Trajectory Simplification: On Minimizing the Direction-based Error


Published in:
Proceedings of the 41st International Conference on Very Large Data Bases

Document Version:
Publisher's PDF, also known as Version of record

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright 2014 VLDB Endowment
This work is licensed under the Creative Commons AttributionNonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ . This license permits distribution and reproduction for non-commercial purposes, provided the author and source are cited.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.
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 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 $O(n^2C\log n)$, 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.
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 \(i = 1, 2, ..., n - 1\).

Table 1: Directions of the segments (running example)

<table>
<thead>
<tr>
<th>Segments</th>
<th>(\mathcal{P}_E)</th>
<th>(\mathcal{P}_D)</th>
<th>(\mathcal{P}_T)</th>
<th>(\mathcal{P}_P)</th>
<th>(\mathcal{P}_R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directions</td>
<td>0.785</td>
<td>5.820</td>
<td>1.107</td>
<td>0</td>
<td>0.464</td>
</tr>
</tbody>
</table>

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

Figure 2: direction

Figure 3: angular range

Figure 4: angular difference

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

\[
\Delta(\theta_1, \theta_2) = \min[(\theta_1 - \theta_2), 2\pi - (\theta_1 - \theta_2)]
\]

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

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

\[
\epsilon(p_{s_j}p_{s_{j+1}}) = \max_{1 \leq k < m} \left|\theta(p_{s_j}p_{s_{j+1}}) - \theta(p_{s_k}p_{s_{k+1}})\right|
\]

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

\[
\epsilon(T') = \max_{1 \leq i \leq n} \epsilon(p_i p_{i+1})
\]

To illustrate, consider our running example in Figure 1. Suppose \(T'' = (p_1, p_5, p_3)\). Since \(\theta(p_1p_5) = 0.322\) and \(\theta(p_5p_3) = 0.785\), we know \(\Delta(\theta(p_1p_5), \theta(p_5p_3)) = 0.463\). Similarly, we can compute \(\Delta(\theta(p_5p_3), \theta(p_3p_1)) = 0.785\), \(\Delta(\theta(p_3p_1), \theta(p_1p_5)) = 0.322\). Thus, we know \(\epsilon(p_1p_5p_3) = \max(0.463, 0.785, 0.785, 0.322) = 0.785\). Besides, we can compute \(\epsilon(p_5p_3) = 0.742\). Thus, we know \(\epsilon(T'') = \max(0.742, 0.322, 0.785) = 0.785\).

In the following, when we write \(\epsilon(p_i p_{i+1})\) (\(0 \leq i < j \leq n\)), we mean the simplification error of \(p_i p_{i+1}\) 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 specify 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 at 3. Then, \(T'' = (p_1, p_3, p_5)\) 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.
error if there exists an affordable simplification $T'$ in $\mathcal{F}$ such that $\epsilon(T') \leq \epsilon$. Thus, $\epsilon_0$ corresponds to the smallest affordable error.

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 $\epsilon$ is an affordable error) which we call the error affordability check on $\epsilon$. 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 $\epsilon$ is an affordable error, then any $\epsilon' > \epsilon$ is also an affordable error. Thirdly, we return $T'_\epsilon$ the affordable simplification corresponding to the smallest affordable error found (which is exactly $\epsilon_0$).

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 $\epsilon$, and (3) the time and space complexities of Error-Search.

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

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

Let $\theta([p_i])$ be the opposite direction of $\theta([p_i])$, i.e., $\theta([p_i]) = \left(\theta([p_i]) + \pi \right) \mod 2\pi$. We observe that $\theta'$ is exactly the direction in $[\theta(i : j)]$ which has the smallest angular difference from $\theta([p_i])$. This is simply because any direction $\theta$ in $[\theta(i : j)]$ has its angular difference from $\theta([p_i])$ equal to $\pi$ minus its angular difference from $\theta([p_i])$, i.e., $\Delta(\theta, \theta([p_i])) = \pi - \Delta(\theta, \theta([p_i]))$.

To illustrate, consider Figure 5. $\theta([p_1]), \theta([p_2]), ... , \theta([p_n])$ correspond to $\theta([1 : 5])$, $\theta([2 : 3])$ and $\theta([2 : 4])$ are also shown. As could be verified, $\theta([p_i])$ is the direction in $[\theta(i : j)]$ which has the greatest angular difference from $\theta([p_i])$ and also the smallest angular difference from $\theta([p_i])$.

Thus, we propose to search $\theta'$ with two steps. First, we sort the directions in $[\theta(i : j)]$ in ascending order and let $\theta_1, \theta_2, ..., \theta_{n-1}$ be the resulting sorted list (note that sorting from scratch incurs a cost of $\mathcal{O}(n \log n)$ 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 $\epsilon([p_i])$ if we compute
\(\epsilon(p_{12} \tau)\) 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 \(\theta(p_{12} \tau)\) (i.e., \(\theta^*\)) 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 \(\theta^*\) in \(O(\log n)\) time.

In view of the above discussion, we know that \(E\) can be constructed in \(O(n^2 \log n)\) time since we have \(O(n^2)\) instances of \(\epsilon(p_{12} \tau)\) each with a computation cost of \(O(\log n)\).

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

\textbf{Lemma 1.} Let \(\epsilon\) be a non-negative value and \(T'\) be a simplification of \(T\) with its \(\epsilon\) and its size minimized. Then, \(\epsilon\) is an affordable error if \(T'\) is affordable.

\textbf{Proof.} "\(\Rightarrow\):" Suppose that \(\epsilon\) 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.

"\(\Leftarrow\):" Clearly, \(\epsilon\) is an affordable error since \(T'\) is affordable and has its error at most \(\epsilon\) by definition.

Lemma 1 suggests that the error affordability check on a given value \(\epsilon\) can be implemented with the following two steps. First, we compute the simplification \(T'\) of \(T\) with its \(\epsilon\) and its size minimized. This essentially corresponds to solving a \textit{Min-Siz}e problem instance [24] with its input trajectory as \(T\) and its input error tolerance as \(\epsilon\). Thus, this step can be done by executing an exact algorithm of the \textit{Min-Size} problem. Second, we check whether \(T'\) is affordable (i.e., the size of \(T'\) is at most \(W\)). If yes, then \(\epsilon\) is an affordable span. Otherwise, it is not. Since the time complexity of the exact algorithm for the \textit{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 \(O(n^2C)\).

2. \textbf{Time & Space Complexity of \textit{Error-Search}.} In conclusion, the time complexity of \textit{Error-Search} is \(O(n^2C\log n)\) since the cost of constructing \(E\) is \(O(n^2C\log n)\), the cost of sorting \(E\) (for binary search) is \(O(n^2 \log n)\) (\(= O(n^2C \log n)\)), 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 \textit{Error-Search} is \(O(n^2)\) which corresponds to the space cost of storing the search space \(E\).

3. \textbf{APPROXIMATE ALGORITHM}

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

4. \textbf{An Estimator of Error}

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

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

(3)

From the above discussion, we know that \(E\) can be constructed in \(O(n^2 \log n)\) time since we have \(O(n^2)\) instances of \(\epsilon(p_{12} \tau)\) each with a computation cost of \(O(\log n)\).

In view of the above discussion, we know that \(E\) can be constructed in \(O(n^2 \log n)\) time since we have \(O(n^2)\) instances of \(\epsilon(p_{12} \tau)\) each with a computation cost of \(O(\log n)\).

In view of the above discussion, we know that \(E\) can be constructed in \(O(n^2 \log n)\) time since we have \(O(n^2)\) instances of \(\epsilon(p_{12} \tau)\) each with a computation cost of \(O(\log n)\).

In view of the above discussion, we know that \(E\) can be constructed in \(O(n^2 \log n)\) time since we have \(O(n^2)\) instances of \(\epsilon(p_{12} \tau)\) each with a computation cost of \(O(\log n)\).

In view of the above discussion, we know that \(E\) can be constructed in \(O(n^2 \log n)\) time since we have \(O(n^2)\) instances of \(\epsilon(p_{12} \tau)\) each with a computation cost of \(O(\log n)\).
with its input trajectory and its input storage budget both the same as the Min-Error problem. Then, \(\epsilon(T'_c) \leq 2 \cdot \epsilon(T'')\).

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

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

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

Suppose that \(mcar(\theta(s_k : s_{k+1})) = [\theta_a, \theta_b]\). Note that \(\theta_a\) and \(\theta_b\) are two directions in \(\theta(s_k : s_{k+1})\). We have two cases. Case 1: \(\xi([\theta_A, \theta_B]) \leq \pi\). For illustration, consider Figure 7(a). In this case, \(\theta(p_{s_k}, p_{s_{k+1}})\) is covered by \([\theta_a, \theta_b]\). Therefore, we have

\[
\epsilon(p_{s_k}, p_{s_{k+1}}) = \max\{\Delta(\theta(p_{s_k}, p_{s_{k+1}}), \theta_a), \Delta(\theta(p_{s_k}, p_{s_{k+1}}), \theta_b)\} 
\in [1/2, 1] \cdot (\Delta(\theta(p_{s_k}, p_{s_{k+1}}), \theta_a) + \Delta(\theta(p_{s_k}, p_{s_{k+1}}), \theta_b))
\]

\[
= [1/2, 1] \cdot \xi([\theta_a, \theta_b]) \in [1/2, 1]
\]

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

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

Case 2(b): \(\theta(p_{s_k}, p_{s_{k+1}})\) is not covered by \([\theta_a, \theta_b]\). Then, \(\theta(p_{s_k}, p_{s_{k+1}})\) is covered by \([\theta_a, \theta_b]\). For illustration, consider Figure 7(c). Let \(\theta_d\) and \(\theta_d\) be two directions in \(\theta(s_k : s_{k+1})\) such that \(\theta_d\) and \(\theta_d\) are in \([\theta_a, \theta_b]\) and no directions in \(\theta(s_k : s_{k+1})\) other than \(\theta_d\) and \(\theta_d\) are in \([\theta_a, \theta_b]\) (Note that \(\theta_d\) and \(\theta_d\) always exist). Then, we know that \(\xi([\theta_d, \theta_d])\) corresponds to a covering angular range of \(\theta(s_k : s_{k+1})\) and \(\theta(p_{s_k}, p_{s_{k+1}})\) falls in \([\theta_d, \theta_d]\). Besides, we know \(\xi([\theta_d, \theta_d]) \geq \xi([\theta_a, \theta_b])\) since \([\theta_a, \theta_b]\) is the minimum covering angular range of \(\theta(s_k : s_{k+1})\). Similar to Case 1, we have

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

which implies that

\[
\xi([\theta_d, \theta_d]) \geq \xi([\theta_a, \theta_b]) \geq \xi([\theta_a, \theta_b])
\]

Furthermore, we have

\[
\epsilon(p_{s_k}, p_{s_{k+1}}) \leq \pi \leq \xi([\theta_a, \theta_b])
\]

In combination, we have

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

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

\[
\xi(T') = \xi(mcar(\theta(s_{k'} : s_{k''}))) \leq 2 \cdot \epsilon(p_{s_{k'}}, p_{s_{k''}}) 
\leq 2 \cdot \epsilon(p_{s_k}, p_{s_{k+1}}) = 2 \cdot \epsilon(T')
\]

and

\[
\xi(T') = \xi(mcar(\theta(s_{k'} : s_{k''}))) \geq \xi(mcar(\theta(s_{k''} : s_{k''+1})))
\geq \epsilon(p_{s_{k''}}, p_{s_{k''+1}}) = \epsilon(T')
\]

By using Equations (6) and (7), we obtain \(\epsilon(T'_c) \in [1, 2]\).

In the second part, we show that \(\epsilon(T'_c) \leq 2 \cdot \epsilon(T'')\) as follows.

\[
\epsilon(T'_c) \leq \xi(T'_c) \leq \xi(T'') \leq 2 \cdot \epsilon(T'')
\]

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

4.3.1 Concepts & Search Space \(S\)

We introduce some concepts used for defining the search space \(S\) and then give a precise definition of \(S\). Suppose that \(T'_c = (p_{s_1}, p_{s_2}, ..., p_{s_m})\). From Equation (4), we know

\[
\xi(T'_c) = \max_{i \in [1, m]} \{\xi(mcar(\theta(s_i : s_{i+1})))\}
\]

Let \(k_* = \arg\max_{i \in [1, m]} \{\xi(mcar(\theta(s_i : s_{i+1})))\}\). Then, we have \(\xi_o = \xi(T'_c) = \xi(mcar(\theta(s_{k_*} : s_{k_*+1})))\) and \(\xi_o\) exist two segments of \(\theta(s_{k_*} : s_{k_*+1})\). Then, we derive that \(\xi_o = \xi([\theta', \theta''])\). By Equation (3), we have

\[
\xi_o = \begin{cases} 
\theta' - \theta & \text{if } \theta' > \theta \\
2\pi - (\theta - \theta') & \text{if } \theta' < \theta 
\end{cases}
\]

Essentially, \(\theta'\) and \(\theta''\) could be the directions of any two segments of \(T\). Thus, we have the following observation.

**Observation 1** (Pairwise Direction Difference). Let \(\xi_o\) be the optimal span of the Min-Span problem. There exist two segments of \(T\) such that \(\xi_o\) is equal to either \(\theta' - \theta\) or \(2\pi - (\theta - \theta')\) where \(\theta\) and \(\theta'\) are the directions of the two segments.

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

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

To illustrate, consider Table 3 which shows the sorted list of \(D\) of the trajectory \(T\) in Figure 1 and Table 4 which shows the corresponding matrix \(\Theta\). With Observation 1, it is easy to verify that \(\xi_o\) is in \(S\). We present this result in the following lemma.

**Lemma 3.** The span of the optimal solution of the Min-Span problem (i.e., \(\xi_o\)) is in \(S\).

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 \(3\), \(\xi_o\) is equal to 1.570 which corresponds to \(\Theta[6][4]\).

Given a value \(\xi\), we say that \(\xi\) is an affordable span iff there exists an affordable simplification of \(T\) with its span at most \(\xi\). It immediately follows that \(\xi_o\) corresponds to the smallest affordable span. With Lemma 3, we know that \(\xi_o\) is the smallest affordable span in \(S\).
Table 3: Sorted list of the directions in θ[1 : 8]

<table>
<thead>
<tr>
<th>i</th>
<th>θ_1</th>
<th>θ_2</th>
<th>θ_3</th>
<th>θ_4</th>
<th>θ_5</th>
<th>θ_6</th>
<th>θ_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
</tbody>
</table>

Table 4: Matrix Θ defined by Equation (9)

<table>
<thead>
<tr>
<th>i</th>
<th>θ_1</th>
<th>θ_2</th>
<th>θ_3</th>
<th>θ_4</th>
<th>θ_5</th>
<th>θ_6</th>
<th>θ_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>1</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>2</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>3</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>4</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>5</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>6</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
<tr>
<td>7</td>
<td>0.464</td>
<td>0.785</td>
<td>1.107</td>
<td>1.499</td>
<td>1.890</td>
<td>2.281</td>
<td>2.672</td>
</tr>
</tbody>
</table>

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

Table 6: The index triplet set (for the original search space S)

<p>| S_ξ |</p>
<table>
<thead>
<tr>
<th>S_ξ (i)</th>
<th>S_ξ (j)</th>
<th>S_ξ (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
</tr>
</tbody>
</table>

Table 7: The index triplet set (for the updated search space resulting from the pruning based on pivot ξ = 1.249)

4.3.2 Strategy of Searching over S

After we introduced the concepts and defined the search space S in the previous section, in this section, we present a strategy called Span-Search for finding the optimal span ξ on 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 ξ(T’) ≤ ξ if ξ is an affordable span.

As we described before, we know that ξ_S is the smallest affordable span in S. Thus, we propose to find ξ_S with three steps.

- **Step 1 (Searching Step):** Step 1 is to find a value ξ from 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 S (i.e., the smallest affordable span in 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 ξ < ξ_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 S to be found until there is no remaining value in 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 S as ξ. The algorithm with this strategy is too costly since S involves Ω(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 ξ < ξ’. 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 S which occupies O(n^2) space) and too costly (since it introduces extra cost for finding the medians which takes O(n log n) time).

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 S explicitly as Linear-Search and Binary-Search do; instead, it materializes a concise representation of 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 a random value from the current search space) or Binary-Search (on the median of the current search space).

- **Span-Search** needs to perform a span affordability check on a given value will be introduced in Section 4.4.3.

- **Span-Search** prunes at least 1/2 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 S

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

We introduce some related concepts first. Given an l-sized array X and two integers i, i’ ∈ [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 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 ∈ [1, n − 1] where s ≤ e, we denote by θ[s, e, j] the array containing the values between the s^th position

1. **Table 3: Sorted list of the directions in θ[1 : 8]**

2. **Table 4: Matrix Θ defined by Equation (9)**

3. **Table 5: The array set representing matrix Θ**

4. **Table 6: The index triplet set (for the original search space S)**

5. **Table 7: The index triplet set (for the updated search space resulting from the pruning based on pivot ξ = 1.249)**

6. **Algorithm 1: Span-Search**

1. Initialize the index triplet set T of S (Section 4.4.1)
2. //Steps 1 & 2
3. while there exist values in the current search space represented by T do
4. Find a pivot ξ wrt the current search space (Section 4.4.2)
5. Perform a span affordability check on ξ (Section 4.4.3)
6. Update ξ_best and T_best if necessary
7. Prune the search space with ξ by updating T (Section 4.4.4)
8. //Step 3
9. return ξ_best and T_best

One can either sort the values in S at the right beginning with O(n log n^2) 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 $\Theta$, i.e., $\Theta[s : e][j] = \{\Theta[s][j], \Theta[s+1][j], ..., \Theta[e][j]\}$.

For each column of $\Theta$, 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 $\Theta$ are organized with $2(n - 1) - 1 (= O(n))$ arrays each in the form of $\Theta[s : e][j]$. Let $A$ be the set containing all these arrays. Thus, the size of $A$ is $O(n)$. Note that the multi-set of values of the arrays in $A$ is exactly equal to $S$. In the following, for clarity, when we write $A$, we mean the array set corresponding to the matrix $\Theta$ of the search space $S$.

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 $A$.

A nice feature about $A$ is that all arrays in $A$ are non-increasing 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 $A$ 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 $B$ be the search space and $A$ be the corresponding array set. The index triplet set of $B$, denoted by $T$, is the set containing triplets of the indices of all arrays in $A$. That is,

$$T = \{(s, e, j) : \Theta[s : e][j] \subseteq A\}$$

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

Interestingly, $T$ alone concisely represents the multi-set of values of the arrays in $A$ (or the search space $S$). This is because for each triplet $(s, e, j) \in T$, 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 using Equation (9). Instead, we materialize $T$ only. In the following, for the sake of convenience, we refer $A$ instead of $T$ to represent the entire search space $S$ (though $T$ is the materialized version for $A$).

### 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 *bisection* and its related concepts.

For each array $\Theta[s : e][j]$ in $A$, we define its bisection, denoted by $b(s, e, j)$, to be $\Theta[\lfloor \frac{s+e}{2} \rfloor][j]$. 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 bisection $b(s, e, j)$ and at least half of the values in $\Theta[s : e][j]$ are at least this bisection $b(s, e, j)$. For example, the bisection of array $\Theta[3 : 7][2]$ is $\Theta[5][2]$ which is equal to 1.249 (See Table 4).

For a given value $\xi \in S$, the arrays in $A$ could be categorized into three disjoint groups, namely the group containing those arrays with the bisections strictly smaller than $\xi$ which we denote by $A(\xi, -)$, and the group containing those arrays with the bisections strictly larger than $\xi$ which we denote by $A(\xi, +)$. Let $N(A(\xi, -))$, $N(A(\xi, +))$, and $N(A(\xi, \pm))$ be the size of the multi-set of the values of the arrays in $A(\xi, -)$, $A(\xi, +)$, and $A(\xi, \pm)$, respectively. Note that $N(A(\xi, -)) + N(A(\xi, +)) + N(A(\xi, \pm)) = |S|$ (see Table 8).

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

**DEFINITION 1 (PIVOT).** Given a value $\xi \in S$, $\xi$ is defined to be a pivot wrt $S$ if

$$\min(N(A(\xi, -)) + N(A(\xi, +))) / |S| \geq 1 / 2$$

For example, $\xi = 1.249$ corresponds to a pivot wrt $S$ since

$$\min(N(A(\xi, -)) + N(A(\xi, +))) = \min(20 + 5, 24 + 5) = 25 \geq \frac{49}{2} = (49/2)/2$$

In the following, we simply write “a pivot wrt $S” as “a pivot” if the context of $S$ is clear.

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

**PROPERTY 3.** Given $\xi, \xi' \in B$ with $\xi, \xi' \not\in B$, we have

$$N(A(\xi, -)) + N(A(\xi, +)) \geq N(A(\xi', -)) + N(A(\xi', +))$$

This essentially says that $N(A(\xi, -)) + N(A(\xi, +))$ is non-decreasing while $N(A(\xi', -)) + N(A(\xi', +))$ is non-increasing when $\xi$ increases.

**PROOF.** This is simply because $A(\xi, -) \cup A(\xi, +) \subseteq A(\xi', -)$ and $A(\xi', +) \subseteq A(\xi', +)$.

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

**LEMA 4. At least one of the values in $B$ is a pivot wrt $S$.**

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

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

Consider $\xi_{n-1}$. Similarly, we know $N(A(\xi_{n-1}, -)) = 0$ and thus $N(A(\xi_{n-1}, +)) + N(A(\xi_{n-1}, +)) = |S|/2$. Therefore, we know $h_2 \in B$ such that $N(A(\xi_{n-1}, -)) + N(A(\xi_{n-1}, +)) = |S|/2$. Similarly, we have $h_2 \in B$ such that $N(A(\xi_{n-1}, -)) + N(A(\xi_{n-1}, +)) = |S|/2$ and $N(A(\xi_{n-1}, +)) + N(A(\xi_{n-1}, +)) = |S|/2$. We consider 3 cases:

**Case 1:** $h_2 = h_1 + 1$. We have $N(A(\xi_{n-1}, -)) + N(A(\xi_{n-1}, +)) = |S|/2 + |S|/2 = |S|$ which leads to a contradiction. **Case 2:**
First, we compute the \( p \) constraint that the span of the minimum covering angular range \( \xi \) is always a pivot, but also implies that we can focus on \( B \).

According to Lemma 4, we can focus on \( B \) and the number of values in an array \( \text{array} \) in a smarter way with a binary search over \( B \) based on the monotonicity properties shown in Property 3. Specifically, we first sort the values in \( B \) in ascending order and obtain a sorted list. Let \( b_m \) be the value at the middle of the list. Then, we compute \( N(A(b_m, = )) \), \( N(A(b_m, = )) \), and \( N(A(b_m, + )) \). If \( \min \{ N(A(b_m, = )) + N(A(b_m, = )) \} \geq \frac{\xi}{2} \), we return \( b_m \) as a pivot; otherwise, we have two cases.

- **Case 1:** \( N(A(b_m, = )) + N(A(b_m, = )) < \frac{\xi}{2} \). In this case, we can safely prune all values that are at most \( b_m \) in \( B \). Hence, 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 \( S \) (or \( A \)).

- **Case 2:** \( N(A(b_m, + )) + N(A(b_m, = )) < \frac{\xi}{2} \). In this case, we can safely prune all values that are at least \( b_m \) in \( B \).

In conclusion, if \( b_m \) is a pivot, we are done, and otherwise we can prune at least half of the search space \( B \) 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 \( \xi \)

In this part, we introduce our method for performing the span affordability check on a given \( \xi \).

Let \( \xi \) be a non-negative value. Given a simplification \( T' \) of \( T \), we say that \( T' \) is a \( \xi \)-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 \( \xi \) as follows. First, we compute the \( \xi \)-simplification with the smallest size, say, \( T'' \). Then, we compare \( |T''| \) with \( W \). If \( |T''| \leq W \), we conclude that \( \xi \) is an affordable span; otherwise, we conclude that \( \xi \) is not an affordable span. The correctness of this method is obvious and the remaining issue is how to find the \( \xi \)-simplification with the smallest size for a given \( \xi \).

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 \( \xi \). To do it, it checks the position index \( j \) starting from \( i + 1 \) one by one. If \( \xi(\text{mear}(\theta[i : j])) \leq \xi \), it continues to check the next position index by updating \( j \) to \( j + 1 \) until either \( j > n \) (i.e., \( j = n + 1 \)) or \( \xi(\text{mear}(\theta[i : j])) > \xi \). Then, it appends \( p_{j+1} \) to \( T' \) in either the case of \( j > n \) (i.e., \( j = n + 1 \)), or the case of \( \xi(\text{mear}(\theta[i : j])) > \xi \), the segments between \( p_j \) and \( p_{j+1} \) form the longest possible sequence starting from \( p_j \) 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.

We illustrate Algorithm 2 with the input trajectory as \( T \) in Figure 1 and \( \xi \) 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 \( \xi(\text{mear}(\theta[1 : 2])) = 0 \) since \( \theta[1 : 2] = \theta[\{p_1,p_2\}] = \{0.785, 0.785\} \). Note that \( \xi(\text{mear}(\theta[1 : 2])) < \xi \) and \( \xi(\text{mear}(\theta[1 : 1])) = \xi \). In this case, \( \xi \) increases \( j \) to \( j = 1 + 3 \). Again, it computes \( \xi(\text{mear}(\theta[1 : 3])) = 1.248 \). Since \( j \) is not \( n \) and \( \xi(\text{mear}(\theta[1 : 3])) < \xi \), it updates \( j \) to \( j + 1 = 4 \). Then, it computes \( \xi(\text{mear}(\theta[1 : 4])) = 1.570 \). This time, \( \xi(\text{mear}(\theta[1 : 4])) > \xi \), 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 \( i = j + 1 = 5 \). It implies that \( (p_2, p_3, p_4) \) is the longest possible sequence starting from \( p_1 \) which has the span at most \( \xi \). It repeats the same process with the new starting position \( p_3 \) and keeps increasing \( j \) by 1 until \( j = 7 \) since \( \xi(\theta[3 : 7]) = 1.892 > \xi \). Then, it appends \( p_{j+1} \) (i.e., \( p_8 \)) to \( T' \) (thus \( T' \) becomes \( (p_1, p_3, p_8) \) and updates \( i \) to \( i = j + 1 = 6 \). It continues the same process with the new starting position \( p_8 \) and keeps increasing \( j \) by 1 until \( j = 9 \) since \( j > n \). Then, it appends \( p_{j+1} \) (i.e., \( p_9 \)) to \( T' \) (thus \( T' \) becomes \( (p_1, p_3, p_8, p_9) \) and stops the process. At the end, it returns \( T' \) which is \( (p_1, p_3, p_8, p_9) \).

**Lemma 5.** Algorithm 2 finds the \( \xi \)-simplification with the smallest size for a given \( \xi \).

**Proof.** Let \( T^n = (p_{s_1}, p_{s_2}, \ldots, p_{s_m}) \) be the simplification returned by Algorithm 2. Let \( T'' = (p_{s_1}, p_{s_2}, \ldots, p_{s_k}) \) be the \( \xi \)-simplification with the smallest size. By definition, we have \( s_1 = r_1 - 1 \) and \( s_m = t_1 \). 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 induction.

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

Therefore, we have \( s_k \geq r_k = n \), 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 $ξ = 1.249$ 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 $(p_1, p_3, p_4, p_6)$. Since $T'$ has its size equal to 4 which is larger than $W = 3$, we know that $ξ = 1.249$ 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 $\frac{1}{5}$ 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 $ξ > ξ$ and we can prune values at most $ξ$. To do this, for each array $Θ[s : e][j]$ in $A(ξ, -) \cup A(ξ, =)$, 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 $Θ[s : \lfloor \frac{ξ + e}{2} \rfloor - 1][j]$ ($Θ[s : \lfloor \frac{ξ + e}{2} \rfloor - 1][j]$ is dropped if $\lfloor \frac{ξ + e}{2} \rfloor - 1 < s$). Note that the number of values pruned is at least $N(A(ξ, -)) + N(A(ξ, =))$. Since $ξ$ is a pivot, we know $N(A(ξ, -)) + N(A(ξ, =)) \leq \frac{|S|}{2}$ which implies that we have pruned at least $\frac{|S|}{2}$ values. Here, by shrinking $Θ[s : e][j]$ to $Θ[s : \lfloor \frac{ξ + e}{2} \rfloor - 1][j]$ in $A$, we mean updating the triple $(s, e, j)$ with $(s, [\frac{ξ + e}{2}] - 1, j)$ in the index triplet set $T$ corresponding to $A$.

- **Case 2**: $ξ$ is an affordable span. We can prune the performing operation in a symmetric way as Case 1 by shrinking each array $Θ[s : e][j]$ in $A(ξ, -) \cup A(ξ, =)$ to $Θ[\lceil \frac{ξ + e}{2} \rceil + 1 : e][j]$ ($Θ[\lceil \frac{ξ + e}{2} \rceil + 1 : e][j]$ is dropped if $\lceil \frac{ξ + e}{2} \rceil + 1 > e$). Similar to Case 1, we derive that $\frac{1}{5}$ of the current search space is pruned, and the corresponding shrinking operation is executed on $T'$.

In conclusion, using a pivot we can prune at least $\frac{1}{5}$ of the current search space.

To illustrate, consider our running example where Table 5 shows the array set corresponding to the current search space $R$ containing 49 values. Suppose that we have found a pivot $ξ = 1.249$. 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 $R$ by updating each array $Θ[s : e][j]$ in $A(ξ, -) \cup A(ξ, =)$ to be $Θ[s : \lfloor \frac{ξ + e}{2} \rfloor - 1][j]$. As discussed before, $A(ξ, -) = \{Θ[1 : 1][1], Θ[2 : 7][1], Θ[1 : 2][2], Θ[3][3], Θ[4 : 7][3], Θ[1 : 4][4]\}$ and $A(ξ, =) = Θ[3][7][2]$. 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 $14 > \frac{|S|}{2}$).

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 $Θ[s : e][j]$, 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 $\frac{1}{5}$ of the search space is pruned at each iteration.

### 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 $\frac{1}{5}$ 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 records the outdoor movements of 182 users in a period of 5 years and T-Drive 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 select 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

### Table 8: Real datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of trajectories</th>
<th>total # of positions</th>
<th>average # of positions per trajectory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geolife</td>
<td>17,621</td>
<td>24,876,978</td>
<td>1,412</td>
</tr>
<tr>
<td>T-Drive</td>
<td>10,359</td>
<td>17,740,902</td>
<td>1,713</td>
</tr>
</tbody>
</table>

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 $\gamma'$ in the original trajectory and the direction of the segment that approximates $\gamma'$ 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**

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 simplified 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., $W = 0.2 \times |T|$). 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/memory complexities than $DP$.

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

**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/memory complexities independent of $W$.

**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.

**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 $e(T')/e(T)$, where $T'$ is the simplified trajectory returned by this approximate algorithm on a given raw trajectory and
\( T'_e \) 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.

**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.

**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.

**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.

**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 \([8, 27, 31, 29, 18]\). 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 \([27, 31, 29, 18]\), 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 Raymon Chi-Wing Wong is supported by the grant FGSRF14EG34, and the research of H. V. Jagadish is supported in part by NSF grant IIS 1250880.

8. REFERENCES