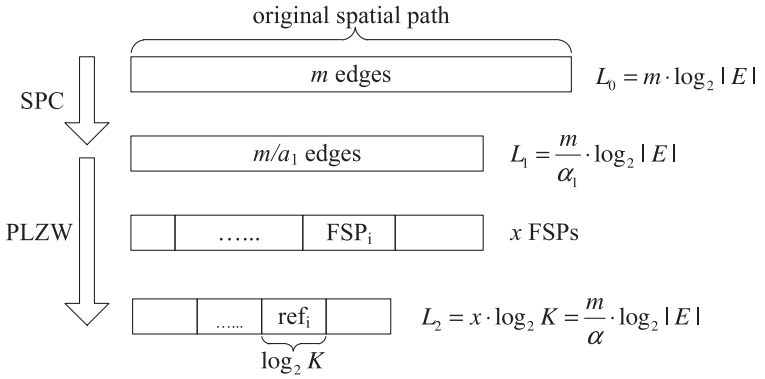



Fig. 28. Average number of FSPs.

with $c = \log_K |E|$. Similarly, the average number of edges that a FSP contains is calculated as

$$y = \frac{m}{\alpha_1 \cdot x} = \frac{\alpha_2}{c}.$$

Consequently, $where(T, t_0)$ visits $\frac{cm}{2\alpha}$ trie nodes, $\frac{\alpha_2}{2c}$ edges, and $\frac{n}{2\beta}$ tuples of a compressed temporal sequence on average. In our experiments, we have $|E| < K < |E|^2$, so $c$ is a constant with $0.5 < c < 1$. Given the fact that $\alpha_1 > 1$, $\alpha_2 > 1$, $\beta > 1$, and $0.5 < c < 1$, the query time is reduced as $\frac{cm}{2\alpha} < \frac{m}{2}$, $\frac{n}{2\beta} < \frac{n}{2}$, and $\frac{\alpha_2}{2c}$ is a constant.

### 6.2. Time Query

The time query $when(T, x_i, y_i)$ returns the timestamp $t_i$ at which the moving object is at the position $(x_i, y_i)$. Given an original trajectory $T$, $when(T, x_i, y_i)$ needs to first locate $(x_i, y_i)$ to an edge in the spatial path $\mathcal{SP}_T$, and then to compute the distance $d_i$ that the moving object travels along $T$ until the input point. Then, it searches the temporal sequence $\mathcal{TS}_T$ for the timestamp $t_i$ to fit the distance $d_i$, based on time function $t(d_i, \mathcal{TS}_T)$. In total, it scans $\frac{m}{2}$ edges in $\mathcal{SP}_T$ and $\frac{n}{2}$ tuples in $\mathcal{TS}_T$ on average.

To avoid fully decompressing the compressed trajectory to support time query, we again need to embed certain information in the two main data structures, that is, Trie and $Prev(e_j, e_l)$. For each trie node $n_i$, we keep the Minimum Bounding Rectangle (MBR), denoted as $MBR(n_i)$, that bounds the sequence of edges corresponding to $n_i$. For each entry $Prev(e_j, e_l)$, we also keep MBR $MBR(e_j, e_l)$ that bounds the shortest path from $e_j$ to $e_l$.

While scanning the references in the compressed trajectory, $when(T, x_i, y_i)$ first checks whether point $(x_i, y_i)$ is inside $MBR(n_j)$. If the answer is no, it is certain that point $(x_i, y_i)$ does not fall along any edge of the frequent spatial path corresponding to trie node $n_j$. Otherwise, we restore the reference and scan the frequent spatial path edge by edge to check whether the point is in $MBR(e_j, e_{j+1})$. If so, we restore and scan the shortest path between $e_j$ and $e_{j+1}$ to check if the point indeed falls on a certain edge. This process may repeat several times because $MBR(e_j, e_{j+1})$ is usually larger than the subspatial path; that is, a point falling inside $MBR(e_j, e_{j+1})$ might not be located at any edge along the shortest path from $e_j$ to $e_{j+1}$. Once we locate the point $(x_i, y_i)$ to an edge, we can obtain the distance $d_i'$ traveled by the object from the starting point of the journey until point $(x_i, y_i)$. Accordingly, we can search the compressed temporal sequence for $t_i'$. To sum up, $when(T, x_i, y_i)$ visits $\frac{cm}{2\alpha}$ trie nodes, $\frac{\alpha_2}{2c}$ edges, and $\frac{n}{2\beta}$ tuples in the temporal sequence on average.

Based on two basic queries introduced earlier, many other common queries can be designed by maintaining different kinds of information in the dictionary of HDC. For example, the range query $range(T, t_i, t_j, R)$ returns if a trajectory $T$ passes a region $R$ during the time period between $t_i$ and $t_j$. Given an original trajectory $T$, the range query first locates $d_i$ and $d_j$ for $t_i$ and $t_j$ in the temporal sequence. Then, it searches the spatial path for the part between $d_i$ and $d_j$ and scans the subspatial path to check if there is a line segment intersecting $R$. The range query also can be processed on partially decompressed trajectories. First, it locates $d_i'$ and $d_j'$ in a compressed temporal sequence. As we maintain MBRs in the dictionary, we scan the spatial path and check if the MBR intersects $R$ before recovering the original spatial path. Given a random interval $[t_i, t_j]$, we have to go through the trajectory until reaching $t_j$. The time cost is similar to that of the position query, but it scans $\frac{2}{3}$ part of the spatial path and the temporal sequence on average, so $range(T, t_i, t_j, R)$ visits $\frac{2cm}{3\alpha}$ trie nodes, $\frac{2\alpha_2}{3c}$ edges, and $\frac{2n}{3\beta}$ tuples.

For all queries mentioned previously, the process over compressed trajectories demonstrates certain nonnegligible advantages, as compared with the process over original trajectories. Though the gain in terms of performance relies on auxiliary structures carrying additional information, which actually causes additional computation cost and storage overhead, their construction can be preprocessed and the cost is shared by all trajectories including those generated in the future. All these auxiliary structures can be used for a relatively long duration unless the road network structure changes and/or the movement patterns of the underlying trajectories change significantly. Compared with the large number of trajectories generated daily and the long time period of collection we have to maintain, the extra storage cost incurred by these auxiliary structures can be well justified. On the other hand, the conventional approach on the original trajectories has to maintain an index for each trajectory to speed up the query processing, which cannot reduce the space cost by compressing trajectories but spends additional space in storing the indexes. We will compare all four approaches in the experimental study to show their performance in time and space.

### 6.3. Discussion on Data Utility

As a summary, we have presented different compression algorithms in Section 4 and Section 5. Those compression algorithms have different strengths in terms of compression effectiveness or query efficiency. As compression effectiveness can be evaluated by compression ratio, we only explain how to evaluate the utility of compressed data (i.e., query efficiency of different compression algorithms) in the following, mainly from two perspectives, that is, *precision* and *readability*. On the one hand, the precision of compressed data is measured by error metrics (e.g., TSND and NSTD). If the deviation of the compressed data from original data is low, the precision of compressed data is high. On the other hand, readability is relevant to the query algorithm designed for compressed data. If the query algorithm can easily extract certain information from encoded data, then the readability is high. Lossless compression does not cause any information loss, so the compressed data have the highest precision. However, lossless compression needs to encode the data so it is hard to preserve the readability. On the contrary, lossy compression skips unnecessary data so the readability is still high while its precision could be low.

As summarized in Table IV, data utility directly affects the performance of query processing. The utility of lossless compressed data mainly depends on readability since the precision does not change. Take spatial compression as an example. HDC is a one-stage encoder that encodes frequent subspatial paths as references in the dictionary, while L&C performs two-stage encoding, which first encodes edges to labels and

Table IV. Effects on Data Utility

| Type of Com. Algo. | Precision | Readability | Effects on Performance | Example |
|---|---|---|---|---|
| Lossless | High | Low | High compression ratio | L&C |
| | | High | High query efficiency | HDC |
| Lossy | Low | High | High query efficiency<br>High compression ratio | BTC |
| | High | | | |

then encodes labels by entropy coding. There is a bijection between frequent subpaths and references in the dictionaries, but it is hard to recover subpaths from sublabel sequences. Further encoding decreases the readability so it brings not only a higher compression ratio but also difficulty in processing queries on compressed data. In the case of lossy compression, the readability is usually high, but precision becomes the main constraint. For example, data compressed by BTC shares the same form as original temporal sequences, so the query algorithms still remain applicable. The higher the compression ratio BTC achieves, the higher the query efficiency it gains on compressed data, which is different from lossless compression.

To sum up, data utility is important if queries on compressed data are concerned. Data utility of lossless compressed data is very different from that of lossy compressed data, in terms of both precision and readability. In the COMPRESS framework, algorithms with different properties are proposed in order to cater for requirements from various applications.

## 7. EXPERIMENTS

In this section, we conduct extensive experiments to verify the previous proved theorems, to test the effectiveness and efficiency of various algorithms proposed, and to compare our COMPRESS framework against other existing methods on trajectory compression. The experiments are based on real trajectory data from one of the largest taxi companies in Shanghai.[6] All trajectories are sampled regularly by GPS stored in taxis. The road network of Shanghai is extracted from OpenStreetMap, which contains 60,456 vertices and 132,207 edges. All the algorithms are implemented with C and run on a computer with Intel Core i7-6820HK CPU@3.50GHz and 32GB memory.

The effectiveness of compression algorithms is measured by the compression ratio, while the efficiency of the algorithms is evaluated by the time taken. In our experiments, we apply different algorithms to real taxi trajectories and report their compression ratios as well as time complexities for comparison. In addition to algorithms proposed in the COMPRESS framework, we also implement our prior framework PRESS [Song et al. 2014] and another three state-of-the-art approaches for trajectory compression in road networks, including Non-material [Cao and Wolfson 2005], Map-matched Trajectory Compression (MMTC) [Kellaris et al. 2013], and Spatial Temporal Compression (STC) [Popa et al. 2015] as the representatives of existing approaches. Please refer to Section 8 for a detailed review of these existing works.

### 7.1. Direct Lossless Compression

Theorem 3.2 indicates that lossless compression algorithms are ineffective on trajectory data with high precision. However, the theorem does not unequivocally show how the compression ratio decreases when precision gets higher. In this set of experiments, we simply use Huffman coding and Lempel-Ziv coding to encode original trajectories to demonstrate the inefficiency of applying lossless compression algorithms on

[6]Though the raw dataset is not allowed to be published according to the terms and conditions, we have published spatial paths and temporal sequences as the sample data on GitHub [Han et al. 2016], which is sufficient for replicating the core compression algorithms proposed in the article.

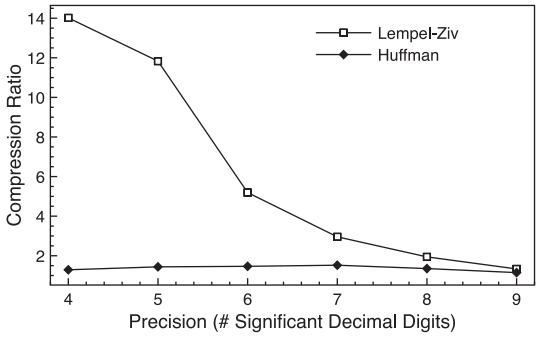



Fig. 29. Compression on original trajectory data.

high-precision trajectories represented as $\langle x_i, y_i, t_i \rangle$ sequences. First, we truncate every floating-point number in trajectories according to the precision given. For instance, given the precision 6, a number with six significant digits is considered to be accurate (e.g., 31.263319 is truncated to 31.2633). Then, we represent same truncated numbers by a unique integer. The translation is necessary because both Huffman coding and Lempel-Ziv coding take integer sequences as input. Note that three dimensions (i.e., $x$, $y$, and $t$) are encoded separately because their probability distributions are different. Finally, three integer sequences are passed to encoding algorithms and the corresponding compression ratio is recorded.

As shown in Figure 29,[7] the compression ratio decreases to 1.3 when the number of significant decimal digits reaches 9. Since we need at least nine decimal digits to guarantee the accuracy of trajectories, conventional lossless compression algorithms are not suitable for the GPS trajectories. In addition, Figure 29 shows that Huffman coding is not as effective as Lempel-Ziv coding because Huffman coding is a symbol-by-symbol encoder that does not consider the relationship between symbols.

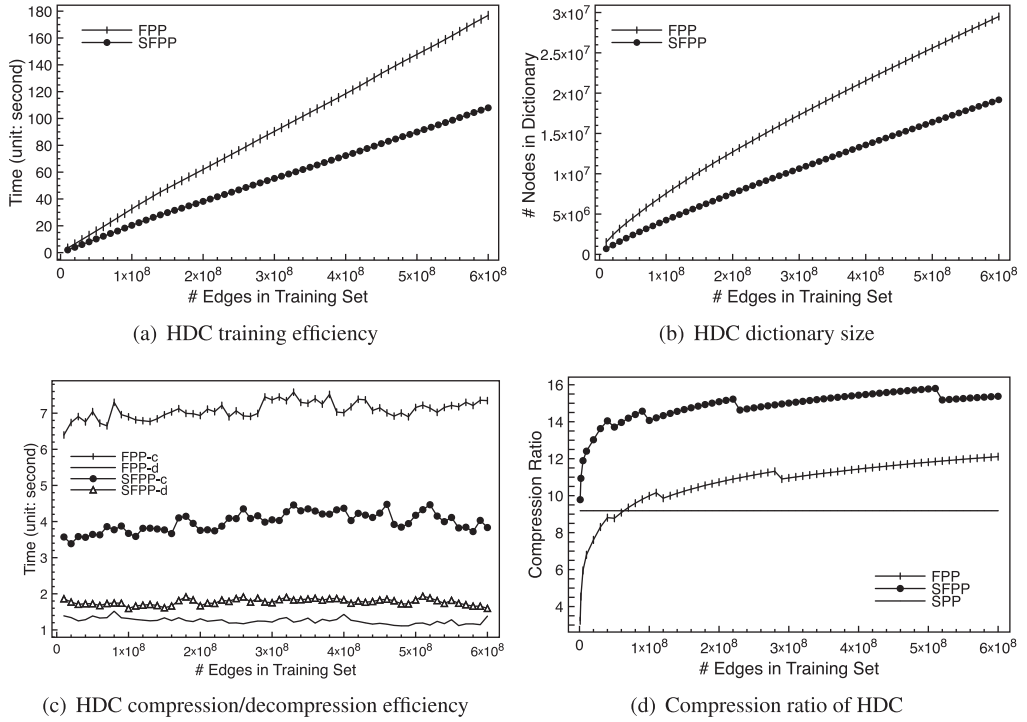
### 7.2. Hybrid Dictionary Compression

HDC performs Shortest-Path Priming (SPP) and Frequent Path Priming (FPP) to mine frequent patterns and then compresses the spatial paths.[8] SPP stores a two-dimensional array, in which every row stores a shortest-path tree of a source vertex by storing the precursor vertices of each vertex on the tree. It takes $O(|V|^2 \log |V| + |V||E|)$ time by applying *Dijkstra's algorithm* to each vertex, while the table occupies $O(|V|^2)$ space. In our experiment, given a road network containing 60, 456 vertices and 132, 207 edges, it takes 30.9 minutes to construct the dictionary and 13.2 seconds to load the dictionary to the memory. The dictionary occupies 6.80GB of storage space on the hard disk.

FPP stores frequent patterns via a tree structure, whose size depends on the number of nodes (i.e., references) it contains. FPP takes $O(\sum |T_i| \log |E|)$ time to mine frequent patterns from a training dataset, where the training set contains $\sum |T_i|$ edges. As shown in Figure 30(a) and Figure 30(b), the larger the training set is, the longer the training time and the higher the storage costs incurred by FPP. When a training set containing $6 \times 10^8$ edges is used, FPP and SFPP cost 151MB and 98MB to store dictionaries, respectively. Furthermore, SFPP takes less time and smaller space because

[7]Only results of at least four significant digits are presented in Figure 29 for better readability, since Theorem 3.2 applies to floating-point data with high precision.

[8]For convenience, we represent the combination of SPP and FPP by SFPP in short, and we simply represent "HDC with FPP" and "HDC with both SPP and FPP" by FPP and SFPP, respectively.




(a) HDC training efficiency

(b) HDC dictionary size

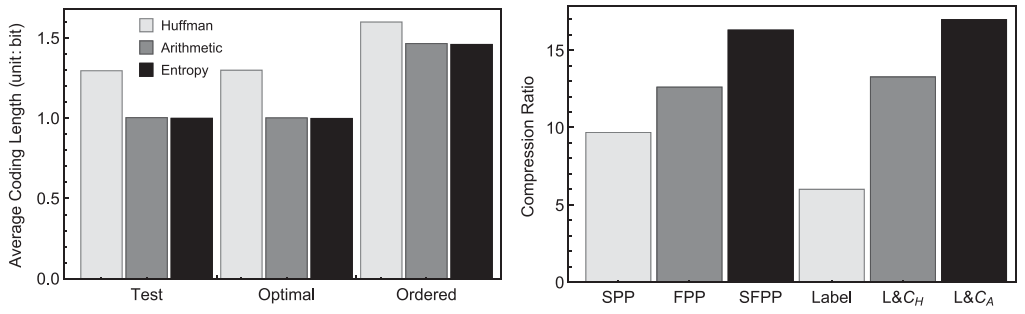
(c) HDC compression/decompression efficiency

(d) Compression ratio of HDC

Fig. 30. Performance of HDC versus training set size.

data processed by SPP contains fewer edges. As Figure 30(c) shows, the compression time and decompression time of HDC do not vary much as the training set size grows. Note that the subscript $c$ ($d$) next to FPP/SFPP in Figure 30(c) indicates compression (decompression). This is because the compression time complexity of PLZW is $O(|T|\log|E|)$, while that of decompression is $O(|T|)$ (here the test set contains $5 \times 10^7$ edges). Compared with FPP, SFPP takes less time in compression but a bit more time in decompression.

Both FPP and SFPP are effective in terms of compression ratio, which is shown in Figure 30(d). The compression ratio does not always increase as the size of the training set increases because more bits are needed to encode references when the dictionary size grows. When the training set contains $6 \times 10^8$ edges, the compression ratio of FPP and SFPP are 12.6 and 15.9, respectively, where the difference is not significant. However, the gap between FPP and SFPP is wide when the training set is relatively small. For example, given a training set of $10^6$ edges, the compression ratios of FPP and SFPP are 3.7 and 10.3, respectively. As a summary, SFPP is more effective than pure FPP in terms of compressing trajectories. Consequently, if the additional cost of SPP does not cause any concern, HDC with both SPP and FPP (i.e., SFPP) is preferred. However, when the road network becomes very big, the construction cost of the shortest-path trees and the storage overhead of the dictionary might not be negligible. In this case, HDC without SPP (i.e., only FPP) could be an alternative. Fortunately, as long as the training set contains sufficient edges, HDC with only FPP can also achieve a good compression ratio. In the following, we adopt HDC with SFPP as its default implementation, with a training set containing $10^8$ edges.




(a) MEL effetiveness (b) Compression ratio of L&C

Fig. 31. Performance of L&C.

### 7.3. Labeling and Coding (L&C) Compression

L&C first transforms spatial paths into label sequences and then applies an entropy coding algorithm. The entropy of label sequence should be as small as possible, since it bounds the performance of the later entropy coding algorithm. In Figure 31(a), we compare the optimal label and the ordered label in terms of entropy, where $H_{ordered} = 1.46$ and $H_{optimal} = 0.99$. The optimal label is generated by the MEL algorithm, while the ordered label is generated by assigning a label to each edge according to edges' input orders. Then, we apply an optimal label generated by a training dataset to the test dataset, and we observe that the difference of frequency distribution between two sets is very small. In fact, the entropy of the label sequence on the training set is 1.00, and that on the test set is 0.99.

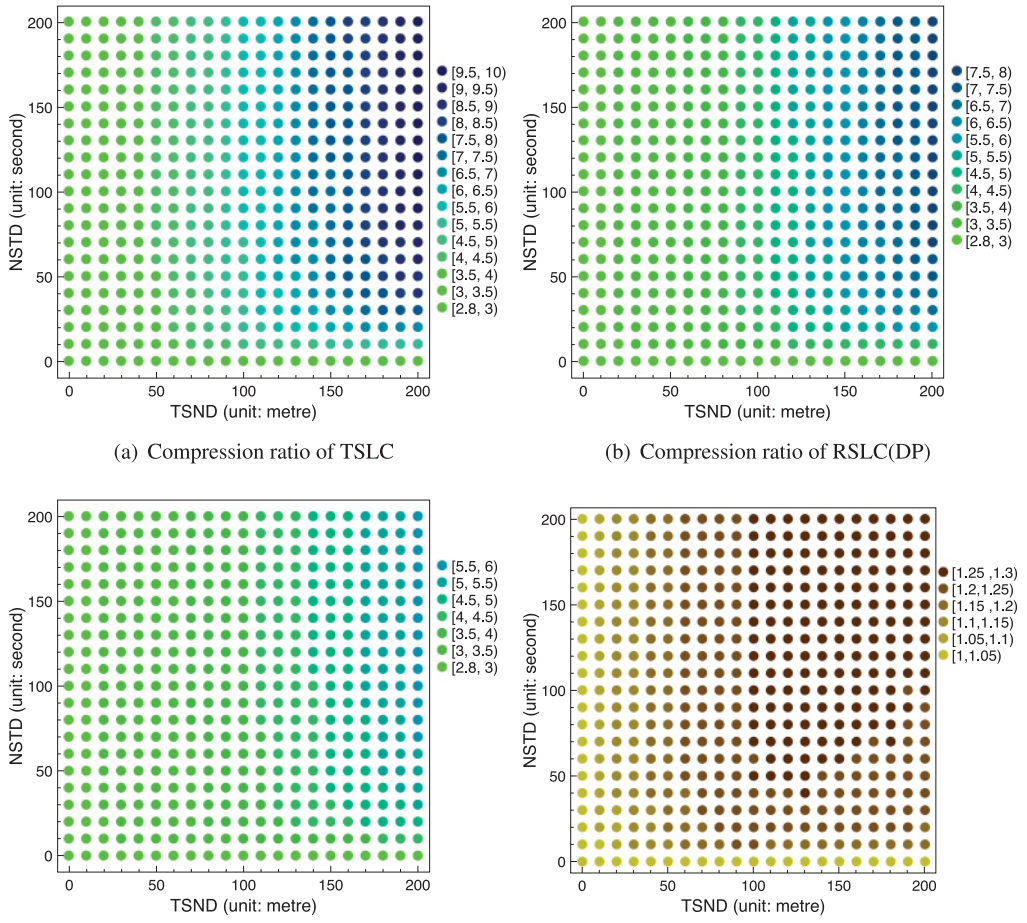
Given an alphabet $\Sigma = \{a, b, c\}$ with equal probabilities of $\frac{1}{3}$, a fixed-length code may be $C = \{00, 01, 11\}$ and its average code length is 2.000 bits per symbol. Meanwhile, Huffman coding ($H = \{0, 10, 11\}$) achieves 1.667 bits per symbol. Moreover, the average code length will be $\log_2 3 = 1.585$ for arithmetic coding. Arithmetic coding sometimes approaches the entropy more than Huffman coding does, because Huffman coding can only assign $k$ bits to a symbol, where $k$ has to be an integer. For example, given an alphabet containing two symbols whose probabilities are 0.1 and 0.9, Huffman coding will assign 1 bit to each symbol, achieving no compression. We compare arithmetic coding with Huffman coding in our experiment. As shown in Figure 31(a), arithmetic coding has a better coding effectiveness than Huffman coding on both datasets. The average code length of arithmetic coding is 1.00, while that of Huffman coding is 1.30 on the test set. Consequently, we suggest adopting arithmetic coding to improve compression effectiveness.

As shown in Figures 31(b) and 33(a), L&C not only achieves a higher compression ratio than HDC but also takes less time in compression and decompression. In addition, L&C only takes 49KB to store the optimal label, which is far less than the storage overhead of HDC.

### 7.4. Bounded Temporal Compression

The COMPRESS framework proposes two methods to compress temporal sequences within given tolerant error bounds, that is, Rigid Stabbing polyLine Compression (RSLC)[9] and Tube Stabbing polyLine Compression (TSLC). Figure 32 shows the compression ratio of TSLC, RSLC(DP), and RSLC, respectively, in which TSLC has the

[9]Although we introduce two different implementations of RSLC, including a dynamic programming approach and a greedy approach, denoted as RSLC(DP) and RSLC(greedy), respectively. When the context is clear, notation RSLC refers to RSLC based on greedy algorithm if not specially mentioned.



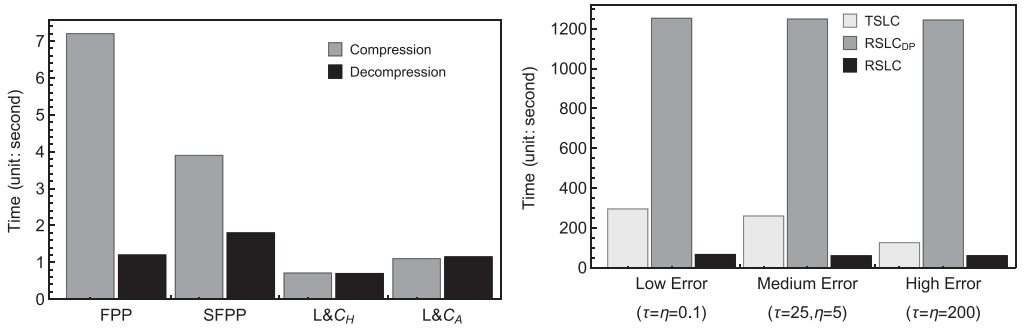

(a) Compression ratio of TSLC

(b) Compression ratio of RSLC(DP)

(c) Compression ratio of RSLC

(d) Improvement Ratio of TSLC over RSLC(DP)

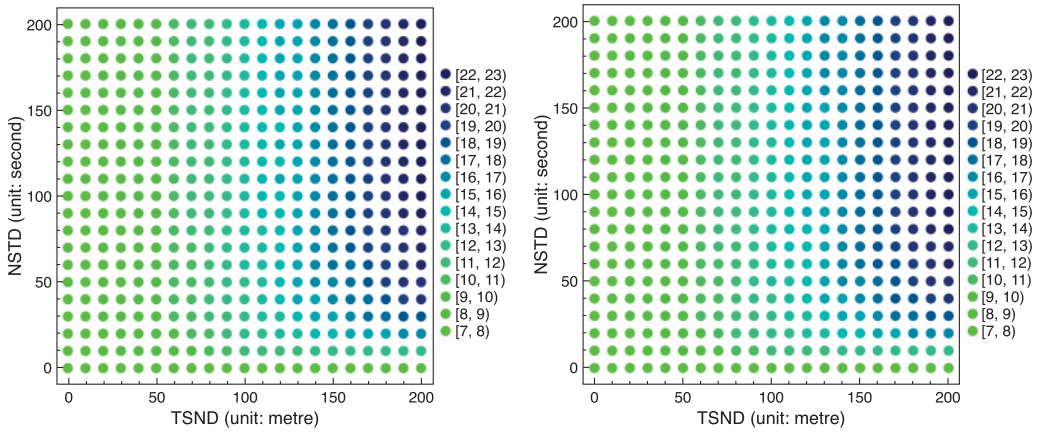
Fig. 32. Performance of TSLC, RSLC, and RSLC(DP).




(a) Efficiency of HDC and L&C

(b) Efficiency of TSLC and RSLC

Fig. 33. Efficiency of COMPRESS.




(a) HDC and TSLC (b) L&C and TSLC

Fig. 34. Compression ratio of COMPRESS.

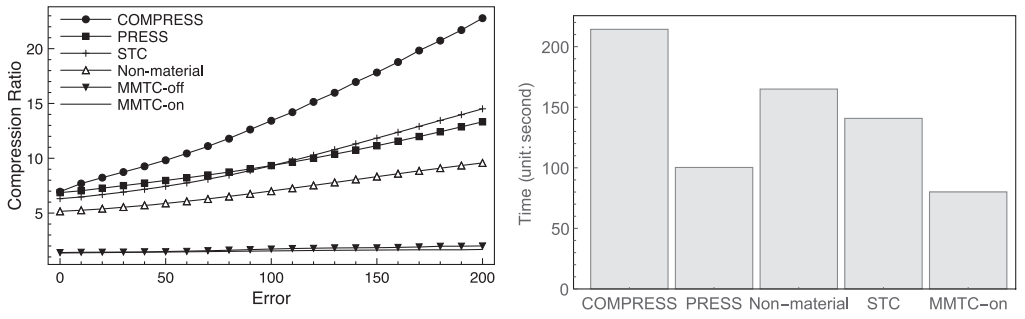
highest compression ratio and RSLC(DP) is more effective than greedy RSLC. Each dot in the figure records a range within which the compression ratio of the corresponding compression algorithm under specific TSND and NSTD settings falls. The darker the dot is, the higher the compression ratio (or improvement ratio reported in Figure 32(d)) is. Note that our algorithms still can achieve a compression ratio of 2.84 even if the error bounds are set to zero, because some points in temporal sequences are collinear (i.e., an object travels at a uniform speed). We further compare the performance of different temporal compression algorithms via the *improvement ratio*, which is defined as the ratio of one compression ratio to another compression ratio. Figure 32(d) shows the improvement ratio of TSLC over RSLC(DP), where TSLC on average is about 1.2 times more effective than RSLC(DP).

Furthermore, given a fixed TSND, the compression ratio grows rapidly first and then increases at a very slow speed when NSTD gets larger. On the other hand, the compression ratio rises smoothly when NSTD is fixed and TSND increases its value. This is because the average speed of the moving object is about 5m/s instead of 1m/s (i.e., 1s of NSTD is approximately equivalent to 5m of TSND), which means TSND is tighter in practice when both error bounds are used.

Both greedy RSLC and TSLC are linear algorithms, but RSLC(DP) has a time complexity of $O(n^2)$. As Figure 34(b) shows, RSLC(DP) takes much more time than TSLC and greedy RSLC for compressing the training set, which contains $10^8$ temporal tuples. As compared with greedy RSLC, TSLC is relatively more time-consuming because polygons contain more edges when error bounds are small. However, the edges of polygons will not exceed $4n + 4$, so the efficiency of TSLC is guaranteed. On the other hand, greedy RSLC always runs fast since it just scans $n$ points in the temporal sequence.

### 7.5. Summary of COMPRESS

Having presented the performance of spatial compression and temporal compression, we are ready to present the overall performance of COMPRESS. In terms of spatial compression, both HDC and L&C are recommendable because they have different advantages and can meet different application needs. In terms of temporal compression, we adopt TSLC as it outperforms RSLC consistently. We then report the overall compression ratio of COMPRESS in Figure 34, where COMPRESS under L&C and TSLC is more effective than that under HDC and TSLC. In addition to the compression ratio,




(a) Compression ratio (b) Compression time

Fig. 35. Comparison of existing works (test set size: $10^8$ sample points).

we also consider the time efficiency. COMPRESS is a linear algorithm, but its running time also depends on some variable factors. As shown in Figure 33, COMPRESS under L&C and TSLC is more efficient than that under HDC and TSLC because L&C is more efficient than HDC. In addition, the running time increases as the error bounds decrease, because TSLC takes more time if the error bounds are small.

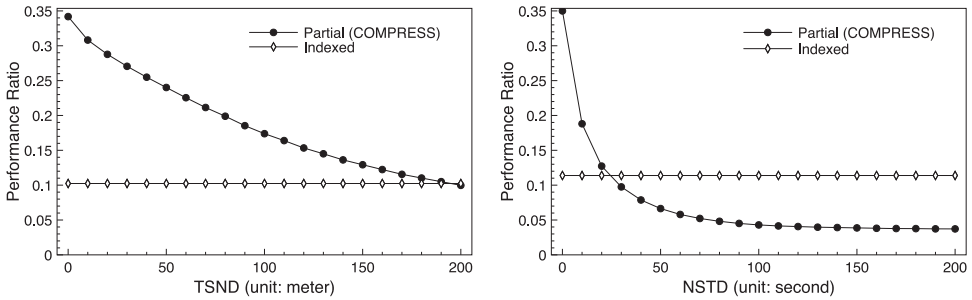
Compared with existing algorithms for trajectory compression including PRESS, Non-material, MMTC, and STC, COMPRESS is much more effective, as reported in Figure 35(a). Note that STC is bounded by only TSND, while both MMTC and Non-material are bounded by TSED. Consequently, the x-axis in Figure 35(a) is named as *error*, which means TSND for COMPRESS, PRESS, and STC but TSED for MMTC and Non-material. Since TSND is a *tighter* bound than TSED, COMPRESS, PRESS, and STC definitely will achieve an even higher compression ratio, as compared with the performance reported in Figure 35(a), if we replace TSND with TSED.

In addition to the effectiveness of different compression algorithms, we also report their efficiency in terms of total time taken in Figure 35(b). Among the six methods considered, offline MMTC is the most time-consuming. For example, when compressing the testing trajectory set, offline MMTC costs 3.3 hours, so it is not plotted in the figure for better readability. As shown in Figure 35(b), COMPRESS takes more time than other algorithms (except offline MMTC), because TSLC has a larger constant factor than other algorithms. However, taking into account that our test set contains $10^8$ sample points, the average execution time on a single trajectory is still acceptable. More information about those existing algorithms will be discussed in Section 8.

### 7.6. Query Processing

In order to compare the performance of different methods for processing queries, we randomly generate $3 \times 10^6$ queries on a test dataset and then process these queries based on different methods. The efficiency of query processing on partially decompressed data, denoted as Partial, is compared with that on original data, denoted as Original, and that on indexed data, denoted as Indexed. Note that we do not include the query processing on fully decompressed data in our comparison as it is very inefficient, taking even longer time than Original. We take query processing on the original data as a benchmark and report the performance ratio of other methods, including Partial and Indexed, that is defined as the ratio of their running time to the running time of Original.

The performance ratios of Partial and Indexed to that of Original for position query and time query are plotted in Figure 36(a) and Figure 36(b), respectively. Note that the label Partial actually refers to COMPRESS as COMPRESS allows us to process




(a) Position query (b) Time query

Fig. 36. Query processing efficiency of Partial and Indexed over that of Original.

location-based queries via partially decompressing the data. For Indexed, a query on a trajectory of length $|T|$ takes $O(\log |T|)$ time because we build balanced search trees on all trajectories to support efficient query processing. As Indexed always returns the accurate result, its deviations are always zero. In other words, the efficiency of Indexed is fixed and it cannot be improved even though we increase the error bounds to larger values. On the other hand, queries on Partial become more efficient when the error bounds increase. As observed from experiments, query processing on Partial is more efficient than that on Indexed when the deviation is greater than 200m in position queries and when the deviation is greater than 30s in time queries.

## 8. RELATED WORK

In this section, we review existing work related to trajectory compression, including commonly used lossless data compression algorithms, lossy compression algorithms on trajectory data, and other specific algorithms designed for road-network-based trajectory data.

### 8.1. Universal Lossless Compression

In information theory and computer science, data compression is the method that encodes information into a new form with fewer bits. Data compression can be either lossless or lossy according to features of data. Lossless compression reduces bits by mining statistical redundancy of data, so it does not lose any information during the compression and we can completely recover the data after decompression. On the other hand, lossy compression finds unnecessary data and removes them directly, so information may be lost during the compression. Though lossy compression may cause information loss, it can achieve a higher compression ratio than lossless compression does.

There are different kinds of lossless data compression algorithms, mainly classified into two types, namely, the *entropy* encoding and the *dictionary* encoding. Entropy encoding algorithms (e.g., Huffman coding, Shannon coding [Shannon 1948], and arithmetic coding) take possibility distribution of symbols as an input and are symbol-by-symbol coders that treat the data source as a stream of unrelated symbols. Consequently, entropy encoding may not achieve a high compression ratio on semantic data with many repeated patterns. However, dictionary coders (e.g., Lempel-Ziv algorithms) search repeated patterns in data and replace them with references of patterns in the data structure to compress data. They are optimal and are the most popular lossless data compression algorithms [Cover and Thomas 2006], which have been widely used in text and image compression.

Both Huffman coding (a representative of entropy encoding) and Lempel-Ziv coding (a representative of dictionary encoding) are proved to be effective by studies in information theory. According to *Shannon's source coding theorem*, $n$ independent and identically distributed random variables each with entropy $H$ cannot be compressed into fewer than $nH$ bits. The average coding length of Huffman coding achieves this entropy limit, where $H(X) \leq L(X) < H(X) + 1$. Arithmetic coding encodes the whole message into a number and its average coding length converges toward the entropy when length of the message goes to infinity. Consequently, both Huffman coding and arithmetic coding are optimal, though Huffman coding may need $n$ more bits than arithmetic coding does to encode $n$ symbols. Meanwhile, the compression ratio of Lempel-Ziv coding achieves the entropy rate of an ergodic source, which also shows its asymptotic optimality. In our work, we perform lossless compression based on those mentioned algorithms since they are optimal and the most representative algorithms.

Although these lossless compression algorithms have been widely used in many applications, they might not be able to outperform certain special methods proposed to compress specific data. Lossless compression algorithms benefit from *priming models* [Witten et al. 2001] built on preliminary prepared text when compressing a huge mass of data, as the PLZW algorithm does in Section 4. By analyzing and mining features of data, we transform the representation of data in such a way that becomes easier for existing algorithms to compress. This transformation strategy includes the well-known Burrows-Wheeler transform [Burrows and Wheeler 1994] in text compression, and the trajectory decomposition and labeling algorithm in our COMPRESS framework.

### 8.2. Lossy Compression for Trajectory Data

Lossy compression is useful if information loss is tolerated to achieve a higher compression ratio while the utility of the original data is partially preserved. Take temporal compression as an example. Compressed temporal sequences are in the same format as original temporal sequences, so it is convenient for other applications to process queries on compressed data.

Because it is hard to compress trajectory data by conventional lossless methods, algorithms based on line simplification are proposed. Line simplification is to approximate a polyline using another one containing fewer vertices, so it can be used to compress Euclidean space trajectories. The uniform sampling algorithm is simple and efficient but not error bounded, as it just keeps every $k^{th}$ points but discards others. The Ramer-Douglas-Peucker (RDP) [Douglas and Peucker 1973] algorithm recursively selects the point that causes the biggest error as a split point, until the trajectory satisfies the error requirement. However, RDP is very expensive as the time complexity of original RDP is $O(n^2)$ and that of improved implementations is $O(n \log n)$. Moreover, RDP is not optimal, which means it does not generate a polyline with a minimum number of vertices.

The problem of approximating ordered objects by the polygonal chain with the minimum number of line segments has been studied in Guibas et al. [1991], where three restrictions including "no restriction," "turn in tubes," and "turn in objects" are introduced to classify variations of the problem. In trajectory compression, the objects are disks centered at sampled points and the radius of the disks is the tolerant error. Furthermore, the tube is a region bounded by two consecutive disks and their outer common tangents. Representative algorithms to solve the minimum stabbing polyline problem include a greedy approach and a dynamic programming approach. The former is a linear greedy algorithm that may not minimize the size of the polyline and hence is not an ideal algorithm for trajectory compression, while the latter takes $O(n^2)$

time. In computational geometry literature, polyline simplification under "no restriction" and "turn in tubes" is called *weak polyline simplification*, while that under "turn in objects" is called *strong polyline simplification*. Strong polyline simplification only removes vertices from the original polyline, but weak polyline simplification may involve new vertices. For instance, the RDP algorithm is strong polyline simplification since the result polyline is a subset of the input polyline. Weak polyline simplification only takes $O(n)$ time to compute a minimal approximate polyline, but strong polyline simplification has not been solved, even when the polyline is monotone. In Varadarajan [1996], an algorithm that takes $O(n^{4/3+\epsilon})$ time is proposed to find a minimal polyline to approximate a monotone polyline. In Agarwal et al. [2005], a linear time algorithm is presented to compute an approximate polyline whose error is at most $\epsilon$ and the size is at most $\kappa(\epsilon/2, n)$, where $\kappa(\epsilon, n)$ donates the size of the optimal polyline, given a polyline containing $n$ vertices and an error bound $\epsilon$.

In our COMPRESS framework, temporal compression is also reduced to the stabbing polyline problem. Our solution RSLC corresponds to the restriction "turn in objects" (strong polyline simplification), while TSLC corresponds to "turn in tubes" (weak polyline simplification). The Window-Partition algorithm proposed in Suri [1990] is a linear algorithm searching for *minimum-link path* between two edges (or two vertices) in a simple polygon. Since the tube here can be divided into simple polygons, it only takes $O(n)$ to construct a minimum tube stabbing polyline by the divide-and-conquer approach. We use the TSLC to compress the temporal sequences and it is proved by our experimental study to be both efficient and effective.

A trajectory is a path that a moving object follows through space as a *continuous* function of time describing an object's motion, but in practice we can only obtain location samples at *discrete* time instants, and how an object moves between sample points is not represented in the database. In COMPRESS, we consider the GPS trajectory in road networks and assume that vehicles move with a fixed speed between two sample points, which leads to polyline simplification stated earlier. Different approximation methods are applied if the moving objects accelerate in short periods (e.g., in particle accelerators). For example, an algorithm to approximate an open polygonal curve with a minimum number of circular arcs and biarcs is proposed in Drysdale et al. [2008], while a method based on piecewise Chebyshev approximation is developed in Hagita et al. [2014] to compress particle trajectory data, where velocity varies a lot in short periods.

Cao et al. [2006] is the first work to discuss error-bounded query processing on lossy compressed trajectory data, which focuses on trajectories in 2D/3D spaces. In that work, the authors reduce the size of trajectories by polyline simplification algorithms. In total, four distance functions are defined, including two-dimensional Euclidean distance $E_2$, three-dimensional Euclidean distance $E_3$, three-dimensional time uniform distance $E_u$, and time distance $E_t$. Afterward, spatial-temporal queries including $where(T, t)$, $when(T, x, y)$, and $range(T, t_1, t_2, R)$ are introduced. They analyze the relationship called *soundness* between distance functions and spatiotemporal queries. For example, though the Euclidean distance between original trajectories and compressed ones is guaranteed after compression, results of $where(T, t)$ on compressed trajectories are not bounded by the Euclidean distance, so $E_2$ or $E_3$ is not sound for $where(T, t)$. In COMPRESS, we extend the technique to trajectory in road networks, where TSND is sound for $where(T, t)$ and $range(T, t_1, t_2, R)$ and NSTD is sound for $when(T, x, y)$. Different from completely lossy compression proposed in Cao et al. [2006], COMPRESS includes lossless spatial compression, which makes it hard to query encoded spatial paths. Consequently, we design HDC as well as query processing algorithms on partial decompressed data to overcome the difficulty, and the error-bounded queries are still supported.

Muckell et al. [2013] introduces a framework to evaluate the lossy trajectory compression algorithms, where both accuracy metrics and performance metrics are used. In Muckell et al. [2013], TSED is considered as a representative accuracy metric, while temporal metrics include compression ratio and compression time. The benchmark framework is designed for those lossy compression algorithms that directly discard unnecessary points and produce a subset of input trajectory as an output trajectory and hence is not applicable to COMPRESS since COMPRESS is partially lossy as both trajectory decomposition and spatial compression are lossless. However, the metrics discussed in Muckell et al. [2013] are still considered in our experiments, including both accuracy metrics (TSND and NSTD) and performance metrics (compression ratio and compression time).

### 8.3. Trajectory Compression in Road Networks

Trajectory data in urban spaces is restricted by road networks; that is, the trajectories have to fall on road segments. Consequently, it is natural to use map-matching algorithms to filter the GPS sensor errors and to map GPS trajectories onto the road network. There are many existing map-matching algorithms [Brakatsoulas et al. 2005; Lou et al. 2009; Newson and Krumm 2009; Song et al. 2012]. They take trajectory data as well as the road network as an input and return the trajectories as sequences of road segments. Our COMPRESS framework first projects the trajectory onto the road network via existing map-matching algorithms and then invokes compression algorithms to compress the trajectory. Here, we review four representatives of trajectory compression in road networks, including our prior framework PRESS [Song et al. 2014], Non-material [Cao and Wolfson 2005], MMTC [Kellaris et al. 2013], and STC [Popa et al. 2015].

Our prior framework PRESS first decomposes the trajectory into a spatial path and a temporal sequence and then compresses the spatial path and the temporal sequence separately. The spatial compression algorithm first applies shortest-path compression and then uses Huffman coding to encode subspatial paths containing no more than $\theta$ (e.g., $\theta = 3$ in the experiments) edges; its temporal compression is an error-bounded algorithm, which is similar to the greedy approach in RSLC. The COMPRESS framework shares some similarities as PRESS. However, we want to highlight that the COMPRESS framework brings *significant* enhancements to the PRESS framework in the following three aspects. First, the COMPRESS framework studies the limitations of the conventional representation of trajectory data in terms of compression and proves the advantages of trajectory decomposition formally. Second, the spatial compressor in the COMPRESS framework implements two new spatial compression algorithms with both outperforming the algorithm used in PRESS. Third, the temporal compressor implements a new algorithm TSLC, which is an optimal algorithm for bounded temporal compression.

Non-material does not take advantage of the map-matching algorithm, but it introduces a novel algorithm for "adjusting" the trajectory to fit the road network to generate road-snapped trajectory. Thereafter, it "nonmaterializes" the road-snapped trajectory and separates temporal information from the spatial information,[10] which is similar to the decomposition stage in COMPRESS. The "nonmaterialize" stage in Non-material inspires us to study the limitation of the original representation and to explain the reason we decompose trajectory data. However, our COMPRESS framework outperforms Non-material in the following aspects. First, Non-material does not compress the spatial

[10]See Definition 3 in Cao and Wolfson [2005]: "A non-materialized trajectory $T$ is a function from time to map locations represented as a sequence of tuples $(\langle p_1, l_1, t_1\rangle, \ldots \langle p_m, l_m, t_m\rangle)$, where each $p_i$ is a street in $P$, $l_i$ is a real number that indicates $T$'s location at time $t_i$ in $p_i$'s linear reference coordinate."

information, while COMPRESS implements HDC and L&C, two effective compression algorithms, to compress spatial paths. Second, Non-material only uses the TSED metric when compressing temporal information, so it cannot guarantee a time difference between a trajectory and its compressed form, while COMPRESS considers both TSND and NSTD to ensure that the deviations of the compressed trajectory from the original one in both the time dimension and distance dimension are bounded. Last but not least, Non-material simply reduces the temporal compression to the stabbing polyline problem defined in Definition 5.5, which can only guarantee the error at original sampled timestamps. On the other hand, both RSLC and TSLC in COMPRESS guarantee the error metrics bounded over the whole time and distance range, as mentioned in Section 5.

MMTC is another algorithm that focuses on trajectory compression in road networks. It combines map-matching algorithms together with lossy trajectory compression algorithms. Moreover, a specific evaluation function called *trajectory similarity* is introduced to guarantee that the compressed trajectory is similar to the original one. MMTC assumes that all sample points in map-matched trajectories should be exactly projected to the vertices of the road network.[11] Based on this assumption, MMTC replaces parts of trajectory with the shortest paths to reduce the storage cost, since points are all vertices of the road network and shortest paths between them can be computed. The lossy compression algorithms in MMTC include an online algorithm and an offline algorithm, both of which replace certain subpaths by shortest paths if the shortest paths contain fewer vertices. The offline algorithm computes all shortest paths between every two points and takes $O(n^2 \log n)$ time, while the online algorithm only computes shortest paths between every continuous $r$ point and takes $O(rn \log r)$ time, where $r$ is a predefined threshold. Offline MMTC is time-consuming, while online MMTC is not as effective as offline MMTC in terms of compression ratio. The assumption that points are projected to vertices leads to the deviation from the original map-matched trajectory, because most points in map-matched trajectories are projected to edges instead of vertices, as Figure 4 shows. Furthermore, the shortest path could be quite different from the original subtrajectory. In other words, the compression ratio of offline MMTC is limited if the error is bounded.

STC is the last algorithm that is included in the experimental study as a competitor of our framework. First, that work analyzes the limitation of existing lossy compression algorithms and considers the data model related to the network space instead of 2D space. In the network model, the location of a moving object is represented by $\langle rid, pos \rangle$, where $rid$ is a road identifier and $pos \in [0, 1]$ is the relative location on the road ($pos = 0$ for the start point and $pos = 1$ for the end point). Then the authors present their generalized data model of in-network trajectories, which eliminates the unnecessary time information in the trajectory representation. Afterward, STC compresses trajectories through two algorithms, including a network partitioning algorithm and a strong polyline simplification algorithm. The network partitioning algorithm partitions the graph into longer roads, after which a trajectory may traverse fewer roads so the storage is reduced. In order to apply the polyline simplification algorithm, STC transforms the generalized trajectory into the sequence of all timestamped locations, that is, a series of $\langle rid, pos, t \rangle$, which is similar to the nonmaterialized trajectories introduced in Cao and Wolfson [2005]. The error metric used in STC is TSND and the strong polyline simplification algorithm is the optimal line algorithm introduced in Imai and Iri [1988], which takes $O(n \log n)$ time.

[11]See Definition 3 in Kellaris et al. [2013]: "A trajectory T is considered to be map-matched to a network map $G = (V, E)$ if all the $\langle x, y \rangle$ coordinates of its tuples belong to $V$ and they are connected to each adjacent through an edge of $E$."

STC is similar to COMPRESS in many aspects, but there are still some differences. First, GPS locations are usually reported in the form of $\langle x, y \rangle$ instead of $\langle rid, pos \rangle$, where $x$ and $y$ are latitude and longitude, respectively. STC assumes that all sample points are right on the edges, but COMPRESS can deal with deviation from the road network on the basis of map-matching algorithms, so COMPRESS is more suitable for trajectory compression in real transport networks. Second, the reduction of spatial information via network partition is limited. Each generalized unit computed by STC contains at least one road identifier,[12] so the compression ratio of road segments will not exceed the compression ratio of the $\langle rid, pos, t \rangle$ sequence, which is at most 7.8 in our experiments, while spatial compression algorithms in COMPRESS can achieve a compression ratio higher than 16. Third, STC applies strong polyline simplification, whose effectiveness is equivalent to RSLC(DP). As shown in Section 7, TSLC is more effective than RSLC, so COMPRESS also does better in temporal compression. Finally, COMPRESS implements not only TSND but also NSTD, based on which position and time queries can be efficiently processed.

## 9. CONCLUSION

In this article, we propose a comprehensive framework called COMPRESS to reduce the size of trajectory data in road networks. First, we prove that lossless compression on the original trajectory can hardly reduce the size of data, especially under high precision. Then, we introduce a simple decomposition method to separate spatial data from temporal data, which transforms the representation and makes it easier to design compression algorithms. Afterward, several specific algorithms are designed according to different data properties. We want to highlight that COMPRESS optimizes these specific algorithms and hence it is able to achieve a much better compression ratio. Last but not least, COMPRESS is also able to support error-bounded queries on partially decompressed data. A comprehensive experimental study has been performed to evaluate the efficiency and effectiveness of the COMPRESS framework.

Data compression is a fundamental problem in information theory and computer science. Many universal compression algorithms like Huffman coding and Lempel-Ziv coding have been proposed during the last century. These algorithms are optimal and have a sound theoretical basis in information theory. However, it is ineffective to apply them on different kinds of data without any pretreatment, so transformation before compression is necessary. Consequently, we wonder if there is any better strategy to improve the representation of trajectory data. Moreover, we *extract* compressible information (i.e., spatial paths) in our trajectory decomposition algorithm. It is interesting to investigate whether this *extraction strategy* is suitable for other kinds of data.

The COMPRESS framework not only reduces the size of data but also tries to retain the utility of data so that the error-bounded queries can be processed efficiently on partially decompressed data. Compressing and indexing are usually contrary to each other, but in HDC we combine them together, which leads to low storage cost and high processing efficiency. In Section 6, we have evaluated data utility from two perspectives including precision and readability. Data precision is defined in Section 5, but we find it difficult to define data readability. Consequently, clearly defining data utility and efficiently processing queries on lossless/lossy compressed data are also interesting research directions.

[12]In the best case, the trajectory compressed by STC is contained by a single road, so each unit has only one road identifier.

## REFERENCES


Pankaj K. Agarwal, Sariel Har-Peled, Nabil H. Mustafa, and Yusu Wang. 2005. Near-linear time approximation algorithms for curve simplification. *Algorithmica* 42 (2005), 203–219.

P. Arpaia, M. Buzio, and V. Inglese. 2010. A two-domain, real-time algorithm for optimal data reduction: A case study on accelerator magnet measurements. *Measurements Science and Technology* 21, 3 (2010), 035103.

Rudolf Bayer. 1972. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica* 1, 4 (1972), 290–306.

Rudolf Bayer and Edward M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (1972), 173–189.

Chazelle Bernard. 1991. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry* 6 (1991), 485–524.

Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk. 2005. On map-matching vehicle tracking data. In *Proceedings of the VLDB Endowment*. 853–864.

Michael Burrows and David J. Wheeler. 1994. *A Block Sorting Lossless Data Compression Algorithm*. Retrieved from http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html.

Hu Cao and Ouri Wolfson. 2005. Nonmaterialized motion information in transport networks. In *International Conference on Database Theory (ICDT'05)*. Vol. 3363. 173–188.

Hu Cao, Ouri Wolfson, and Goce Trajcevski. 2006. Spatio-temporal data reduction with deterministic error bounds. *VLDB Journal* 15, 1 (2006), 211–228.

Yi Chiang and Roberto Tamassia. 1997. Optimal shortest path and minimum-Link path queries between two convex polygons in the presence of obstacles. *International Journal of Computational Geometry* 7 (1997), 85–121.

Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory* (2nd ed.). Wiley-Interscience, Chapter 5: Data Compression & 13: Universal Source Coding.

Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 2000. *Computational Geometry: Algorithms and Applications* (2nd ed.). Springer-Verlag.

Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 1 (1959), 269–271.

David Douglas and Thomas Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer* 10, 2 (1973), 112–122.

R. L. Scot Drysdale, Gunter Rote, and Astrid Sturm. 2008. Approximation of an open polygonal curve with a minimum number of circular arcs and biarcs. *Computational Geometry* 41, 1–2 (2008), 31–47.

Joseph S. B. Mitchell Esther M. Arkin and Subhash Suri. 1995. Logarithmic-time link path queries in a simple polygon. *International Journal of Computational Geometry* 5, 4 (1995), 369–395.

Chenglin Fan, Jun Luo, and Binhai Zhu. 2010. Fréchet-distance on road networks. In *Computational Geometry, Graphs and Applications*. 61–72.

Leonidas J. Guibas, John E. Hershberger, Joseph S. B. Mitchell, and Jack Scott Snoeyink. 1991. Approximating polygons and subdivisions with minimum link paths. In *ISA'91 Algorithms*. Vol. 557. 151–162.

Katsumi Hagita, Hiroaki Ohtani, Tsunehiko Kato, and Seiji Ishiguro. 2014. TOKI compression for plasma particle simulations. *Plasma and Fusion Research* 9 (2014), 3401083.

Yunheng Han, Weiwei Sun, and Baihua Zheng. 2016. Sample data and the online appendix. Retrieved from https://github.com/MasterChivu/TODS-EXP.

David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*. Vol. 40. 1098–1101. Issue 9.

Hiroshi Imai and Masao Iri. 1988. Polygonal approximations of a curve formulations and algorithms. *Machine Intelligence and Pattern Recognition* 6 (1988), 71–86.

Georgios Kellaris, Nikos Pelekis, and Yannis Theodoridis. 2013. Map-matched trajectory compression. *Journal of Systems & Software* 86, 6 (2013), 1566–1579.

Donald E. Knuth. 1997. *The Art of Computer Programming Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley, Chapter 6.3: Digital Searching.

Abraham Lempel and Jacob Ziv. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

Abraham Lempel and Jacob Ziv. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.

Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang. 2009. Map-matching for low-sampling-rate GPS trajectories. In *Proceedings of the 17th ACM SIGSPATIAL GIS*. 352–361.

Joseph S. B. Mitchell, Gnter Rote, and Gerhard Woeginger. 1992. Minimum-link paths among obstacles in the plane. *Algorithmica* 8, 1–6 (1992), 431–459.

Jonathan Muckell, Paul W. Olsen Jr., Jeong-Hyon Hwang, S. S. Ravi, and Catherine T. Lawson. 2013. A framework for efficient and convenient evaluation of trajectory compression algorithms. In *Proceedings of the 2013 4th International Conference on Computing for Geospatial Research and Application*. 24–31.

Paul Newson and John Krumm. 2009. Hidden markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL GIS*. 336–343.

OHPI. 2014. Highway Statistics Series. Retrieved from http://www.fhwa.dot.gov/policyinformation/statistics/2011/vm2.cfm.

Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, and Ahmed Kharrat. 2015. Spatio-temporal compression of trajectories in road networks. *GeoInformatica* 19, 1 (2015), 117–145.

Jorma J. Rissanen. 1976. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development* 30, 3 (1976), 198–203.

Zhangqing Shan, Hao Wu, Weiwei Sun, and Baihua Zheng. 2015. COBWEB: A robust map update system using GPS trajectories. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 927–937.

Claude E. Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal* 27, 3 (1948), 379–243.

Renchu Song, Wei Lu, Weiwei Sun, Yan Huang, and Chunan Chen. 2012. Quick map matching using multi-core CPUs. In *Proceedings of the 20th ACM SIGSPATIAL GIS*. 605–608.

Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. 2014. PRESS: A novel framework of trajectory compression in road networks. In *Proceedings of the VLDB Endowment*. Vol. 7. 661–672. Issue 9.

Subhash Suri. 1990. On some link distance problems in a simple polygon. *IEEE Journal of Robotics and Automation* 6, 1 (1990), 108–113.

K. R. Varadarajan. 1996. Approximating monotone polygonal curves using the uniform metric. In *Symposium on Computational Geometry*.

Terry Welch. 1984. A technique for high-performance data compression. *Computer* 17, 6 (1984), 8–19.

Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 2001. *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd ed.). Elsevier, Chapter 2.7: Synchronization.

Hao Wu, Chuanchuan Tu, Weiwei Sun, Baihua Zheng, Hao Su, and Wei Wang. 2015. GLUE: A parameter-tuning-free map updating system. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management*. 683–692.

Received August 2015; revised August 2016; accepted November 2016