Trajectory Simplification: On Minimizing the Direction-based Error

Cheng Long[†], Raymond Chi-Wing Wong[†]
[†]The Hong Kong University of Science and Technology
{clong, raywong}@cse.ust.hk

H. V. Jagadish[‡]
[‡]University of Michigan
jag@umich.edu

ABSTRACT

Trajectory data is central to many applications with moving objects. Raw trajectory data is usually very large, and so is simplified before it is stored and processed. Many trajectory simplification notions have been proposed, and among them, the direction-preserving trajectory simplification (DPTS) which aims at protecting the direction information has been shown to perform quite well. However, existing studies on DPTS require users to specify an error tolerance which users might not know how to set properly in some cases (e.g., the error tolerance could only be known at some future time and simply setting one error tolerance does not meet the needs since the simplified trajectories would usually be used in many different applications which accept different error tolerances). In these cases, a better solution is to minimize the error while achieving a pre-defined simplification size. For this purpose, in this paper, we define a problem called *Min-Error* and develop two exact algorithms and one 2-factor approximate algorithm for the problem. Extensive experiments on real datasets verified our algorithms.

1. INTRODUCTION

Trajectory data, which records the traces of moving objects, is ubiquitous nowadays due to the popularity of GPS devices. People use trajectory data for many different purposes, e.g., traffic analysis [23], route recommendation [26, 35], social relationship analysis [30, 37], and user behavior analysis [39, 33].

Raw trajectory data is usually of large volume, which incurs high storage and processing costs. Thus, common practice is to simplify the raw trajectory data before it is stored and processed. This procedure is called *trajectory simplification* [24].

Most existing trajectory simplification techniques aim to preserve the *position information* when trajectory data is simplified, which is referred to as *position-preserving trajectory simplification* (PPTS) [4, 31, 27]. PPTS guarantees that at any time stamp, the distance is bounded between the position captured by the original trajectory and the position captured by the simplified trajectory.

Recently, Long et al. [24] proposed a new trajectory simplification framework called *direction-preserving trajectory simplification* (DPTS) which aims to preserve the *direction information* when trajectory data is simplified. DPTS guarantees that at any time stamp, the *angular difference* between the direction of the movement captured by the original trajectory and the direction of the movement captured by the simplified trajectory, which corresponds to the *error* of the simplified trajectory, is bounded. According to [24], DPTS is superior over PPTS since DPTS not only preserves the direction information but also bounds position information loss, but the converse is not true. In this paper, we focus on DPTS.

The DPTS problem studied in [24] is to simplify a given trajectory such that the error of the simplified trajectory is at most a given *error threshold* and its *size* is minimized. Here, the size of a simplified trajectory is defined to be the total number of positions kept in the trajectory. We call this problem the *Min-Size* problem. The Min-Size problem is suitable only when users have clear knowledge about the error tolerance.

In some cases, users might not know how to specify the error tolerance clearly. This could be because the simplified trajectories will be used *in the future* and thus the details are not available at the moment or the simplified trajectories will be used in different applications which might have different accuracy requirements and thus it is not suitable to specify *one* error tolerance for simplifying trajectories. In these cases, a better way is to retain the accuracy as much as possible while achieving a certain degree of compression rate for simplifying trajectories. Specifically, we are given a *storage budget* denoting the greatest size of a simplified trajectory to be stored (note that the storage budget implies a compression rate requirement), and the goal is to minimize the error of the simplified trajectory. We call this problem the *Min-Error* problem which corresponds to the dual problem of the Min-Size problem.

In this paper, we develop multiple algorithms for the Min-Error problem, both exact and faster approximate algorithms. Specifically, our major contributions are summarized as follows. First, we define a new problem called *Min-Error* which minimizes the simplification error under a storage budget. Second, to solve the Min-Error problem exactly, we explore the idea of *dynamic programming* and *binary search*, resulting in two different algorithms, with the time complexities of $O(Wn^3)$ and $O(n^2 C \log n)$, respectively (W is the storage budget, n is the size of the trajectory and C is a small constant). Third, motivated by the relatively high complexities of the exact algorithms, we further develop an approximate algorithm which runs in $O(n \log^2 n)$ time and gives a 2-factor approximation. Fourth, we conducted extensive experiments on real datasets which verified our proposed algorithms.

The remainder of this paper is organized as follows. Section 2 defines the Min-Error problem. Section 3 and Section 4 introduce our exact and approximate algorithms, respectively. Section 5 gives our empirical study. Section 6 studies the related work and Section 7 concludes the paper.

Proceedings of the VLDB Endowment, Vol. 8, No. 1

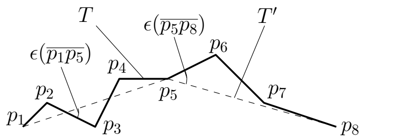


Figure 1: A trajectory (used as the running example throughout this paper)

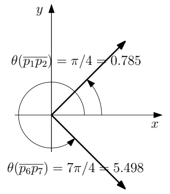


Figure 2: direction

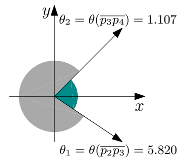


Figure 3: angular range

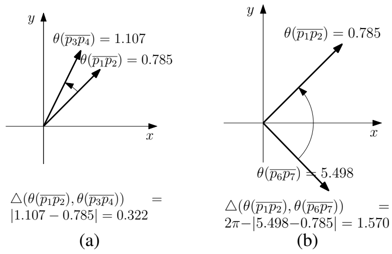


Figure 4: angular difference

Segments	$\overline{p_1p_2}$	$\overline{p_2p_3}$	$\overline{p_3p_4}$	$\overline{p_4p_5}$	$\overline{p_5p_6}$	$\overline{p_6p_7}$	$\overline{p_7p_8}$
Directions	0.785	5.820	1.107	0	0.464	5.498	5.961

Table 1: Directions of the segments (running example)

2. PROBLEM DEFINITION

A trajectory corresponds to the spatial and temporal trace of a moving object and is usually represented by a sequence of (position, time stamp)-pairs: $(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)$ which implies that the object is located at position p_i at time stamp t_i for $i = 1, 2, ..., n$. An implicit assumption here which is commonly used is that the object moves along the *line segment* linking p_i and p_{i+1} from time stamp t_i to time stamp t_{i+1} for $i = 1, 2, ..., n-1$.

Since we aim at preserving the *direction information* of a trajectory (which will be introduced later) when doing trajectory simplification and the direction information of a trajectory is solely captured by the sequence of positions of the trajectory, we represent the trajectory by the sequence of its positions only, e.g., $T = (p_1, p_2, ..., p_n)$ corresponds to a trajectory of the moving object appearing at $p_1, p_2, ..., p_n$ sequentially. We define the *size* of T, denoted by $|T|$, as the number of positions involved in T.

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory. Each line segment linking two adjacent positions p_h and p_{h+1} $(1 \leq h \leq n-1)$ in a trajectory which we denote by $\overline{p_hp_{h+1}}$ is called a *segment* of the trajectory. That is, T with size n involves $n-1$ segments.

To illustrate, consider Figure 1 where there is a trajectory $T = (p_1, p_2, ..., p_8)$. T has 8 positions, i.e., $p_1, p_2..., p_8$, and thus the size of T is equal to 8. T has 7 segments, i.e., $\overline{p_1p_2}, \overline{p_2p_3}, ..., \overline{p_7p_8}$, which correspond to the solid line segments in the figure.

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory. We denote by $T[i:j]$ the portion of T from p_i to p_j, i.e., $T[i:j] = (p_i, p_{i+1}, ..., p_j)$.

We say that trajectory T' is a *simplification* of T if $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ where $m \leq n$ and $1 = s_1 < s_2 < ... < s_m = n$. That is, any trajectory resulted from T by dropping some positions (that are not the first nor the last position) corresponds to a simplification of T. T' has m positions and $m-1$ segments. Segment $\overline{p_{s_k}p_{s_{k+1}}}$ in T' $(1 \leq k \leq m-1)$ *approximates* the sequence of segments between p_{s_k} and $p_{s_{k+1}}$ in T, namely $\overline{p_{s_k}p_{s_k+1}}, \overline{p_{s_k+1}p_{s_k+2}}, ..., \overline{p_{s_{k+1}-1}p_{s_{k+1}}}$.

To illustrate, consider our running example in Figure 1. $T' = (p_1, p_5, p_8)$ is a simplification of T. T' has 3 positions and 2 segments which correspond to the dash line segments in the figure. Segment $\overline{p_1p_5}$ in T' approximates $\overline{p_1p_2}, \overline{p_2p_3}, \overline{p_3p_4}, \overline{p_4p_5}$ in T and $\overline{p_5p_8}$ in T' approximates $\overline{p_5p_6}, \overline{p_6p_7}, \overline{p_7p_8}$ in T.

Direction-based Error Measurement E_d. [24] Let $\overline{p_ip_j}$ be a line segment. The *direction* of $\overline{p_ip_j}$, denoted by $\theta(\overline{p_ip_j})$, is defined to be the angle of an *anticlockwise* rotation from the positive x-axis to a vector from p_i to p_j. Thus, all directions fall in range $[0, 2\pi)$.

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory. We denote by $\theta[i:j]$ the set containing the directions of the segments between p_i and p_j, i.e., $\theta[i:j] = \{\theta(\overline{p_hp_{h+1}}) | h \in [i, j)\}$.

For example, Figure 2 shows that $\theta(\overline{p_1p_2}) = 0.785$ radian and $\theta(\overline{p_6p_7}) = 5.498$ radian. In the following, by default, all angles/directions are measured in radians. The directions of all segments of T in Figure 1, i.e., $\theta[1:8]$, are shown in Table 1.

Let θ_1 and θ_2 be two directions. An *angular range* in the form of $[\theta_1, \theta_2]$ is defined to be the range of all possible directions of a vector when it is rotated *anticlockwise* from θ_1 to θ_2. To illustrate, consider Figure 3. Angular range $[\theta_1, \theta_2]$ covers all the directions in the sector area in darker color while angular range $[\theta_2, \theta_1]$ covers all the directions in the sector area in lighter color.

Let θ_1 and θ_2 be two directions. The *angular difference* between θ_1 and θ_2, denoted by $\triangle(\theta_1, \theta_2)$, is defined to be the smaller one between the angle of an anticlockwise rotation from θ_1 to θ_2 and that from θ_2 to θ_1. We have

$$\triangle(\theta_1, \theta_2) = \min\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\} \quad (1)$$

To illustrate, consider Figure 4(a) where $\triangle(\theta_1, \theta_2) = |\theta_1 - \theta_2|$ and Figure 4(b) where $\triangle(\theta_1, \theta_2) = 2\pi - |\theta_1 - \theta_2|$. Note that the angular difference between two directions is symmetric, i.e., $\triangle(\theta_1, \theta_2) = \triangle(\theta_2, \theta_1)$.

Let $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be a simplification of T. The *simplification error* of segment $\overline{p_{s_k}p_{s_{k+1}}}$ in T', denoted by $\epsilon(\overline{p_{s_k}p_{s_{k+1}}})$, is defined to be the *greatest* angular difference between the direction of $\overline{p_{s_k}p_{s_{k+1}}}$ and the direction of a segment in T that is approximated by $\overline{p_{s_k}p_{s_{k+1}}}$. That is,

$$\epsilon(\overline{p_{s_k}p_{s_{k+1}}}) = \max_{s_k \leq h < s_{k+1}} \triangle(\theta(\overline{p_{s_k}p_{s_{k+1}}}), \theta(\overline{p_hp_{h+1}}))$$

Then, the *simplification error* of T', denoted by $\epsilon(T')$, is defined to be the *greatest* simplification error of its segments [24]. That is,

$$\epsilon(T') = \max_{1 \leq k < m} \epsilon(\overline{p_{s_k}p_{s_{k+1}}}) \quad (2)$$

To illustrate, consider our running example in Figure 1. Suppose $T' = (p_1, p_5, p_8)$. Since $\theta(\overline{p_1p_5}) = 0.322$ and $\theta(\overline{p_1p_2}) = 0.785$, we know $\triangle(\theta(\overline{p_1p_5}), \theta(\overline{p_1p_2})) = 0.463$. Similarly, we can compute $\triangle(\theta(\overline{p_1p_5}), \theta(\overline{p_2p_3})) = 0.785$, $\triangle(\theta(\overline{p_1p_5}), \theta(\overline{p_3p_4})) = 0.785$, and $\triangle(\theta(\overline{p_1p_5}), \theta(\overline{p_4p_5})) = 0.322$. Thus, we know $\epsilon(\overline{p_1p_5}) = \max\{0.463, 0.785, 0.785, 0.322\} = 0.785$. Besides, we can compute $\epsilon(\overline{p_5p_8}) = 0.742$. Thus, we know $\epsilon(T') = \max\{\epsilon(\overline{p_1p_5}), \epsilon(\overline{p_5p_8})\} = \max\{0.785, 0.742\} = 0.785$.

In the following, when we write $\epsilon(\overline{p_ip_j})$ $(0 \leq i < j \leq n)$, we mean the simplification error of $\overline{p_ip_j}$ when it is used to approximate the segments between p_i and p_j in T.

Problem Statement of Min-Error. The Min-Error problem is to simplify a given trajectory such that the error of the simplified trajectory is the smallest and the size of the simplified trajectory is at most a given positive integer W called the *storage budget*. The formal definition is as follows.

PROBLEM 1 (MIN-ERROR). *Given a trajectory T and a positive integer W, **the Min-Error problem** is to find a simplification T' of T such that $|T'| \leq W$ and $\epsilon(T')$ is minimized.* □

To illustrate, consider the Min-Error problem with its input trajectory as T in Figure 1 and its input storage budget as 3. Then, $T' = (p_1, p_5, p_8)$ is the optimal solution since we cannot find any other simplification of T with its size at most 3 and its error smaller than $\epsilon(T')$ $(= 0.785)$.

We summarize the notations used in this paper in Table 2.

Notation	Description
$T = (p_1, p_2, ..., p_n)$	a trajectory
$T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$	a simplification of trajectory T
p_i	the i^{th} position of trajectory T
$\overline{p_h p_{h+1}}$	the h^{th} segment of trajectory T
$T[i : j]$	the portion of T from p_i to p_j
$\theta(\overline{p_i p_j})$	the direction of segment $\overline{p_i p_j}$
$\theta[i : j]$	the set containing $\theta(\overline{p_h p_{h+1}})$ for $h \in [i, j-1]$
$[\theta_1, \theta_2]$	the angular range formed by rotating a vector anti-clockwise from θ_1 to θ_2
$\triangle(\theta_1, \theta_2)$	the angular difference between θ_1 and θ_2
$\epsilon(\overline{p_{s_k} p_{s_{k+1}}})$	the simplification error of segment $\overline{p_{s_k} p_{s_{k+1}}}$
$\epsilon(T')$	the simplification error of T'
W	the storage budget
T'_o	the optimal solution of the Min-Error problem
ϵ_o	the error of T'_o
$\xi([\theta_1, \theta_2])$	the span of angular range $[\theta_1, \theta_2]$
$\mathcal{D}$	the set of the directions of all possible segments in T
$\mathcal{D}'$	a subset of $\mathcal{D}$
$mcar(\mathcal{D}')$	the maximum covering angular range of $\mathcal{D}'$
$\xi(T')$	the span of T'
T'_ξ	the optimal solution of the Min-Span problem
ξ_o	the span of T'_ξ
$L = (\theta_1, \theta_2, ..., \theta_{n-1})$	the sorted list of the directions in $\mathcal{D}$
Θ	a $(n-1) \times (n-1)$ matrix with $\Theta[i][j]$ defined in Equation (9)
$\mathcal{S}$	the multi-set of all entries in the matrix Θ
$\Theta[s : e][j]$	the array containing the values between the s^{th} position and the e^{th} position of the j^{th} column of Θ
$\mathcal{A}$	the set containing $\Theta[1 : j][j]$'s and $\Theta[j+1 : n-1][j]$'s for $j \in [1, n-1]$
$\mathcal{T}$	the index triplet set of $\mathcal{A}$ as defined in Equation (10)
$b(s, e, j)$	the bisector of $\Theta[s : e][j]$
$\mathcal{A}(\xi, -), \mathcal{A}(\xi, =), \mathcal{A}(\xi, +)$	groups of arrays in $\mathcal{A}$ with bisectors smaller than, equal to, and larger than ξ, respectively
$N(\mathcal{A}(\xi, -)), N(\mathcal{A}(\xi, =)), N(\mathcal{A}(\xi, +))$	numbers of arrays in $\mathcal{A}(\xi, -)$, $\mathcal{A}(\xi, =)$, and $\mathcal{A}(\xi, +)$, respectively
$\mathcal{B}$	the multi-set of the bisectors of all arrays in $\mathcal{A}$

Table 2: Summary of notations

3. EXACT ALGORITHMS

Given a simplification T' of T, we say that T' is *affordable* iff $|T'| \leq W$. Let T'_o be the optimal solution of the Min-Error problem and ϵ_o be the error of T'_o, i.e., $\epsilon_o = \epsilon(T'_o)$. Then, T'_o corresponds to one affordable simplification with the smallest error.

Let $\mathcal{F}$ be the set containing all affordable simplifications of T. A naive method for the Min-Error is to perform an exhaustive search over $\mathcal{F}$ and find the one with the smallest error, which, however, is not feasible since the size of $\mathcal{F}$ is exponential in terms of W (specifically, $|\mathcal{F}| = \binom{n-2}{W-2}$). A better way is to design a *dynamic programming* algorithm since we have the following *sub-problem optimality property*: If $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ is an optimal solution for the Min-Error problem instance with its input trajectory of T and its input storage budget of W, then $T'' = (p_{s_2}, ..., p_{s_m})$ is also an optimal solution for another Min-Error problem instance with its input trajectory of $T[s_2 : n]$ and its input storage budget of $W - 1$. We call this dynamic programming algorithm *DP*, and since there is not much surprise in the development of *DP*, we omit the details of *DP* here and put them in our technical report [25]. Unfortunately, *DP* has a time complexity of $O(Wn^3)$, which is prohibitively expensive on large datasets. Thus, in the following, we design a *binary search* algorithm called *Error-Search* for the Min-Error problem. *Error-Search* has a time complexity of $O(n^2 C \log n)$ (C is usually a small constant) which is significantly smaller than that of *DP*.

Let $\mathcal{E}$ be the set containing all $\epsilon(\overline{p_i p_j})$'s for $1 \leq i < j \leq n$, i.e., $\mathcal{E} = \{\epsilon(\overline{p_i p_j}) | 1 \leq i < j \leq n\}$. Note that $|\mathcal{E}| = O(n^2)$. We observe that the minimized error ϵ_o is contained in $\mathcal{E}$, i.e., $\epsilon_o \in \mathcal{E}$. This could be easily verified by the fact that any simplification has its error equal to the greatest simplification error of its segment, which is covered by $\mathcal{E}$ by definition.

Given a non-negative real value ϵ, we say that ϵ is an *affordable error* if there exists an affordable simplification T' in $\mathcal{F}$ such that $\epsilon(T') \leq \epsilon$. Thus, ϵ_o corresponds to the *smallest* affordable error.

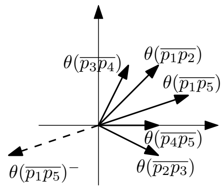


Figure 5: Opposite direction

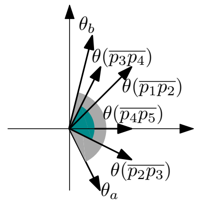


Figure 6: $mcar(\cdot)$

In view of the above discussion, we design an algorithm called *Error-Search* as follows. Firstly, we construct the search space $\mathcal{E}$. Secondly, for each $\epsilon \in \mathcal{E}$, we check whether there exists an affordable simplification T' with $\epsilon(T') \leq \epsilon$ (i.e., we check whether ϵ is an affordable error) which we call the *error affordability check on* ϵ. We note here that we can adopt a *binary search* strategy (instead of a linear scan strategy) for searching on $\mathcal{E}$ since we have the following *monotonicity property*: if ϵ is an affordable error, then any $\epsilon' > \epsilon$ is also an affordable error. Thirdly, we return as T'_o the affordable simplification corresponding to the smallest affordable error found (which is exactly ϵ_o).

The correctness of *Error-Search* is obvious. In the following, we discuss (1) how to construct the search space $\mathcal{E}$, (2) how to perform the error affordability check on a given ϵ, and (3) the time and space complexities of *Error-Search*.

(1) Construction of $\mathcal{E}$. Recall that $\mathcal{E} = \{\epsilon(\overline{p_i p_j}) | 1 \leq i < j \leq n\}$. Thus, we have $O(n^2)$ instances of $\epsilon(\overline{p_i p_j})$'s in $\mathcal{E}$. A straightforward method for computing $\epsilon(\overline{p_i p_j})$ ($1 \leq i < j \leq n$) is to compare $\theta(\overline{p_i p_j})$ with $\theta(\overline{p_h p_{h+1}})$ for each $h \in [i, j)$. This method, though simple, incurs the worst-case cost of $O(n)$. Thus, the overall cost of constructing $\mathcal{E}$ based on this method is $O(n^3)$, which is too costly. In the following, we develop a more efficient method for computing $\epsilon(\overline{p_i p_j})$ ($1 \leq i < j \leq n$) which runs in $O(\log n)$ time only instead of $O(n)$ time, resulting in the overall cost of constructing $\mathcal{E}$ being $O(n^2 \log n)$.

Our method is based on the concept of "opposite direction" which will be described in detail next. Recall that $\epsilon(\overline{p_i p_j})$ corresponds to the *greatest* angular difference between $\theta(\overline{p_i p_j})$ and a direction in $\theta[i : j]$. Thus, computing $\epsilon(\overline{p_i p_j})$ could be finished by finding the direction in $\theta[i : j]$ which has the *greatest* angular difference from $\theta(\overline{p_i p_j})$. Let θ^* denote this direction. With θ^*, we can easily compute $\epsilon(\overline{p_i p_j})$ by computing the angular difference between $\theta(\overline{p_i p_j})$ and θ^* with Equation (1). In the following, we focus on how to find θ^*.

Let $\theta(\overline{p_i p_j})^-$ be the *opposite direction* of $\theta(\overline{p_i p_j})$, i.e., $\theta(\overline{p_i p_j})^- = [(\theta(\overline{p_i p_j}) + \pi) \mod 2\pi]$. We observe that θ^* is exactly the direction in $\theta[i : j]$ which has the *smallest* angular difference from $\theta(\overline{p_i p_j})^-$. This is simply because any direction θ in $\theta[i : j]$ has its angular difference from $\theta(\overline{p_i p_j})$ equal to π minus its angular difference from $\theta(\overline{p_i p_j})^-$, i.e., $\triangle(\theta, \theta(\overline{p_i p_j})) = \pi - \triangle(\theta, \theta(\overline{p_i p_j})^-)$.

To illustrate, consider Figure 5. $\theta(\overline{p_1 p_2})$, $\theta(\overline{p_2 p_3})$, $\theta(\overline{p_3 p_4})$, and $\theta(\overline{p_4 p_5})$ correspond to $\theta[1 : 5]$. $\theta(\overline{p_1 p_5})$ and $\theta(\overline{p_1 p_5})^-$ are also shown. As could be verified, $\theta(\overline{p_2 p_3})$ is the direction in $\theta[1 : 5]$ which has the *greatest* angular difference from $\theta(\overline{p_1 p_5})$ and also the *smallest* angular difference from $\theta(\overline{p_1 p_5})^-$.

Thus, we propose to search θ^* with two steps. First, we sort the directions in $\theta[i : j]$ in ascending order and let $\theta_1, \theta_2, ..., \theta_{j-i}$ be the resulting sorted list (note that sorting from scratch incurs a cost of $O(n \log n)$ here, and what we do is to *incrementally* maintain the sorted list of $\theta[i : j]$ based on the one of $\theta[i : j-1]$ which has already been maintained for computing $\epsilon(\overline{p_i p_{j-1}})$ if we compute

$\epsilon(\overline{p_ip_{j-1}})$ first, and thus this step could be finished in $O(\log n)$ time). Second, we find the direction in the sorted list which has the smallest angular difference from $\theta(\overline{p_ip_j})^-$ (i.e., θ^*) and this step could also be done in $O(\log n)$ time with a binary search process based on the sorted list. In combination of the first step and the second step, our method finds θ^* in $O(\log n)$ time.

In view of the above discussion, we know that $\mathcal{E}$ could be constructed in $O(n^2 \log n)$ time since we have $O(n^2)$ instances of $\epsilon(\overline{p_ip_j})$ each with a computation cost of $O(\log n)$.

(2) Error Affordability Check on ϵ. Given a value ϵ, the task is to check whether there exists an affordable simplification T' in $\mathcal{F}$ with $\epsilon(T') \leq \epsilon$. A linear scan method over $\mathcal{F}$ is not feasible since the size of $\mathcal{F}$ is exponential. In the following, we propose a method which runs in $O(n^2C)$ time where C is usually a small constant.

LEMMA 1. *Let ϵ be a non-negative value and T' be a simplification of T with its error at most ϵ and its size minimized. Then, ϵ is an affordable error iff T' is affordable.* □

PROOF. "⇒": Suppose that ϵ is an affordable error, i.e., there exists an affordable simplification T'' with $\epsilon(T'') \leq \epsilon$. We have $|T'| \leq |T''| \leq W$, and thus we know that T' is affordable.

"⇐": Clearly, ϵ is an affordable error since T' is affordable and has its error at most ϵ by definition. □

Lemma 1 suggests that the error affordability check on a given value ϵ can be implemented with the following two steps. First, we compute the simplification T' of T with its error at most ϵ and its size minimized. This essentially corresponds to solving a *Min-Size* problem instance [24] with its input trajectory as T and its input error tolerance as ϵ. Thus, this step can be done by executing an exact algorithm of the Min-Size problem. Second, we check whether T' is affordable (i.e., the size of T' is at most W). If yes, then ϵ is an affordable span. Otherwise, it is not. Since the time complexity of the exact algorithm for the Min-Size problem in [24] is $O(n^2C)$ (where C is usually a small constant, e.g., $C = 1$ if $\epsilon \leq \pi/2$), we know that the time complexity of the above method of performing an error affordability check is also $O(n^2C)$.

(3) Time & Space Complexity of *Error-Search*. In conclusion, the time complexity of *Error-Search* is $O(n^2 C \log n)$ since the cost of constructing $\mathcal{E}$ is $O(n^2 \log n)$, the cost of sorting $\mathcal{E}$ (for binary search) is $O(n^2 \log n^2)$ $(= O(n^2 \log n))$, and the cost of performing the error affordability check $O(\log n^2)$ times (in the binary search) is $O(n^2C \cdot \log n^2)$ $(= O(n^2C \log n))$. The space complexity of *Error-Search* is $O(n^2)$ which corresponds to the space cost of storing the search space $\mathcal{E}$.

4. APPROXIMATE ALGORITHM

In this section, we present our approximate algorithm called *Span-Search* for the Min-Error problem which runs in $O(n \log^2 n)$ time and gives a 2-factor approximation. Specifically, in Section 4.1, we introduce an *estimator* of the error of a simplification of T called *span*. In Section 4.2, based on this estimator, we define a new problem called *Min-Span* whose optimal solution corresponds to a 2-factor approximation of the Min-Error problem. In Section 4.3, we give an overview of *Span-Search* which returns the optimal solution of the Min-Span problem in $O(n \log^2 n)$ time. In Section 4.4, we give the details of *Span-Search* and analyze its time and space complexities.

4.1 An Estimator of Error

We define the *span* of an angular range $[\theta_1, \theta_2]$, denoted by $\xi([\theta_1, \theta_2])$, to be equal to the angle of an *anti-clockwise* rotation from a vector with its direction equal to θ_1 to another vector with its direction equal to θ_2. Specifically, we have

$$\xi([\theta_1, \theta_2]) = \begin{cases} \theta_2 - \theta_1 & \text{if } \theta_2 \geq \theta_1 \\ 2\pi - (\theta_1 - \theta_2) & \text{if } \theta_2 < \theta_1 \end{cases} \tag{3}$$

Note that $\xi([\theta_1, \theta_2])$ is non-negative, and for any θ_1 and θ_2 in $[0, 2\pi)$, we have $\xi([\theta_1, \theta_2]) + \xi([\theta_2, \theta_1]) = 2\pi$.

To illustrate, consider Figure 3 where we have $\theta_1 = 5.820$ and $\theta_2 = 1.107$. Thus, we know $\xi([\theta_1, \theta_2]) = \xi([5.820, 1.107]) = 2\pi - (5.820 - 1.107) = 1.570$ and $\xi([\theta_2, \theta_1]) = \xi([1.107, 5.820]) = 5.820 - 1.107 = 4.713$.

Let $\mathcal{D}$ be the set of the directions of all possible segments in T, i.e., $\mathcal{D} = \theta[1:n]$. Note that $|\mathcal{D}| = n - 1$. Given a set $\mathcal{D}' \subseteq \mathcal{D}$, any angular range that covers *all* directions in $\mathcal{D}'$ is said to be a *covering angular range* of $\mathcal{D}'$. Among all covering angular ranges of $\mathcal{D}'$, the one with the *smallest* span is called the *minimum covering angular range* of $\mathcal{D}'$ which we denote by $mcar(\mathcal{D}')$.

To illustrate, consider Figure 6 where we show $\mathcal{D}' = \theta[1:5] = \{\theta(\overline{p_1p_2}), \theta(\overline{p_2p_3}), \theta(\overline{p_3p_4}), \theta(\overline{p_4p_5})\}$ and two other directions θ_a and θ_b. Then, $[\theta_a, \theta_b]$ (see the sector area in lighter color) is a covering angular range of $\mathcal{D}'$ since all directions in $\mathcal{D}'$ fall in $[\theta_a, \theta_b]$. Besides, the minimum covering angular range of $\mathcal{D}'$, i.e., $mcar(\mathcal{D}')$, is $[\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})]$ (see the sector area in darker color) since $[\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})]$ covers all directions in $\mathcal{D}'$ (i.e., $[\theta(\overline{p_3p_4}), \theta(\overline{p_5, p_6})]$ is a covering angular range of $\mathcal{D}'$) and there exists no other covering angular range of $\mathcal{D}'$ with its span smaller than that of $[\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})]$ $(= 1.570)$ (See Figure 3).

Note that the two boundaries of $mcar(\mathcal{D}')$ always come from $\mathcal{D}'$ since otherwise the range could be shrunk further and it does not have the minimum span.

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory and $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be a simplification of T. The *span* of T', denoted by $\xi(T')$, is defined to be the greatest span of the minimum covering angular ranges of $\theta[s_k : s_{k+1}]$ where $k \in [1, m)$, i.e.,

$$\xi(T') = \max_{1 \leq k < m}\{\xi(mcar(\theta[s_k : s_{k+1}]))\} \tag{4}$$

To illustrate, consider our running example in Figure 1. $T' = (p_1, p_5, p_8)$ is a simplification of T. As mentioned before, $mcar(\theta[1:5]) = [\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})] = [5.821, 1.107]$ and thus $\xi(mcar(\theta[1:5])) = \xi([5.821, 1.107]) = 1.570$. Besides, $mcar(\theta[5:8]) = [\theta(\overline{p_6p_7}), \theta(\overline{p_5p_6})] = [5.498, 0.464]$ and thus $\xi(mcar(\theta[5:8])) = \xi([5.498, 0.464]) = 1.249$. Therefore, $\xi(T') = \max\{\xi(mcar(\theta[1:5])), \xi(mcar(\theta[5:8]))\} = \max\{1.570, 1.249\} = 1.570$.

4.2 The Min-Span Problem

In this part, we define a problem called *Min-Span* which is quite similar to *Min-Error*, but with a different objective.

PROBLEM 2 (MIN-SPAN). *Given a trajectory T and a positive integer W, the **Min-Span** problem is to find a simplification T' of T such that $|T'| \leq W$ and $\xi(T')$ is minimized.* □

To illustrate, consider a Min-Span problem instance with its input trajectory as T in Figure 1 and its input W as 3. It could be verified that $T' = (p_1, p_5, p_8)$ is the optimal solution of this problem instance since we cannot find any other simplification of T which has its size at most 3 and its span smaller than $\xi(T')$ $(= 1.570)$.

Interestingly, the *optimal* solution of the Min-Span problem is a *2-factor approximation* of the Min-Error problem.

LEMMA 2. *Let T'_o be the optimal solution of the Min-Error problem with its input trajectory as T and its input storage budget as W. Let T'_ξ be the optimal solution of the Min-Span problem* *with its input trajectory and its input storage budget both the same as the Min-Error problem. Then, $\epsilon(T'_\xi) \le 2 \cdot \epsilon(T'_o)$.* □

PROOF. This proof is divided into two parts. In the first part, we show that any simplification $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ of T satisfies $\frac{\xi(T')}{\epsilon(T')} \in [1, 2]$ which we prove with two steps.

First, we show that for any $k \in [1, m)$, we have

$$\frac{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})}{\xi(mcar(\theta[s_k : s_{k+1}]))} \in [1/2, 1] \quad (5)$$

Suppose that $mcar(\theta[s_k : s_{k+1}])$ is $[\theta_a, \theta_b]$. Note that θ_a and θ_b are two directions in $\theta[s_k : s_{k+1}]$. We have two cases.

Case 1: $\xi([\theta_a, \theta_b]) \le \pi$. For illustration, consider Figure 7(a). In this case, $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is covered by $[\theta_a, \theta_b]$. Therefore, we have

$$\begin{aligned}\epsilon(\overline{p_{s_k} p_{s_{k+1}}}) &= \max\{\triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_a), \triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_b)\} \\ &\in [1/2, 1] \cdot (\triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_a) + \triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_b)) \\ &= [1/2, 1] \cdot \xi([\theta_a, \theta_b])\end{aligned}$$

Case 2: $\xi([\theta_a, \theta_b]) > \pi$. In this case, $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ could be or not be covered by $[\theta_a, \theta_b]$. We further consider two sub-cases.

Case 2(a): $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is covered by $[\theta_a, \theta_b]$. For illustration, consider Figure 7(b). The proof of this case is similar to the one of Case 1 and thus it is omitted here.

Case 2(b): $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is not covered by $[\theta_a, \theta_b]$. Then, $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is covered by $[\theta_b, \theta_a]$. For illustration, consider Figure 7(c). Let θ_c and θ_d be two directions in $\theta[s_k : s_{k+1}]$ such that θ_c and θ_d are in $[\theta_a, \theta_b]$ and no directions in $\theta[s_k : s_{k+1}]$ other than θ_c and θ_d are in $[\theta_c, \theta_d]$ (Note that θ_c and θ_d always exist). Then, we know that $[\theta_d, \theta_c]$ corresponds to a covering angular range of $\theta[s_k : s_{k+1}]$ and $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ falls in $[\theta_d, \theta_c]$. Besides, we know $\xi([\theta_d, \theta_c]) \ge \xi([\theta_a, \theta_b])$ since $[\theta_a, \theta_b]$ is the minimum covering angular range of $\theta[s_k : s_{k+1}]$. Similar to Case 1, we have $\frac{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})}{\xi([\theta_d, \theta_c])} \in [1/2, 1]$ which implies that $\epsilon(\overline{p_{s_k} p_{s_{k+1}}}) \ge \frac{1}{2} \cdot \xi([\theta_d, \theta_c]) \ge \frac{1}{2} \cdot \xi([\theta_a, \theta_b])$. Furthermore, we have $\epsilon(\overline{p_{s_k} p_{s_{k+1}}}) \le \pi < \xi([\theta_a, \theta_b])$. In combination, we have $\frac{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})}{\xi([\theta_a, \theta_b])} \in [\frac{1}{2}, 1]$.

Second, Let $k' = \arg\max_{k \in [1,m)}\{\xi(mcar(\theta[s_k : s_{k+1}]))\}$ and $k'' = \arg\max_{k \in [1,m)}\{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})\}$. By using Equation (5), we have

$$\begin{aligned}\xi(T') = \xi(mcar(\theta[s_{k'} : s_{k'+1}])) &\le 2 \cdot \epsilon(\overline{p_{s_{k'}} p_{s_{k'+1}}}) \\ &\le 2 \cdot \epsilon(\overline{p_{s_{k''}} p_{s_{k''+1}}}) = 2 \cdot \epsilon(T')\end{aligned} \quad (6)$$

and

$$\begin{aligned}\xi(T') = \xi(mcar(\theta[s_{k'} : s_{k'+1}])) &\ge \xi(mcar(\theta[s_{k''} : s_{k''+1}])) \\ &\ge \epsilon(\overline{p_{s_{k''}} p_{s_{k''+1}}}) = \epsilon(T')\end{aligned} \quad (7)$$

By using Equations (6) and (7), we obtain $\frac{\xi(T')}{\epsilon(T')} \in [1, 2]$.

In the second part, we show that $\epsilon(T'_\xi) \le 2 \cdot \epsilon(T'_o)$ as follows.

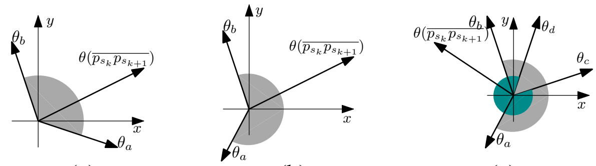
$$\epsilon(T'_\xi) \le \xi(T'_\xi) \le \xi(T'_o) \le 2 \cdot \epsilon(T'_o)$$

□

4.3 Overview of Span-Search

In this part, we develop an algorithm called *Span-Search* which returns the optimal solution of the Min-Span problem in $O(n \log^2 n)$ time and thus gives a 2-factor approximation for the Min-Error problem (Lemma 2).

Let T'_ξ be the optimal solution of the Min-Span problem and ξ_o be the span of T'_ξ. Essentially, T'_ξ corresponds to the affordable simplification with the smallest span. *Span-Search* first maintains a search space $\mathcal{S}$ containing ξ_o and then searches ξ_o over $\mathcal{S}$ (T'_ξ can also be retrieved when ξ_o is found).



(a) (b) (c)

Figure 7: Proof of Lemma 2

4.3.1 Concepts & Search Space $\mathcal{S}$

We introduce some concepts used for defining the search space $\mathcal{S}$ and then give a precise definition of $\mathcal{S}$. Suppose that T'_ξ is $(p_{s_1}, p_{s_2}, ..., p_{s_m})$. From Equation (4), we know $\xi(T'_\xi) = \max_{1 \le k < m}\{\xi(mcar(\theta[s_k : s_{k+1}]))\}$. Let $k^* = \arg\max_{1 \le k < m}\{\xi(mcar(\theta[s_k : s_{k+1}]))\}$. Then, we have $\xi_o = \xi(T'_\xi) = \xi(mcar(\theta[s_{k^*} : s_{k^*+1}]))$. Consider $\xi(mcar(\theta[s_{k^*} : s_{k^*+1}]))$. Let $[\theta, \theta']$ be $mcar(\theta[s_{k^*} : s_{k^*+1}])$. Note that θ and θ' are two directions in $\theta[s_{k^*} : s_{k^*+1}]$. Then, we derive that $\xi_o = \xi([\theta, \theta'])$). By Equation (3), we have

$$\xi_o = \begin{cases} \theta' - \theta & \text{if } \theta' \ge \theta \\ 2\pi - (\theta - \theta') & \text{if } \theta' < \theta \end{cases} \quad (8)$$

Essentially, θ and θ' could be the directions of any two segments of T. Thus, we have the following observation.

OBSERVATION 1 (PAIRWISE DIRECTION DIFFERENCE). *Let ξ_o be the optimal span of the Min-Span problem. There exist two segments of T such that ξ_o is equal to either $\theta' - \theta$ or $2\pi - (\theta - \theta')$ where θ and θ' are the directions of the two segments.* □

Based on the above observation, we construct an $(n-1) \times (n-1)$ matrix Θ containing both $\theta' - \theta$ and $2\pi - (\theta - \theta')$ for each possible pair $(\theta, \theta') \in \mathcal{D} \times \mathcal{D}$, where $\mathcal{D}$ is the set of the directions of all possible segments of T, and define the search space $\mathcal{S}$ to be the multi-set of all values in the matrix Θ. Note that the size of $\mathcal{S}$ is $(n-1)^2 = O(n^2)$. Specifically, Θ is defined as follows. Let $L = (\theta_1, \theta_2, ..., \theta_{n-1})$ be the sorted list of the values in $\mathcal{D}$ in *ascending* order. For each $i \in [1, n-1]$ and each $j \in [1, n-1]$, we define

$$\Theta[i][j] = \begin{cases} \theta_j - \theta_i & \text{if } j \ge i \\ 2\pi - (\theta_i - \theta_j) & \text{if } j < i \end{cases} \quad (9)$$

To illustrate, consider Table 3 which shows the sorted list of $\mathcal{D}$ of the trajectory T in Figure 1 and Table 4 which shows the corresponding matrix Θ.

With Observation 1, it is easy to verify that ξ_o is in $\mathcal{S}$. We present this result in the following lemma.

LEMMA 3. *The span of the optimal solution of the Min-Span problem (i.e., ξ_o) is in $\mathcal{S}$.* □

For example, as mentioned before, for the Min-Span problem with its input trajectory as T presented in Figure 1 and its input W as 3, ξ_o is equal to 1.570 which corresponds to $\Theta[6][4]$.

Given a value ξ, we say that ξ is an *affordable span* iff there exists an affordable simplification of T with its span at most ξ. It immediately follows that ξ_o corresponds to the *smallest* affordable span. With Lemma 3, we know that ξ_o is the smallest affordable span in $\mathcal{S}$.

	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7
L:	0	0.464	0.785	1.107	5.498	5.820	5.961

Table 3: Sorted list of the directions in $\theta[1:8]$

0	0.464	0.785	1.107	5.498	5.820	5.961
5.819	0	0.321	0.643	5.034	5.356	5.497
5.498	5.962	0	0.322	4.713	5.035	5.176
5.176	5.640	5.961	0	4.391	4.713	4.854
0.785	1.249	1.570	1.892	0	0.322	0.463
0.463	0.927	1.248	1.570	5.961	0	0.141
0.322	0.786	1.107	1.429	5.820	6.142	0

Table 4: Matrix Θ defined by Equation (9)

$\Theta[1:1][1]$	$\Theta[1:2][2]$	$\Theta[1:3][3]$	$\Theta[1:4][4]$	$\Theta[1:5][5]$	$\Theta[1:6][6]$	$\Theta[1:7][7]$
$\Theta[2:7][1]$	$\Theta[3:7][2]$	$\Theta[4:7][3]$	$\Theta[5:7][4]$	$\Theta[6:7][5]$	$\Theta[7:7][6]$	

Table 5: The array set representing matrix Θ

(1,1,1), (1,2,2), (1,3,3), (1,4,4), (1,5,5), (1,6,6), (1,7,7)
(2,7,1), (3,7,2), (4,7,3), (5,7,4), (6,7,5), (7,7,6)

Table 6: The index triplet set (for the original search space $\mathcal{S}$)

~~(1,**0**,1)~~, (1,**1**,2), (1,**1**,3), (1,**2**,4), (1,5,5), (1,6,6), (1,7,7)
(2,**4**,1), (3,**4**,2), (4,**5**,3), (5,7,4), (6,7,5), (7,7,6)

Table 7: The index triplet set (for the updated search space resulted from the pruning based on pivot $\xi = 1.249$)

4.3.2 *Strategy of Searching over $\mathcal{S}$*

After we introduced the concepts and defined the search space $\mathcal{S}$ in the previous section, in this section, we present a strategy called *Span-Search* for finding the optimal span ξ_o on $\mathcal{S}$. Given a value ξ, we call the procedure of checking whether ξ is an affordable span the *span affordability check on ξ*. This procedure, when called with an input of ξ, also returns an affordable simplification T' with $\xi(T') \leq \xi$ if ξ is an affordable span.

As we described before, we know that ξ_o is the smallest affordable span in $\mathcal{S}$. Thus, we propose to find ξ_o with three steps.

- **Step 1 (Searching Step):** Step 1 is to find a value ξ from $\mathcal{S}$ and perform a span affordability check on ξ. If ξ is an affordable span, it also obtains an affordable simplification T'. Let ξ_{best} be a variable denoting the best-known affordable span in $\mathcal{S}$ (i.e., the smallest affordable span in $\mathcal{S}$ seen so far), initialized to ∞. Let T'_{best} be a variable denoting the simplified trajectory with its span at most ξ_{best}. If ξ is an affordable span and $\xi < \xi_{best}$, it updates ξ_{best} and T'_{best} with ξ and T', respectively.
- **Step 2 (Iterative Step):** Step 2 is to perform Step 1 iteratively with one of the remaining values in $\mathcal{S}$ to be found until there is no remaining value in $\mathcal{S}$.
- **Step 3 (Output Step):** Step 3 is to return ξ_{best} and T'_{best}.

A simple strategy of implementing Step 1 called *Random-Search* is to select a *random* value from $\mathcal{S}$ as ξ. The algorithm with this strategy is too costly since $\mathcal{S}$ involves $O(n^2)$ values and thus the algorithm needs to perform $O(n^2)$ span affordability checks.

Another strategy of implementing Step 1 called *Binary-Search* is to always select the *median* of the values in the current search space as ξ since the result of the span affordability check on the median could be used to prune at least half of the current search space due to the following *monotonicity* property.

PROPERTY 1 (MONOTONICITY). *Let ξ and ξ' be two real numbers where $\xi < \xi'$. If ξ is an affordable span, then ξ' is also an affordable span.* □

Specifically, if ξ is an affordable span, we can prune all values at least ξ in the current search space; otherwise, we can prune all values at most ξ in the current search space. Although the algorithm with the *Binary-Search* strategy performs $2\log n = O(\log n)$ span affordability checks only, it is still not scalable (since it needs to materialize a search space $\mathcal{S}$ which occupies $O(n^2)$ space) and too costly (since it introduces extra cost for finding the medians which takes $O(n^2 \log n)$ time[1]).

Algorithm 1 *Span-Search*

Initialize the *index triplet set* $\mathcal{T}$ of $\mathcal{S}$ (Section 4.4.1)
//Steps 1 & 2
while there exist values in the current search space represented by $\mathcal{T}$ **do**
  Find a *pivot* ξ wrt the current search space (Section 4.4.2)
  Perform a span affordability check on ξ (Section 4.4.3)
  Update ξ_{best} and T'_{best} if necessary
  Prune the search space with ξ by updating $\mathcal{T}$ (Section 4.4.4)
//Step 3
return ξ_{best} and T'_{best}

In this paper, we propose a new strategy called *Span-Search*. which differs from *Random-Search* and *Binary-Search* as follows.

- *Span-Search* does not materialize the search space $\mathcal{S}$ *explicitly* as *Linear-Search* and *Binary-Search* do, instead, it materializes a concise representation of $\mathcal{S}$ called *index triplet set* (the details will be introduced in Section 4.4.1) which occupies $O(n)$ space only.
- *Span-Search* performs the span affordability check always on a *pivot* wrt the current search space (the details will be introduced in Section 4.4.2) at Step 1, which is different from *Random-Search* (on an *random* value from the current search space) or *Binary-Search* (on the *median* of the current search space). The details of how to perform a span affordability check on a given value will be introduced in Section 4.4.3.
- *Span-Search* prunes at least $\frac{1}{4}$ of the current search space after each span affordability check (whose details will be introduced in Section 4.4.4). This implies that *Span-Search* needs to perform $O(\log n)$ span affordability checks only.
- *Span-Search* has the time complexity of $O(n \log^2 n)$ and the space complexity of $O(n)$ both superior over those of *Linear-Search* and *Binary-Search* (the details will be discussed in Section 4.4.5).

The pseudo-code of *Span-Search* is given in Algorithm 1.

4.4 Details and Time Complexity Analysis of Span-Search

In this section, we give the details of *Span-Search*.

4.4.1 *Concise Representation of $\mathcal{S}$*

In this part, we introduce our *index triplet set* which can concisely represent the search space $\mathcal{S}$ with $O(n)$ space (note that a full materialization of $\mathcal{S}$ occupies $O(n^2)$ space).

We introduce some related concepts first. Given an l-sized array X and two integers $i, i' \in [1, l]$, if $i < i'$, then $X[i]$ is said to be *before* the position of $X[i']$ in the array X, and $X[i']$ is said to be *after* the position of $X[i]$ in the array X. If $i = i'$, $X[i]$ is said to be *at* the position of $X[i']$ in the array X.

Since $\mathcal{S}$ is the multi-set containing all values in matrix Θ, we focus on describing how to represent Θ concisely.

For each s, e and $j \in [1, n-1]$ where $s \leq e$, we denote by $\Theta[s:e][j]$ the array containing the values between the s^{th} position

[1]One can either sort the values in $\mathcal{S}$ at the right beginning with $O(n^2 \log n^2) = O(n^2 \log n)$ time and then pick the medians each with $O(1)$ time afterwards or run a *median selection* algorithm [3] which returns the median of N values with $O(N)$ time whenever a median is required. Both take $O(n^2 \log n)$ time.

and the e^{th} position in the j^{th} column of Θ, i.e., $\Theta[s : e][j] = \{\Theta[s][j], \Theta[s+1][j], ..., \Theta[e][j]\}$.

For each column of Θ, say, $\Theta[1 : n-1][j]$ where $j \in [1, n-1]$, which itself is an array, we maintain it with two arrays, namely $\Theta[s_1 : e_1][j]$ and $\Theta[s_2 : e_2][j]$, where $s_1 = 1$, $e_1 = j$, $s_2 = j+1$, and $e_2 = n-1$ (note that the $(n-1)^{th}$ column corresponds to one array (i.e., $\Theta[1 : n-1][n-1]$) only). As a result, the values in Θ are organized with $2(n-1)-1$ $(= O(n))$ arrays each in the form of $\Theta[s : e][j]$. Let $\mathcal{A}$ be the set containing all these arrays. Thus, the size of $\mathcal{A}$ is $O(n)$. Note that the multi-set of values of the arrays in $\mathcal{A}$ is exactly equal to $\mathcal{S}$. In the following, for clarity, when we write $\mathcal{A}$, we mean the array set corresponding to the matrix Θ of the search space $\mathcal{S}$.

To illustrate, consider Table 4 where each column is divided into two arrays: one with white background and the other one with gray background. Table 5 shows the corresponding array set $\mathcal{A}$.

A nice feature about $\mathcal{A}$ is that all arrays in $\mathcal{A}$ are *non-increasing* [2] which could be verified easily by using Equation (9) and the fact that for each $i, i' \in [1, n-1]$ where $i \leq i'$, we have $\theta_i \leq \theta_{i'}$.

PROPERTY 2 (NON-INCREASING ARRAYS). *Each array in $\mathcal{A}$ is non-increasing.* □

To illustrate, consider the arrays in Table 5 and their corresponding content shown in Table 4. It could be easily noticed that all the arrays are non-increasing.

We have introduced the concepts used to define the index triplet set. Let $\mathcal{S}$ be the search space and $\mathcal{A}$ be the corresponding array set. The *index triplet set* of $\mathcal{S}$, denoted by $\mathcal{T}$, is the set containing triplets of the indices of all arrays in $\mathcal{A}$. That is,

$$\mathcal{T} = \{(s, e, j) | \Theta[s : e][j] \in \mathcal{A}\} \tag{10}$$

Note that each triplet in $\mathcal{T}$ identifies an array in $\mathcal{A}$ concisely. The space complexity of $\mathcal{T}$ is $O(n)$ only since we have $O(n)$ arrays in $\mathcal{A}$ each with its space cost of $O(1)$ in $\mathcal{T}$. For example, Table 6 shows the index triplet set corresponding the array set shown in Table 5.

Interestingly, $\mathcal{T}$ alone concisely represents the multi-set of values of the arrays in $\mathcal{A}$ (or the search space $\mathcal{S}$). This is because for each triplet $(s, e, j) \in \mathcal{T}$, we know that *conceptually*, we have $\Theta[i][j]$ where $i = s, s+1, ..., e$. We do not materialize the content of $\Theta[i][j]$ explicitly since given the indices (i.e., i and j), the content can be retrieved in $O(1)$ time by using Equation (9). Instead, we materialize $\mathcal{T}$ only. In the following, for the sake of convenience, we refer $\mathcal{A}$ instead of $\mathcal{T}$ to represent the entire search space $\mathcal{S}$ (though $\mathcal{T}$ is the materialized version for $\mathcal{A}$).

4.4.2 Definition, Search Space & Retrieval of a Pivot

In this part, we answer three questions: (1) *what is a pivot*, (2) *where can we find a pivot* and (3) *how to find a pivot*.

(1) What is a pivot? Before we define what is a *pivot*, we introduce a concept called *bisector* and its related concepts.

For each array $\Theta[s : e][j]$ in $\mathcal{A}$, we define its *bisector*, denoted by $b(s, e, j)$, to be $\Theta[\lceil \frac{s+e}{2} \rceil][j]$. Since $\Theta[s : e][j]$ is non-increasing, we know that at least half of the values in $\Theta[s : e][j]$ are at most its bisector $b(s, e, j)$ and at least half of the values in $\Theta[s : e][j]$ are at least its bisector $b(s, e, j)$. For example, the bisector of array $\Theta[3 : 7][2]$ is $\Theta[5][2]$ which is equal to 1.249 (See Table 4).

For a given value $\xi \in \mathcal{S}$, the arrays in $\mathcal{A}$ could be categorized into three disjoint groups, namely the group containing those arrays with the bisectors strictly *smaller than* ξ which we denote by $\mathcal{A}(\xi, -)$, the group containing those arrays with the bisectors exactly *equal to* ξ which we denote by $\mathcal{A}(\xi, =)$, and the group containing those arrays with the bisectors strictly *larger than* ξ which we denote by $\mathcal{A}(\xi, +)$. Let $N(\mathcal{A}(\xi, -))$, $N(\mathcal{A}(\xi, =))$, and $N(\mathcal{A}(\xi, +))$ be the size of the multi-set of the values of the arrays in $\mathcal{A}(\xi, -)$, $\mathcal{A}(\xi, =)$, and $\mathcal{A}(\xi, +)$, respectively. Note that $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) + N(\mathcal{A}(\xi, +)) = |\mathcal{S}|$.

To illustrate, consider the array set $\mathcal{A}$ shown in Table 5. Suppose $\xi = 1.249$. Then, we know $\mathcal{A}(\xi, -) = \{\Theta[1 : 1][1], \Theta[2 : 7][1], \Theta[1 : 2][2], \Theta[1 : 3][3], \Theta[4 : 7][3], \Theta[1 : 4][4]\}$, $\mathcal{A}(\xi, =) = \{\Theta[3 : 7][2]\}$, and $\mathcal{A}(\xi, +) = \{\Theta[5 : 7][4], \Theta[1 : 5][5], \Theta[6 : 7][5], \Theta[1 : 6][6], \Theta[7 : 7][6], \Theta[1 : 7][7]\}$. As a result, we have $N(\mathcal{A}(\xi, -)) = 20$, $N(\mathcal{A}(\xi, =)) = 5$, and $N(\mathcal{A}(\xi, +)) = 24$. Note that $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) + N(\mathcal{A}(\xi, +)) = 49 = |\mathcal{S}|$.

Now, we are ready to define what is a *pivot*.

DEFINITION 1 (PIVOT). *Given a value $\xi \in \mathcal{S}$, ξ is defined to be a* pivot *wrt $\mathcal{S}$ if* $\min\{N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)), N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =))\} \geq \frac{|\mathcal{S}|}{2}$. □

For example, $\xi = 1.249$ corresponds to a pivot wrt $\mathcal{S}$ since $\min\{N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)), N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =))\} = \min\{20 + 5, 24 + 5\} = 25 \geq \frac{|\mathcal{S}|}{2}$ $(= 49/2)$.

In the following, we simply write "a pivot wrt $\mathcal{S}$" as "a pivot" if the context of $\mathcal{S}$ is clear.

(2) Where can we find a pivot? Before we give the details, we introduce a property first.

PROPERTY 3. *Given $\xi, \xi' \in \mathcal{B}$ with $\xi < \xi'$, we have*

$$N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) \leq N(\mathcal{A}(\xi', -)) + N(\mathcal{A}(\xi', =))$$
$$N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =)) \geq N(\mathcal{A}(\xi', +)) + N(\mathcal{A}(\xi', =))$$

This essentially says that $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =))$ is non-decreasing *while $N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =))$ is* non-increasing *when ξ increases.* □

PROOF. This is simply because $\mathcal{A}(\xi, -) \cup \mathcal{A}(\xi, =) \subseteq \mathcal{A}(\xi', -)$ and $\mathcal{A}(\xi', +) \cup \mathcal{A}(\xi', =) \subseteq \mathcal{A}(\xi, +)$. □

Let $\mathcal{B}$ be the multi-set containing the bisectors of all arrays in $\mathcal{A}$, i.e., $\mathcal{B} = \{\Theta[\lceil \frac{s+e}{2} \rceil][j] | \Theta[s : e][j] \in \mathcal{A}\}$. Note that the size of $\mathcal{B}$ is $O(n)$, and $\mathcal{B} \subseteq \mathcal{S}$. We claim that there exists a pivot in $\mathcal{B}$.

LEMMA 4. *At least one of the values in $\mathcal{B}$ is a pivot wrt $\mathcal{S}$.* □

PROOF. Let $\xi_1, \xi_2, ..., \xi_{|\mathcal{B}|}$ be the sorted list of $\mathcal{B}$ in ascending order. We prove Lemma 4 by contradiction. Assume that none of the values in $\mathcal{B}$ is a pivot.

Consider ξ_1. Clearly, $N(\mathcal{A}(\xi_1, -)) = 0$ and thus $N(\mathcal{A}(\xi_1, =)) + N(\mathcal{A}(\xi_1, +)) = |\mathcal{S}|$. Therefore, we know $N(\mathcal{A}(\xi_1, -)) + N(\mathcal{A}(\xi_1, =)) < |\mathcal{S}|/2$ since otherwise ξ_1 is a pivot which leads to a contradiction.

Consider $\xi_{|\mathcal{B}|}$. Similarly, we know $N(\mathcal{A}(\xi_{|\mathcal{B}|}, +)) = 0$ and thus $N(\mathcal{A}(\xi_{|\mathcal{B}|}, -)) + N(\mathcal{A}(\xi_{|\mathcal{B}|}, =)) = |\mathcal{S}|$. Therefore, we know $N(\mathcal{A}(\xi_{|\mathcal{B}|}, +)) + N(\mathcal{A}(\xi_{|\mathcal{B}|}, =)) < |\mathcal{S}|/2$ since otherwise $\xi_{|\mathcal{B}|}$ is a pivot wrt $\mathcal{S}$ which leads to a contradiction.

By using Property 3 and the above results, we know there exists $h_1 \in [1, |\mathcal{B}|)$ such that $N(\mathcal{A}(\xi_{h_1}, -)) + N(\mathcal{A}(\xi_{h_1}, =)) < |\mathcal{S}|/2$ and $N(\mathcal{A}(\xi_{h_1+1}, -)) + N(\mathcal{A}(\xi_{h_1+1}, =)) \geq |\mathcal{S}|/2$. Similarly, there exists $h_2 \in (1, |\mathcal{B}|]$ such that $N(\mathcal{A}(\xi_{h_2}, +)) + N(\mathcal{A}(\xi_{h_2}, =)) < |\mathcal{S}|/2$ and $N(\mathcal{A}(\xi_{h_2-1}, +)) + N(\mathcal{A}(\xi_{h_2-1}, =)) \geq |\mathcal{S}|/2$.

We consider 3 cases. Case 1: $h_2 < h_1 + 1$. We have $N(\mathcal{A}(\xi_{h_2}, -)) + 2 \cdot N(\mathcal{A}(\xi_{h_2}, =)) + N(\mathcal{A}(\xi_{h_2}, +)) < |\mathcal{S}|/2 + |\mathcal{S}|/2 = |\mathcal{S}|$ which leads to a contradiction. Case 2: $h_2 = h_1 + 1$.

[2] Given an l-sized array X where l is a positive integer, X is said to be *non-increasing* if for each $i, i' \in [1, l]$ where $i < i'$, $X[i] \geq X[i']$.

This contradicts the fact that $N(\mathcal{A}(\xi_{h_1}, -)) + N(\mathcal{A}(\xi_{h_1}, =)) + N(\mathcal{A}(\xi_{h_2}, =)) + N(\mathcal{A}(\xi_{h_2}, +)) = |\mathcal{S}|$. Case 3: $h_2 > h_1 + 1$. We deduce that ξ_{h_1+1} is a pivot which leads to a contradiction. That is, we deduce a contradiction in all cases which finishes our proof. □

To illustrate, consider the search space $\mathcal{S}$ corresponding to the array set shown in Table 5. We can compute $\mathcal{B} = \{0, 0.785, 0, 1.249, 0.321, 1.248, 0.322, 1.570, 4.713, 5.820, 4.713, 6.142, 4.854\}$. As mentioned before, 1.249 is a pivot wrt $\mathcal{S}$ which is contained in $\mathcal{B}$.

Lemma 4 is very usefully since it not only implies that there always exists a pivot, but also implies that we can focus on $\mathcal{B}$ which has its size of $O(n)$ for finding a pivot.

(3) How to find a pivot? According to Lemma 4, we can focus on $\mathcal{B}$ for finding a pivot. A straightforward method is to traverse the values in $\mathcal{B}$ one by one, check whether it is a pivot, and stop when a pivot is found. Note that given a value ξ, the cost of checking whether ξ is a pivot or not is $O(n)$ since we have $O(n)$ arrays in $\mathcal{A}$ and the number of values in an array $\Theta[s:e][j]$ is simply $e - s + 1$ which could be computed in $O(1)$ time.

Fortunately, we can find a pivot in a smarter way with a binary search over $\mathcal{B}$ based on the monotonicity properties shown in Property 3. Specifically, we first sort the values in $\mathcal{B}$ in ascending order and obtain a sorted list. Let b_m be the value at the middle of the list. Then, we compute $N(\mathcal{A}(b_m, -))$, $N(\mathcal{A}(b_m, =))$, and $N(\mathcal{A}(b_m, +))$. If $\min\{N(\mathcal{A}(b_m, -)) + N(\mathcal{A}(b_m, =)), N(\mathcal{A}(b_m, +)) + N(\mathcal{A}(b_m, =))\} \geq \frac{|\mathcal{S}|}{2}$, we return b_m as a pivot; otherwise, we have two cases.

- Case 1: $N(\mathcal{A}(b_m, -)) + N(\mathcal{A}(b_m, =)) < \frac{|\mathcal{S}|}{2}$. In this case, we can safely prune all values that are *at most* b_m in $\mathcal{B}$ (Here, pruning a value means that we ignore this value for finding a pivot, which is considered in this section, but this value is still in the current search space $\mathcal{S}$ (or $\mathcal{A}$)).
- Case 2: $N(\mathcal{A}(b_m, +)) + N(\mathcal{A}(b_m, =)) < \frac{|\mathcal{S}|}{2}$. In this case, we can safely prune all values that are *at least* b_m in $\mathcal{B}$.

In conclusion, if b_m is a pivot, we are done, and otherwise we can prune at least half of the search space $\mathcal{B}$ and repeat the process based on the remaining search space until we find a pivot.

The time complexity of the above method is simply $O(n \log n)$ since the sorting procedure has the cost of $O(n \log n)$ and the binary search procedure has $O(\log n)$ iterations each has the cost of $O(n)$ for checking whether a given value is a pivot.

4.4.3 Span Affordability Check on ξ

In this part, we introduce our method for performing the span affordability check on a given ξ.

Let ξ be a non-negative value. Given a simplification T' of T, we say that T' is a ξ-simplification (of T) iff $\xi(T') \leq \xi$.

Similar to the error affordability check described in Section 3, we perform the span affordability check on a given ξ as follows. First, we compute the ξ-simplification with the smallest size, say, T'. Then, we compare $|T'|$ with W. If $|T'| \leq W$, we conclude that ξ is an affordable span; otherwise, we conclude that ξ is not an affordable span. The correctness of this method is obvious and the remaining issue is how to find the ξ-simplification with the smallest size for a given ξ.

We design our algorithm as follows. Let i be the position index of T where the algorithm starts at. Initially, i is set to 1 and p_i is appended to T'. It tries to approximate as many consecutive segments starting from p_i in T as possible while adhering to the constraint that the span of the minimum covering angular range of the set containing the directions of these segments is at most ξ. To do it, it checks the position index j starting from $i + 1$ one by one. If $\xi(mcar(\theta[i:j])) \leq \xi$, it continues to check the next position index by updating j to $j + 1$ until either $j > n$ (i.e., $j = n + 1$) or $\xi(mcar(\theta[i:j])) > \xi$. Then, it appends p_{j-1} to T' since in either the case of $j > n$ (i.e., $j = n + 1$), or the case of $\xi(mcar(\theta[i:j])) > \xi$, the segments between p_i and p_{j-1} form the longest possible sequence starting from p_i that could be approximated by one segment in T'. After that, it continues the process from p_{j-1} by updating i with $j - 1$. It stops if $j > n$ which implies $j = n + 1$. The pseudo-code of the algorithm is shown in Algorithm 2.

Algorithm 2 Finding ξ-simplification with the smallest size

$T' \leftarrow (p_1)$
$i \leftarrow 1$; $j \leftarrow i + 1$
while $j \leq n$ **do**
  while $j \leq n$ and $\xi(mcar(\theta[i:j])) \leq \xi$ **do**
    $j \leftarrow j + 1$
  Append p_{j-1} to T'
  $i \leftarrow j - 1$
return T'

We illustrate Algorithm 2 with the input trajectory as T in Figure 1 and ξ as 1.249. Note that $\xi = 1.249$ is a pivot. In this case, $n = 8$. T' is first initialized as (p_1) and $i = 1$. It starts from $j = i+1 = 2$. It computes $\xi(mcar(\theta[1:2])) = 0$ since $\theta[1:2] = \{\theta(\overline{p_1p_2})\} = \{0.785\}$ and thus $mcar(\theta[1:2]) = [0.785, 0.785]$. Since $j \leq n$ and $\xi(mcar(\theta[1:2])) \leq \xi = 1.249$, it updates j to be $j + 1 = 3$. Again, it computes $\xi(mcar(\theta[1:3])) = 1.248$. Since $j \leq n$ and $\xi(mcar(\theta[1:3])) \leq \xi = 1.249$, it updates j to be $j + 1 = 4$. Then, it computes $\xi(mcar(\theta[1:4])) = 1.570$. This time, since $\xi(mcar(\theta[1:4])) > \xi = 1.249$, it stops updating j, but appends p_{j-1} (i.e., p_3) to T' (thus T' becomes (p_1, p_3)) and updates i to be $j - 1 = 3$. It implies that (p_1, p_2, p_3) is the longest possible sequence starting from p_1 which has the span at most $\xi = 1.249$. It repeats the same process with the new starting position p_3 and keeps increasing j by 1 until $j = 7$ since $\xi(\theta[3:7]) = 1.892 > \xi = 1.249$. Then, it appends p_{j-1} (i.e., p_6) to T' (thus T' becomes (p_1, p_3, p_6)) and updates i to be $j - 1 = 6$. It continues the same process with the new starting position p_6 and keeps increasing j by 1 until $j = 9$ since $j > n$. Then, it appends p_{j-1} (i.e., p_8) to T' (thus T' becomes (p_1, p_3, p_6, p_8)) and stops the process. At the end, it returns T' which is (p_1, p_3, p_6, p_8).

LEMMA 5. *Algorithm 2 finds the ξ-simplification with the smallest size for a given ξ.* □

PROOF. Let $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be the simplification returned by Algorithm 2. Let $T'' = (p_{t_1}, p_{t_2}, ..., p_{t_l})$ be the ξ-simplification with the smallest size. By definition, we have $s_1 = r_1 = 1$ and $s_m = t_l = n$. Note that $|T'| = m$ and $|T''| = l$.

Assume that $m > l$. We prove that for each $k \in [1, l]$, we have $s_k \geq r_k$ by deduction.

Base step: $k = 1$. We have $s_k = r_k = 1$.

Deduction step: $k > 1$. Assume that we have $s_{k-1} \geq r_{k-1}$. According to Algorithm 2, we have $\xi(mcar(\theta[s_{k-1}:j])) \leq \xi$ for $j \in [s_{k-1} + 1, s_k]$ while $\xi(mcar(\theta[s_{k-1} : s_k + 1])) > \xi$. Since $s_{k-1} \geq r_{k-1}$, we know $r_k \leq s_k$ since otherwise $\xi(mcar(\theta[r_{k-1} : r_k])) \geq \xi(mcar(\theta[r_{k-1} : s_k + 1)) \geq \xi(mcar(\theta[s_{k-1} : s_k + 1])) > \xi$, which leads to a contradiction. The above inequalities are based on the fact that $\xi(mcar(\mathcal{D}))$ is non-decreasing when the set $\mathcal{D}$ includes more directions.

Therefore, we have $s_l \geq r_l = n$, which leads to a contradiction that $s_l < s_m = n$. This finishes our proof. □

Algorithm 2 has the time complexity of $O(n \log n)$, whose implementation details and time complexity analysis could be found in our technical report [25].

Recall that the span affordability check on a given ξ is performed by first finding the ξ-simplification T' with the smallest size via Algorithm 2 and then comparing $|T'|$ with W. Thus, the cost of performing the span affordability check is dominated by the cost of Algorithm 2, which is $O(n \log n)$.

To illustrate, consider the span affordability check on $\xi = 1.249$ with the input trajectory as T in Figure 1 and the input W as 3. As discussed before, the ξ-simplification T' with the smallest size is (p_1, p_3, p_6, p_8). Since T' has its size equal to 4 which is larger than $W = 3$, we know that $\xi = 1.249$ is not an affordable span.

4.4.4 How to Prune Search Space with a Pivot

In this part, we describe how we can prune at least $\frac{1}{4}$ of the current search space based on a pivot. Suppose that ξ is a pivot. We can prune the current search space based on two different cases.

- *Case 1:* ξ is not an affordable span. In this case, we know that $\xi_o > \xi$ and we can prune values at most ξ. To do this, for each array $\Theta[s : e][j]$ in $\mathcal{A}(\xi, -) \cup \mathcal{A}(\xi, =)$, we prune its values that are at or after the position of its bisector (because they are at most its bisector and its bisector is at most ξ) by shrinking it to $\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$ ($\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$ is dropped if $\lceil \frac{s+e}{2} \rceil - 1 < s$). Note that the number of values pruned is at least $\frac{N(\mathcal{A}(\xi,-))+N(\mathcal{A}(\xi,=))}{2}$. Since ξ is a pivot, we know $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) \geq \frac{|\mathcal{S}|}{2}$ which implies that we have pruned at least $\frac{|\mathcal{S}|}{4}$ values. Here, by shrinking $\Theta[s : e][j]$ to $\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$ in $\mathcal{A}$, we mean updating the triple (s, e, j) with $(s, \lceil \frac{s+e}{2} \rceil - 1, j)$ in the index triplet set $\mathcal{T}$ corresponding to $\mathcal{A}$.
- *Case 2:* ξ is an affordable span. We can perform the pruning operation in a symmetric way as Case 1 by shrinking each array $\Theta[s : e][j]$ in $\mathcal{A}(\xi, +) \cup \mathcal{A}(\xi, =)$ to $\Theta[\lceil \frac{s+e}{2} \rceil + 1 : e][j]$ ($\Theta[\lceil \frac{s+e}{2} \rceil + 1 : e][j]$ is dropped if $\lceil \frac{s+e}{2} \rceil + 1 > e$). Similar to Case 1, we derive that $\frac{1}{4}$ of the current search space is pruned, and the corresponding shrinking operation is executed on $\mathcal{T}$.

In conclusion, using a pivot can prune $\frac{1}{4}$ of the current search space.

To illustrate, consider our running example where Table 5 shows the array set corresponding to the current search space $\mathcal{R}$ containing 49 values. Suppose that we have found a pivot $\xi = 1.249$. Now, we illustrate the pruning process based on ξ. Since ξ is not an affordable span (we know it from the examples discussed before), we prune the search space $\mathcal{R}$ be updating each array $\Theta[s : e][j]$ in $\mathcal{A}(\xi, -) \cup \mathcal{A}(\xi, =)$ to be $\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$. As discussed before, $\mathcal{A}(\xi, -) = \{\Theta[1 : 1][1], \Theta[2 : 7][1], \Theta[1 : 2][2], \Theta[1 : 3][3], \Theta[4 : 7][3], \Theta[1 : 4][4]\}$ and $\mathcal{A}(\xi, =) = \Theta[3 : 7][2]\}$. The arrays in these two sets will be updated and the index triplet set of the updated array set is shown in Table 7. As could be verified, the number of values in the search space represented by this updated index triplet set is equal to 35, i.e., $(49 - 35) = 14$ values have been pruned (note that $14 > \frac{49}{4}$).

Note that our index triplet set for representing the search space makes the process of executing the pruning operations extremely convenient, i.e., all we need is to update the indices (i.e., s and e) of each array $\Theta[s : e][j]$, and thus the pruning operations could be executed in $O(n)$ time since we have $O(n)$ arrays only.

Remark. The pruning operations only shrink the arrays and thus Property 2 still holds for the updated array set which further implies that we can repeat our process to find a pivot ξ wrt the updated search space, perform a span affordability on ξ and prune the updated search space at the next iteration until the search space becomes empty. Note that the process involves $O(\log n)$ iterations only since at least $\frac{1}{4}$ of the search space is pruned at each iteration.

	# of trajectories	total # of positons	average # of positions per trajectory
Geolife	17,621	24,876,978	1,412
T-Drive	10,359	17,740,902	1,713

Table 8: Real datasets

4.4.5 Time & Space Complexity of Span-Search

In this part, we analyze the time and space complexities of *Span-Search*. *Span-Search* proceeds with iterations. At each iteration, it first finds a pivot ξ wrt the current search space (which can be done in $O(n \log n)$ as shown in Section 4.4.2), checks the span affordability on ξ (which can be done in $O(n \log n)$ as shown in Section 4.4.3), and prunes at least $\frac{1}{4}$ of the current search space (which can be done in $O(n)$ as shown in Section 4.4.4). It could be verified easily that the process involves $2 \log n / \log(4/3) = O(\log n)$ iterations. Therefore, the time complexity of *Span-Search* is $O(\log n \cdot (n \log n + n \log n + n)) = O(n \log^2 n)$. Besides, the space complexity of *Span-Search* is simply $O(n)$ which corresponds to the space cost for maintaining the index triplet set.

5. EXPERIMENTS

We used two real datasets in our experiments, namely Geolife and T-Drive. Geolife[3] records the outdoor movements of 182 users in a period of 5 years and T-Drive[4] is a set of taxi trajectories in Beijing. These two datasets are widely used for a broad range of applications on trajectory data [40, 38]. The statistics of these datasets are summarized in Table 8.

Since the experimental results in [24] already show the advantage of using DPTS over PPTS, we focus on the performance of our proposed algorithms in this paper. All algorithms were implemented in C/C++ and ran on a Linux platform with a 2.66GHz machine and 40GB RAM.

5.1 Comparison with Wavelet Transformation

First, following [4], we use *wavelet transformation* [1] as a baseline of trajectory simplification and compare it with our Min-Error mechanism in terms of how good they are for preserving the direction information. The major idea of wavelet transformation is to transform the raw data which corresponds to a set of n numbers into a set of n coefficients (this step does not introduce any information loss and the raw data could be completely restored with these n coefficients) and store k coefficients only where $k < n$, e.g., top-k coefficients (note that this step saves some storage space with the compression rate of k/n, but introduces some information loss since $n - k$ coefficients are dropped). To get an approximation of the raw data (which contains n values), a set of n values is constructed based on the k stored coefficients. We adopt wavelet transformation for trajectory simplification with the purpose of preserving the direction information as follows. We maintain the set of the directions of the segments of a given trajectory (this corresponds to the direction information of the trajectory), perform wavelet transformation on the set of directions and store a certain number of coefficients according to the storage budget. The goodness of wavelet transformation for preserving the direction information is measured by the *maximum* and also *average* angular

[3]http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

[4]http://research.microsoft.com/apps/pubs/?id=152883

difference between an original direction and its corresponding approximated direction constructed based on the stored coefficients. For both measures, the smaller, the better.

We conducted experiments on Min-Error and wavelet transformation by varying the storage budget W, and the results are shown in Figure 8 where "Wavelet trans. (max.)" and "Wavelet trans. (avg.)" denote the maximum and the average angular difference of wavelet transformation, respectively, and "Min-Error (max.)" denotes the maximum angular difference between the direction of a segment $\overline{pp'}$ in the original trajectory and the direction of the segment that approximates $\overline{pp'}$ in the simplified trajectory generated by the exact algorithm, *Error-Search*, for *Min-Error* (note that this corresponds to the direction-based error). Note that for Min-Error, we do not show the average angular difference since it is extremely small. According to these results, we have the following observations. First, wavelet transformation performs poorly when being used for preserving the direction information, e.g., in most cases, wavelet transformation results in high maximum and average angular difference, and this holds even when the storage budget W is near to $|T|$. This essentially tells that wavelet transformation is not suitable for preserving the direction information when being used for trajectory simplification. Second, Min-Error performs significantly better than wavelet transformation in terms of preserving the direction information. Thus, in the following, we focus on Min-Error only in our experiments.

We also show the effects of the storage budget W on the direction-based error in more detail in Figure 9 and we observe that when W is relatively small (e.g., $W \leq 0.2$), a small increase on W yields a significant reduction on the optimal error, while when W is relatively large (e.g., $W \geq 0.5$), even a large increase on W helps a little to reduce the optimal error.

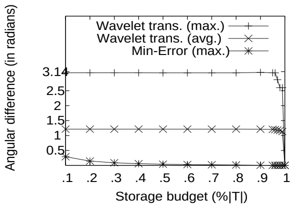


Figure 8: MinError vs. Wavelet

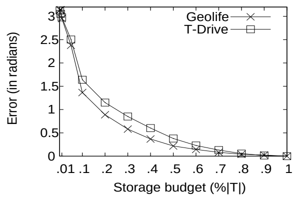


Figure 9: Error trends

We do not adopt the principle of *minimum description length* (MDL) [11] for our Min-Error problem since MDL is for balancing between the size and the error of the simplied trajectory and thus it does not allow users to specify a size constraint or optimize the simplification error.

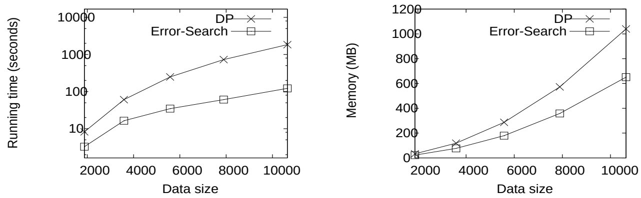
Next, we study the performance of our exact and approximate algorithm in Section 5.2 and in Section 5.3, respectively.

5.2 Performance Study of the Exact Algorithms

In this part, we study the effects of 2 factors, namely the data size (i.e., $|T|$) and the storage budget (i.e., W) on the performance of our exact algorithms, namely *DP* and *Error-Search*. We use 2 measures, namely the running time and the memory.

Effect of $|T|$. The values used for $|T|$ are around 2,000, 4,000, 6,000, 8,000 and 10,000 (W is fixed to be 0.2, i.e., $W = 0.2 * |T|$). For each setting of $|T|$, we select a set of 10 trajectories each of which has its size near to this value and run our exact algorithms on each of these trajectories. Then, we average the experimental results on these trajectories (this policy is used throughout our experiments without specification). Figure 10 show the results on Geolife. According to these results, *Error-Search* is always faster than *DP*, and the efficiency gap between them becomes larger when the data size increases. This could be easily explained by the fact that *Error-Search* has smaller time/space complexities than *DP*.

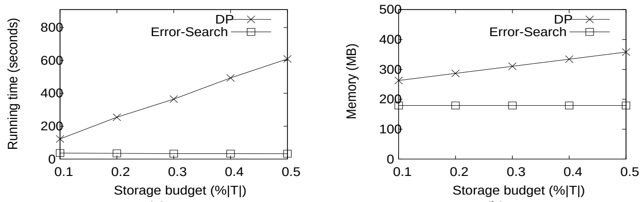
The experimental results on T-Drive are similar and thus they are omitted due to page limit.



(a) (b)

Figure 10: Effects of data size $|T|$ (Geolife)

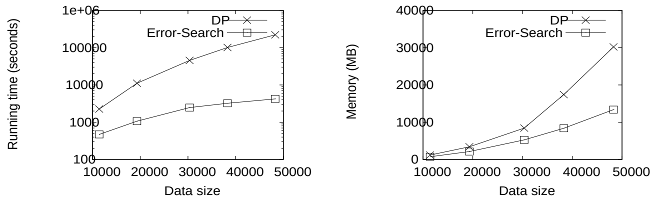
Effect of W. The values used for W are 0.1, 0.2, 0.3, 0.4 and 0.5 ($|T|$ is fixed to about 6,000). The results are presented in Figure 11. We observe that *DP* has both its running time and its memory increase with W, which could be explained by the fact that *DP* has its problem space proportional to W. In contrast, W has no significant effects on *Error-Search* since *Error-Search* has its time/space complexities independent of W.



(a) (b)

Figure 11: Effects of storage budget W (Geolife)

Scalability test. Figure 12 shows the scalability test results on the exact algorithms. We observe that *DP* is limited to medium-sized datasets only while *Error-Search* can go much further. For example, on a trajectory with about 50,000 positions, *DP* runs for several days and occupies nearly 30GB memory, while *Error-Search* runs for about 1hr and occupies about 10GB memory.



(a) (b)

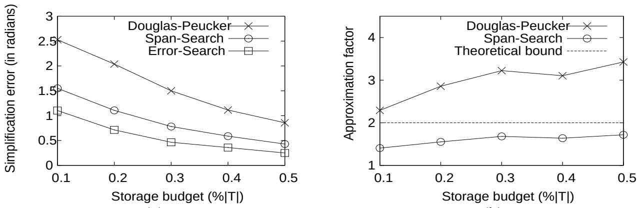
Figure 12: Scalability test (Geolife)

5.3 Performance Study of the Approximate Algorithms

In this part, we study the effects of $|T|$ and W on two approximate algorithms, namely *Span-Search* and *Douglas-Peucker*. *Douglas-Peucker* is an adaptation of the traditional Douglas-Peucker algorithm [8], whose major idea is to recursively cut the trajectory at one of the end of the segment that has the *greatest* angular difference from the segment linking the start position and the end position of this trajectory until we have $W - 1$ sub-trajectories and then use one segment to approximate each sub-trajectory. We note here that *Douglas-Peucker* is the most popular algorithm for trajectory simplification in the literature [8, 27, 12]. We use 3 measures, namely the running time, the memory and the *approximation factor*. The approximation factor of an approximate algorithm is defined to be $\epsilon(T')/\epsilon(T'_o)$, where T' is the simplified trajectory returned by this approximate algorithm on a given raw trajectory and

T'_o is the simplified trajectory returned by an exact algorithm on the same raw trajectory. Clearly, the smaller the approximation factor is, the better approximation quality the algorithm has.

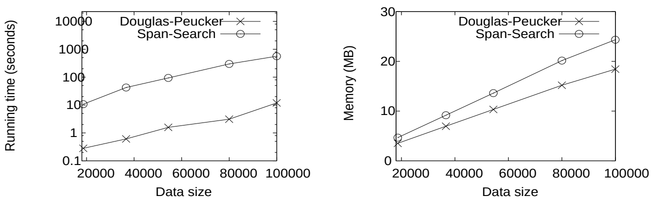
Approximation factor. We present the results with two figures, Figure 13(a) and Figure 13(b). Figure 13(a) shows for each approximate algorithm, the (absolute) error of the simplified trajectory returned and also the optimal error (i.e., the error of the simplified trajectory returned by an exact algorithm such as *Error-Search*), and Figure 13(b) shows the approximation factors of the approximate algorithms. According to these results, *Span-Search* is consistently better than *Douglas-Peucker* in terms of approximation quality. We emphasize here that Douglas-Peucker has its approximation factor usually around 3. In contrast, Span-Search usually achieves an approximation factor around 1.5, though its theoretical worst-case bound is 2. In other words, Douglas-Peucker has an error that is 200% greater than optimum while Span-Search achieves an error only 50% greater than optimum.



(a) (b)

Figure 13: Approximation quality (Geolife)

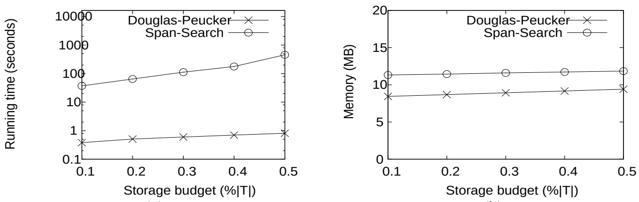
Effect of $|T|$. The values used for $|T|$ are around 20,000, 40,000, 60,000, 80,000 and 100,000 (W is fixed to 0.2). Figure 14 shows the results. According to these results, *Span-Search*, though slower than *Douglas-Peucker*, runs reasonably fast (e.g., on a dataset with about 100,000 positions, *Span-Search* runs less than 1000s). Besides, both *Span-Search* and *Douglas-Peucker* are space efficient (e.g., they occupy less than 30MB) which could be explained by the fact that *Span-Search* has a linear space complexity and so does *Douglas-Peucker*.



(a) (b)

Figure 14: Effects of data size $|T|$ (Geolife)

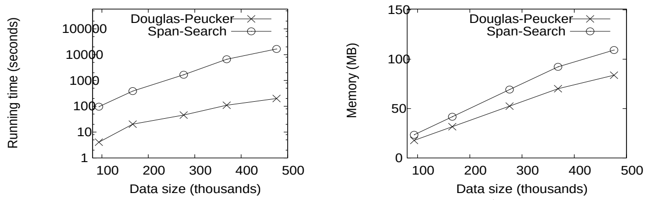
Effect of W. The values used for W are 0.1, 0.2, 0.3, 0.4 and 0.5 ($|T|$ is fixed to about 60,000). The results are shown in Figure 15. We notice that both *Span-Search* and *Douglas-Peucker* are only slightly affected by W. Specifically, when W increases, both the algorithms run a little bit slower. For *Span-Search*, with a larger W, the span affordability check procedure would probably maintain a larger binary search tree and also a larger priority queue which incurs more cost. For *Douglas-Peucker*, with a larger W, it would do more "cut" operations and thus it incurs more cost.



(a) (b)

Figure 15: Effects of storage budget W (Geolife)

Scalability test. Figure 16 shows the scalability test results on the approximate algorithms. According to results, we know that *Span-Search* is scalable to large datasets. For example, on a dataset with about 500,000 positions, *Span-Search* runs for a couple of hours and occupies less than 150MB memory.



(a) (b)

Figure 16: Scalability test (Geolife)

Additional experiments. We also conducted experiments on a variant of *Error-Search* which adopts the *Douglas-Peucker* algorithm for performing each error affordability check *approximately* and thus it corresponds to an approximate algorithm for the Min-Error problem. We observed that this variant of *Error-Search* was dominated by our *Span-Search* algorithm in terms of both the running time and the minimized error. Due to the page limit, we put the details in our technical report [25].

Empirical conclusion. About the exact algorithms, *Error-Search* has its superiority over *DP* in terms of both time and space efficiency. About the approximate algorithms, *Span-Search* has its approximation quality consistently better than *Douglas-Peucker* and is scalable to large datasets.

6. RELATED WORK

Most existing studies on trajectory simplification aim to preserve the *position information* of the trajectory, which we call *position-preserving trajectory simplification* (PPTS), by adopting a *position-based error measurement* for measuring the error of the simplified trajectory [8, 27, 31, 21, 29, 18]. A position-based error of a simplified trajectory is usually defined to be the maximum Euclidean distance between a position on the original trajectory and its "mapped" position on the simplified trajectory. Two major methods have been proposed to define for a position p on the original trajectory its "mapped" position on the simplified trajectory, namely the *closest distance function* [8], which defines the "mapped" position to be the closest position from p on the simplified trajectory, and the *synchronous distance function* [27, 31, 21, 29, 18], which defines the "mapped" position to be the position with the same time stamp on the simplified trajectory as p. The algorithms used by these studies are mainly heuristic-based.

Some other existing studies on trajectory simplification include [22] which aims to minimize the area enclosed by the original trajectory and the simplified trajectory, [32, 7] which consider the semantic information of a trajectory for trajectory simplification, [16, 10, 17] which study the trajectory simplification problem on trajectories constrained on road networks, [36, 34, 31, 14, 19, 21, 15, 29, 20, 18] which study the online trajectory simplification problem, [5] which combines the trajectory simplification process and the encoding process for better compression rate, [4, 9] which study the effects of trajectory simplification on some spatio-temporal queries, [28] which provides a preliminary empirical study on several trajectory simplification algorithms, [6] which proposes a multi-resolution trajectory simplification method, and [41] which provides a preliminary literature study on trajectory simplification.

Another closely related topic is *polygonal curve approximation* [2] (a good survey could be found in [13]). However, none of these studies consider the direction-based error as adopted in this paper.

Recently, Long et al. [24] proposed to preserve the *direction information* of the trajectory for simplification, which is referred to as *direction-preserving trajectory simplification* (DPTS). The authors showed that DPTS not only preserved the direction information by its nature, but also provided guarantees on the position information loss both theoretically and empirically. Within DPTS, the authors identified the Min-Size problem which was to find the simplification of a given trajectory with its error at most a given error tolerance and its size minimized. In this paper, we focus on DPTS, but study a different problem from the Min-Size problem [24], i.e., the Min-Error problem.

7. CONCLUSION

In this paper, we identified a new application scenario for DPTS and defined a corresponding problem, i.e., the Min-Error problem. Then, we designed two exact algorithms, *DP* and *Error-Search*, based on dynamic programming and binary search, respectively. Since the time complexities of the exact algorithms are relatively high, we further developed an approximate algorithm *Span-Search* which runs in $O(n \log^2 n)$ time and gives a 2-factor approximation. We conducted extensive experiments on real datasets which verified our proposed algorithms. There are several interesting research directions. One is to study the DPTS problem in an online setting. Another is to explore other functions based on the direction information for defining the simplification error.

Acknowledgements: We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Cheng Long and Raymond Chi-Wing Wong is supported by the grant FSGRF14EG34, and the research of H. V. Jagadish is supported in part by NSF grant IIS 1250880.

8. REFERENCES

[1] Special issue on data reduction techniques. *IEEE Data Engineering*, 20, 1998.

[2] P. K. Agarwal and K. R. Varadarajan. Efficient algorithms for approximating polygonal chains. *Discrete Computational Geometry*, 23(2):273–291, 2000.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *JCSS*, 7(4):448–461, 1973.

[4] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 2006.

[5] M. Chen, M. Xu, and P. Franti. Compression of gps trajectories. In *Data Compression Conference (DCC), 2012*, pages 62–71, 2012.

[6] M. Chen, M. Xu, and P. Franti. A fast o(n) multiresolution polygonal approximation algorithm for gps trajectory simplification. *IEEE Transactions on Image Processing*, 21(5), 2012.

[7] Y. Chen, K. Jiang, Y. Zheng, C. Li, and N. Yu. Trajectory simplification method for location-based social networking services. In *IWLBSN*, pages 33–40. ACM, 2009.

[8] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 11(2):112–122, 1973.

[9] E. Frentzos and Y. Theodoridis. On the effect of trajectory compression in spatiotemporal querying. In *Advances in Databases and Information Systems*, pages 217–233. Springer, 2007.

[10] O. Ghica, G. Trajcevski, O. Wolfson, U. Buy, P. Scheuermann, F. Zhou, and D. Vaccaro. Trajectory data reduction in wireless sensor networks. *International Journal of Next-Generation Computing*, 1(1), 2010.

[11] P. D. Grünwald, I. J. Myung, and M. A. Pitt. *Advances in minimum description length: Theory and applications*. MIT press, 2005.

[12] J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. *Algorithms and Computation*, pages 763–775, 2007.

[13] P. S. Heckbert. *Survey of polygonal surface simplification algorithms*, 1997.

[14] N. Hönle, M. Grossmann, D. Nicklas, and B. Mitschang. Preprocessing position data of mobile objects. In *MDM'08*.

[15] N. Hönle, M. Grossmann, S. Reimann, and B. Mitschang. Usability analysis of compression algorithms for position data streams. In *SIGSPATIAL*. ACM, 2010.

[16] G. Kellaris, N. Pelekis, and Y. Theodoridis. *Trajectory compression under network constraints*, pages 392–398. Advances in Spatial and Temporal Databases. Springer, 2009.

[17] G. Kellaris, N. Pelekis, and Y. Theodoridis. Map-matched trajectory compression. *Journal of Systems and Software*, 2013.

[18] A. Kolesnikov. Efficient online algorithms for the polygonal approximation of trajectory data. In *MDM'11*, pages 49–57.

[19] R. Lange, F. Dürr, and K. Rothermel. Online trajectory data reduction using connection-preserving dead reckoning. In *ICMUS: Computing, Networking, and Services*, page 52. ICST, 2008.

[20] R. Lange, F. Dürr, and K. Rothermel. Efficient real-time trajectory tracking. *The VLDB Journal*, 20(5):671–694, 2011.

[21] R. Lange, T. Farrell, F. Dürr, and K. Rothermel. Remote real-time trajectory simplification. In *PerComm'09*, pages 1–10, 2009.

[22] G. Liu, M. Iwai, and K. Sezaki. A method for online trajectory simplification by enclosed area metric. *ICMU'12*, 2012.

[23] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing. Discovering spatio-temporal causal interactions in traffic data streams. In *SIGKDD*, 2011.

[24] C. Long, R. C.-W. Wong, and H. V. Jagadish. Direction-preserving trajectory simplification. In *VLDB*, 2013.

[25] C. Long, R. C.-W. Wong, and H. V. Jagadish. Trajectory simplification: On minimizing the direction-based error (technical report). In *http://www.cse.ust.hk/~raywong/paper/minError.pdf*, 2014.

[26] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding time period-based most frequent path in big trajectory data. In *SIGMOD*, 2013.

[27] N. Meratnia and R. de By. Spatiotemporal compression techniques for moving point objects. *EDBT'04*, pages 561–562.

[28] J. Muckell, J. H. Hwang, C. T. Lawson, and S. Ravi. Algorithms for compressing gps trajectory data: an empirical evaluation. In *SIGSPATIAL*, pages 402–405, 2010.

[29] J. Muckell, J. H. Hwang, V. Patil, C. T. Lawson, F. Ping, and S. Ravi. Squish: an online approach for gps trajectory compression. In *COM.Geo'11*, pages 13:1–13:8.

[30] H. Pham, C. Shahabi, and Y. Liu. Ebm-an entropy-based model to infer social strength from spatiotemporal data. *SIGMOD*, 2013.

[31] M. Potamias, K. Patroumpas, and T. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *SSDBM'06*, pages 275–284.

[32] F. Schmid, K.-F. Richter, and P. Laube. *Semantic trajectory compression*, pages 411–416. Advances in Spatial and Temporal Databases. Springer, 2009.

[33] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. C. Hung, and W. C. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE'12*, pages 186–197.

[34] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, and D. Vaccaro. On-line data reduction and the quality of history in moving objects databases. In *WDEWMA*, pages 19–26, 2006.

[35] L.-Y. Wei, Y. Zheng, and W.-C. Peng. Constructing popular routes from uncertain trajectories. In *SIGKDD*, pages 195–203. ACM, 2012.

[36] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and parallel databases*, 7(3):257–387, 1999.

[37] X. Xiao, Y. Zheng, Q. Luo, and X. Xie. Inferring social ties between users with human location history. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–17, 2012.

[38] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *KDD'11*, pages 316–324.

[39] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *ICDE*, 2013.

[40] Y. Zheng, X. Xie, and W. Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Engineering Bulletin*, 33(2):32–40, 2010.

[41] Y. Zheng and X. Zhou. *Computing with Spatial Trajectories*. Springer Publishing Company, Incorporated, 1st edition, 2011.