Direction-Preserving Trajectory Simplification

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

Trajectories of moving objects are collected in many applications. Raw trajectory data is typically very large, and has to be simplified before use. In this paper, we introduce the notion of direction-preserving trajectory simplification, and show both analytically and empirically that it can support a broader range of applications than traditional position-preserving trajectory simplification. We present a polynomial-time algorithm for optimal direction-preserving simplification, and another approximate algorithm with a quality guarantee. Extensive experimental evaluation with real trajectory data shows the benefit of the new techniques.

1. INTRODUCTION

With the proliferation of GPS-embedded devices (e.g., smart phones and taxis), trajectory data is becoming ubiquitous. Indeed, it has been studied extensively in the past decades in the literature of Moving Objects Databases (MOD) [22, 19].Trajectory data is usually generated by periodically collecting the position of a moving object with the help of the GPS technologies.

Since the raw trajectory data is usually very large, simplifying trajectory data is important. To appreciate this, consider a city with 10k taxis. Suppose that we track the trajectory of each taxi by sampling its position once every 5 seconds (i.e., the sampling rate is 5s). The size of the collected trajectories for just one day is approximately 4 GB.

Raw trajectory data is large, and hence expensive to store. Even worse, it is expensive to manipulate and to analyze on account of its large size. In fact, most existing query processing and data mining algorithms on trajectory data are memory-resident and thus cannot be used with raw trajectory data that is too large to fit in memory.

A question one may ask is why not just sampling less frequently to reduce the size of the data. The answer is that, in real life, objects have great variance in their velocities. A taxi moving at 40 mph would have moved about 100 yards in 5s, whereas another taxi stuck at a traffic signal may not have moved at all. Obviously, we need more frequent observations of the former than of the latter. Similarly, we need more observations to capture a taxi that makes a turn and fewer for one that continues straight. Therefore, standard practice is to oversample initially, and then to simplify by eliminating observations that add little information.

In view of this, several algorithms have been developed for simplifying trajectory data [17, 6, 21, 18, 11]. All these algorithms make the natural assumption that the goal should be to simplify trajectories such that the *position information* captured in the simplified trajectories is "similar" to the position information captured in the original trajectories. We can call them *position-preserving trajectory simplification algorithms*. However, as we will soon see, this objective, though natural, is not the best choice in many situations. To illustrate, let us work through a toy example in detail.

EXAMPLE 1 (MOTIVATING EXAMPLE). Consider three raw trajectories T_1, T_2 and T_3 as shown in bold lines in Figure 1(a)(i), (a)(ii) and (a)(iii), respectively. Each of these trajectories has *four* positions, p_1, p_2, p_3 and p_4. T_1 and T_2 are *similar* to each other, and each of them is *dissimilar* to T_3. Thus, a trajectory clustering algorithm, such as [15], should group T_1 and T_2 in the same cluster and place T_3 by itself in a separate cluster.

Now suppose that these raw trajectories are too large, and so must be simplified to three points each before being further processed. We could use an existing position-preserving trajectory simplification, denoted by $\mathcal{A}_{pos}$, for this simplification. Following existing studies, the first position p_1 and the last position p_4 in each trajectory have to be kept. Therefore, one of position p_2 and position p_3 is to be retained, and the other one dropped.

Consider the simplification process on T_1. It can drop either p_2 or p_3 in the simplified trajectory. Let d_1 (d_2) be p_2's (p_3)'s perpendicular distance to line segment $\overline{p_1p_3}$ ($\overline{p_2p_4}$). Since $d_1 > d_2$, $\mathcal{A}_{pos}$ drops p_3 and returns the simplified trajectory T_1' as shown in Figure 1(b)(i). Similarly, $\mathcal{A}_{pos}$ return T_2' (Figure 1(b)(ii)) and T_3' (Figure 1(b)(iii)) as the simplified trajectories of T_2 and T_3, respectively. We now see that, though raw trajectories T_1 and T_2 are *similar*, their simplified trajectories T_1' and T_2' generated by $\mathcal{A}_{pos}$ are *dissimilar*. On the other hand, raw trajectories T_1 and T_3 are *dissimilar*, but their simplified trajectories T_1' and T_3' generated by $\mathcal{A}_{pos}$ are *similar*. In consequence, the clustering algorithm on the simplified trajectories T_1', T_2' and T_3', places T_1 and T_3 together into one cluster, and thus fails to produce correct (or expected) clusters.

In contrast, as will be shown later, a *direction-preserving* trajectory simplification method we introduce below, denoted by $\mathcal{A}_{dir}$, would simplify T_1, T_2 and T_3 to T_1'' (Figure 1(c)(i)), T_2'' (Figure 1(c)(ii)) and T_3'' (Figure 1(c)(iii)), respectively. Since T_1'' and T_2'' are similar to each other and each of them is dissimilar to T_3'', the clustering algorithm based on these simplified trajectories would produce the expected clusters. □

Before we can discuss direction-preserving trajectory simplification in depth, we first have to describe what direction information is, which we do next.

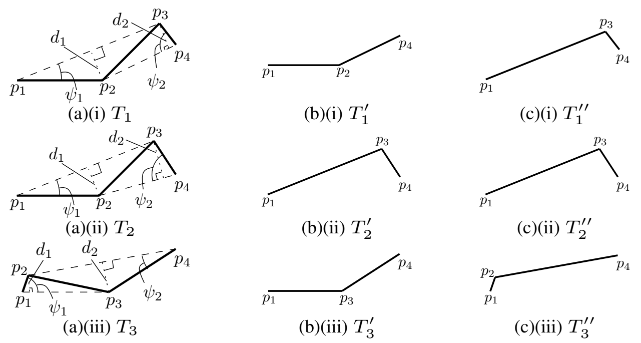


Figure 1: A motivating example

1.1 Direction Information

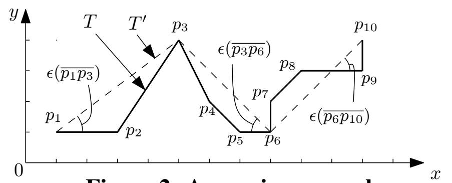
When an object moves from position p to position p', we define the *direction* of this movement to be the angle of an anticlockwise rotation from the positive x-axis to a vector from p to p'. The directions of all movements captured in the trajectory is called the *direction information*, and is used heavily, both directly and indirectly, in a wide range of applications on trajectory data. We list some of them as follows.

- *Map Matching* [1]. Given a digital map of a road network and a trajectory of an object moving on the road network, the map matching problem is to locate the trajectory on the digital map. Since each road segment in the road network has its own *orientation*, restricting the *directions* of the movements in the trajectory, the direction information plays an essential role in most map matching algorithms [1].
- *Knowledge Discovery on Trajectory Data*. As with other types of data, a rich set of knowledge discovery tasks has been proposed on the trajectory data [7]. Among them, many algorithms rely heavily on the direction information, which include [15, 9] for *Clustering*, [13] for *Outlier Detection* and [14] for *Classification*.
- *Direction-based Query Processing*. Sometimes, there are reasons to query trajectory information directly. One example is to find the trajectories moving within a direction range in a given time slot [2]. Another example is to find trajectories *similar* to a given trajectory, where the similarity measurement is based solely on moving direction [20].

In short, there are many situations in which direction preservation is important. Furthermore, as we show analytically in Section 3 and empirically in Section 6, direction preservation is stronger than position preservation, in that a simplification that preserves direction information well can be shown to preserve position information also, within some reasonable bounds. However, the converse is not true: position-preserving simplifications can be very bad at direction preservation.

1.2 Direction-Preserving Trajectory Simplification (DPTS)

In this paper, we propose a new trajectory simplification mechanism called *Direction-Preserving Trajectory Simplification* (DPTS) such that the direction information loss due to the simplification process is bounded. Within DPTS, we propose a *direction-based measurement* E_d, which is new and is defined to measure the *error* of a simplified trajectory in terms of the direction information. Let T be a trajectory and T' be a simplification of T. The *error* (or *simplification error*) of T' under E_d, denoted by $\epsilon(T')$, is equal to the maximum *angular difference* between the direction of the movement during each time period in T and the direction of the movement during the same time period in T'. Then, the problem of DPTS is to simplify a given trajectory such that its *size* is minimized and its incurred simplification error (i.e., $\epsilon(T')$) is bounded by a given error tolerance ϵ_t where $\epsilon_t \in [0, \pi)$.



y
T
T'
p_3
p_{10}
$\epsilon(\overline{p_3p_6})$
$\epsilon(\overline{p_1p_3})$
p_8
p_9
p_7
p_1
p_2
p_4
p_5
p_6
$\epsilon(\overline{p_6p_{10}})$
0
x

Figure 2: A running example

We use the *maximum* angular difference rather than the average angular difference to preserve better the shape of the trajectory. If we used the average one, we could still have a few segments that were completely off, resulting in the types of errors illustrated in Figure 1 for position-preserving techniques.

In this paper, we study the properties of DPTS, develop multiple algorithms to solve the DPTS problem, both exactly and approximately, and evaluate our algorithms experimentally. Specifically, we make the following contributions.

Contribution. First, we propose a novel notion of direction-preserving trajectory simplification, which favors a wide spectrum of applications on trajectory data. Second, we show that DPTS not only preserves direction information, but also preserves position information, thereby supporting a wide range of applications. Third, we adopt a common dynamic programming (DP) technique for DPTS. Since it is not scalable, we propose a novel optimal algorithm called *SP* for DPTS. *SP* solves DPTS by first constructing a graph based on the given trajectory, then computing a shortest path in this graph and finally returning the solution for DPTS according to the shortest path found. The time complexity of *SP* is $O(C \cdot n^2)$, where C is usually a small constant ($C = 1$ if $\epsilon_t \leq \pi/2$). Fourth, since even an $O(n^2)$ running time is likely to be unacceptable for a large n, we propose a scalable approximate algorithm called *Intersect* which runs in $O(n)$ time. We show that *Intersect* provides a certain degree of the quality guarantee in terms of the size of the simplified trajectory returned, in spite of running so fast. Finally, we perform a careful experimental comparison of these algorithms and a baseline using real trajectory data. The baseline is developed by common sense modifications of standard trajectory simplification techniques to address the DPTS problem.

The remainder of this paper is organized as follows. We define the DPTS problem in Section 2 and review the related work in Section 3. We introduce the optimal and approximate algorithms of DPTS in Section 4 and Section 5, respectively. We give the empirical study in Section 6 and conclude the paper in Section 7.

2. PROBLEM DEFINITION

A trajectory is represented by a sequence of n triplets in the form of $((x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n))$, where (x_i, y_i) is the position in the 2D Euclidean space at time stamp t_i. We define *positions* $p_i = (x_i, y_i)$ for each $i \in [1, n]$. Then, T's *trace* is the sequence of ordered positions, i.e., $(p_1, p_2, ..., p_n)$.

Since the direction information of a trajectory is captured by its trace only, in the following, following existing studies, we focus on the trace part of the trajectory and use the terms "trajectory" and "trace" interchangeably. Thus, we simply denote T by $(p_1, p_2, ..., p_n)$ by keeping the position information only. The *size* of T, denoted by $|T|$, is defined to be the number of positions in T.

Consider a running example as shown in Figure 2. In this figure, the trajectory T is represented in the form of $(p_1, p_2, ..., p_{10})$. The

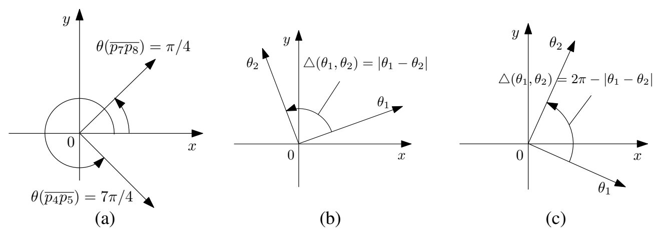

y
$\theta(\overline{p_7p_8}) = \pi/4$
0
x
$\theta(\overline{p_4p_5}) = 7\pi/4$
(a)
y
θ_2
$\triangle(\theta_1,\theta_2) = |\theta_1 - \theta_2|$
θ_1
0
x
(b)
y
θ_2
$\triangle(\theta_1,\theta_2) = 2\pi - |\theta_1 - \theta_2|$
0
x
θ_1
(c)

Figure 3: Examples illustrating the definition of "direction" and "angular difference"

size of this trajectory (i.e., $|T|$) is 10. The start position of T is p_1 and the end position of T is p_{10}.

The straight line linking two positions p_i and p_j in T where $1 \leq i < j \leq n$ is denoted by $\overline{p_ip_j}$. If p_i and p_j are *adjacent* in T (i.e., $j = i+1$), then $\overline{p_ip_j}$ is said to be a *segment* in T. Thus, a trajectory could also be regarded as a sequence of $n-1$ segments joining at $n-2$ positions (in addition to unique start and end positions).

In Figure 2, the solid horizontal straight line connecting p_1 and p_2 is denoted by $\overline{p_1p_2}$. Similarly, the dashed inclined straight line connecting p_1 and p_3 is denoted by $\overline{p_1p_3}$. Here, $\overline{p_1p_2}$ is a segment in T but $\overline{p_1p_3}$ is not a segment in T. All segments in T are shown in solid lines in the figure. In T, there are 9 segments jointing at 8 positions, namely $p_2, p_3, ..., p_9$.

Trajectory T' is said to be a *simplification* of T if T' is of the form of $(p_{s_1}, p_{s_2}, ..., p_{s_m})$ where $m \leq n$ and $1 = s_1 < s_2 < ... < s_m = n$. Note that p_1 and p_n in T must be kept in any simplification of T. There are $m-1$ segments in T', and T' is using $m-1$ segments to represent T containing $n-1$ segments. For each $k \in [1, m)$, the segment $\overline{p_{s_k}p_{s_{k+1}}}$ in T' is used to *approximate* the sequence of segments between p_{s_k} and $p_{s_{k+1}}$ in T, namely $\overline{p_{s_k}p_{s_k+1}}$, $\overline{p_{s_k+1}p_{s_k+2}}$, ..., $\overline{p_{s_{k+1}-1}p_{s_{k+1}}}$. In other words, this sequence of segments in T is approximated by a single segment (i.e., $\overline{p_{s_k}p_{s_{k+1}}}$) in T' only.

Consider our running example. Let $T' = (p_1, p_3, p_6, p_{10})$. T' is a simplification of T in Figure 2. Here, $s_1 = 1, s_2 = 3, s_3 = 6$ and $s_4 = 10$. Note that the size of T' is 4. All segments in T' are shown in dashed lines in the figure. There are 3 segments in T'. In other words, T' is using 3 segments to approximate 9 segments in T. Consider segment $\overline{p_1p_3}$ in T'. It is used to approximate the sequence of segments between p_1 and p_3 in T, namely $\overline{p_1p_2}$ and $\overline{p_2p_3}$. In other words, $\overline{p_1p_2}$ and $\overline{p_2p_3}$ are approximated by a single segment $\overline{p_1p_3}$. Similarly, trajectory $T'' = (p_1, p_{10})$ is also a simplification of T, which uses only one segment (i.e., $\overline{p_1p_{10}}$) to approximate the whole trajectory T.

Direction-based Error Measurement E_d. Given a segment $\overline{p_ip_{i+1}}$ in T, the *direction* of $\overline{p_ip_{i+1}}$, denoted by $\theta(\overline{p_ip_{i+1}})$, is defined to be the angle of an *anticlockwise* rotation from the positive x-axis to a vector from p_i to p_{i+1}. Thus, each direction falls in $[0, 2\pi)$. Consider our running example (Figure 2). $\theta(\overline{p_7p_8})$ is $\pi/4(= 0.788)$ radian and $\theta(\overline{p_4p_5})$ is $7\pi/4(= 5.498)$ radian, as illustrated in Figure 3(a). It is easy to verify that $\theta(\overline{p_1p_2})$ is equal to 0 radian, $\theta(\overline{p_2p_3})$ is equal to 0.983 radian ($= \tan^{-1} 3/2$) and $\theta(\overline{p_1p_3})$ is equal to 0.644 radian ($= \tan^{-1} 3/4$).

The *angular difference* between two directions θ_1 and θ_2, denoted by $\triangle(\theta_1, \theta_2)$, is defined to be the minimum of the angle of the anticlockwise rotation from θ_1 to θ_2 and that from θ_2 to θ_1, i.e.,

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

For illustration, Figure 3(b) shows the case where $\triangle(\theta_1, \theta_2) = |\theta_1 - \theta_2|$ and Figure 3(c) shows the case where $\triangle(\theta_1, \theta_2) = 2\pi - |\theta_1 - \theta_2|$. Note that the angular difference between any two directions falls in $[0, \pi]$.

Consider our running example. The angular difference between $\theta(\overline{p_1p_2})$ and $\theta(\overline{p_1p_3})$ is $|0 - 0.644| = 0.644$ and that between $\theta(\overline{p_2p_3})$ and $\theta(\overline{p_1p_3})$ is $|0.983 - 0.644| = 0.339$.

Let $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be a simplification of T The *simplification error* of T' under E_d, denoted by $\epsilon(T')$, is defined as follows. Given a segment $\overline{p_{s_k}p_{s_{k+1}}}$ in T', the *simplification error* of $\overline{p_{s_k}p_{s_{k+1}}}$, 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 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' under E_d is defined to be the *greatest* simplification error of a segment in T'. That is,

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

Consider back our running example (Figure 2). Each segment in T' has its simplification error. Consider the first segment $\overline{p_1p_3}$ in T' which approximates two segments in T, namely $\overline{p_1p_2}$ and $\overline{p_2p_3}$. Recall that $\triangle(\theta(\overline{p_1p_3}), \theta(\overline{p_1p_2})) = 0.644$ and $\triangle(\theta(\overline{p_1p_3}), \theta(\overline{p_2p_3})) = 0.339$. Thus, the simplification error of $\overline{p_1p_3}$ (i.e., $\epsilon(\overline{p_1p_3})$) is equal to $\max\{0.644, 0.339\} = 0.644$. Similarly, we compute the simplification errors of the second segment $\overline{p_3p_6}$ and the third segment $\overline{p_6p_{10}}$ in T' which are both equal to 0.785. Thus, the simplification error of T' in this example is equal to $\max\{0.644, 0.785, 0.785\} = 0.785$.

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 line segments between p_i and p_j in T.

Problem Statement of DPTS. Let T be a trajectory and ϵ_t be the error tolerance ($\epsilon_t < \pi$). Trajectory T' is said to be an ϵ_t-*simplification* of T if T' is a simplification of T and $\epsilon(T') \leq \epsilon_t$.

The DPTS problem is formalized as follows.

PROBLEM 1 (DPTS). *Given a trajectory T and an error tolerance ϵ_t, **the DPTS problem** is to find the ϵ_t-simplification of T with the smallest size.* □

Consider our running example. Suppose that we set ϵ_t to 0.785. T' is an ϵ_t-simplification of T since $\epsilon(T') = 0.785 \leq \epsilon_t$. In fact, T' is the ϵ_t-simplification of T with the smallest size (which involves only four remaining positions).

3. ANALYSIS OF PREVIOUS WORK

We describe how DPTS relates to existing error measurements (Section 3.1) and trajectory simplification techniques (Section 3.2).

3.1 Existing Error Measurements

In this section, we show that the direction-preserving simplified trajectories give error guarantees in position-related properties, such as length and speed. However, the reverse is not true. That is, the position-preserving simplified trajectories [17, 18, 21, 11, 6] do not give any error guarantee on the direction information.

Before we give our claims/properties (Section 3.1.2), we review some representative existing error measurements (Section 3.1.1).

3.1.1 Existing Error Measurements

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 $(m \leq n)$. Several *position-based* measurements for evaluating the "simplification error" of T' have been defined in the literature. These measurements for T' are usually defined to be a *distance measure* which takes T and T' as input. For each position p_h of T at the time stamp equal to t_h where $1 \leq h \leq n$, the distance measure defines the *estimated* *position* of p_h, denoted by p'_h, in T' based on some criteria. Let $d(\cdot,\cdot)$ be the Euclidean distance between two given points. Thus, the distance measure is defined to be $\max_{h\in[1,n]} d(p_h, p'_h)$. Since different distance measures have different definitions on estimated positions, in the following, we focus on how to define estimated positions for some representative distance measures.

(1) *Closest Euclidean Distance:* With this distance measure, p'_h is defined to be the location on the segment $\overline{p_{s_k}p_{s_{k+1}}}$ of T' with $s_k \le h \le s_{k+1}$, which has the smallest Euclidean distance from p_h. We define a mapping function $M_{\mathcal{C}}$ which maps p_h and T' to p'_h for this distance measure. That is, $p'_h = M_{\mathcal{C}}(p_h, T')$.

(2) *Synchronous Euclidean Distance:* Under this distance measure, p'_h can be calculated with the following steps. The first step is to find the segment $\overline{p_{s_k}p_{s_{k+1}}}$ of T' with $s_k \le h \le s_{k+1}$. The second step is to find a point along the line passing through two points, namely $(x_{s_k}, y_{s_k}, t_{s_k})$ and $(x_{s_{k+1}}, y_{s_{k+1}}, t_{s_{k+1}})$, in a three-dimensional space such that the third dimensional value (representing the time dimension) of this point is t_h. Then, p'_h is set to be the first two-dimensional values of this point. Similarly, we define a mapping function $M_{\mathcal{S}}$ which maps both p_h and T' to p'_h. That is, $p'_h = M_{\mathcal{S}}(p_h, T')$.

However, all of them adopt *position-based* distances instead of the *direction-based* distance studied in this paper. In the next section, we show that they do not give any error guarantee on the direction information.

3.1.2 Theoretical Properties

The *length* between two positions p_i and p_j wrt T ($i < j$), denoted by $len(p_i, p_j|T)$, is defined to be the length of the trace from p_i to p_j in T. That is,

$$len(p_i, p_j|T) = \textstyle\sum_{k=i}^{j-1} d(p_k, p_{k+1})$$

The (average) *speed* between two positions p_i and p_j wrt T ($i < j$), denoted by $speed(p_i, p_j|T)$, is equal to $len(p_i, p_j|T)/(t_j - t_i)$, where t_i (t_j) is the time stamp corresponding to p_i (p_j).

Interestingly, DPTS gives error guarantees on the length and speed information. Consider that T is a trajectory and T' is an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For any two adjacent positions p_i and p_{i+1} in T where $i \in [1, n)$, both the length and the speed between the two corresponding *estimated* positions wrt T' are theoretically bounded. The estimated positions are determined by a mapping function. For the sake of space, in the remaining of the paper, if we do not specify the distance measure, we mean that we adopt the mapping function used in the Closest Euclidean Distance (i.e., $M_{\mathcal{C}}(\cdot,\cdot)$). The results based on the other mapping function (i.e., $M_{\mathcal{S}}(\cdot,\cdot)$) can be found in [16].

LEMMA 1 (BOUNDED LENGTH/SPEED). *Let T be a trajectory and T' be an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For any two adjacent positions p_i and p_{i+1} in T where $i \in [1, n)$,*

$$\cos(\epsilon_t) \le \tfrac{len(p'_i, p'_{i+1}|T')}{len(p_i, p_{i+1}|T)} \le 1 \; and \; \cos(\epsilon_t) \le \tfrac{speed(p'_i, p'_{i+1}|T')}{speed(p_i, p_{i+1}|T)} \le 1$$

where $p'_i = M_{\mathcal{C}}(p_i, T')$ and $p'_{i+1} = M_{\mathcal{C}}(p_{i+1}, T')$. □

PROOF. Let $\overline{p_{s_k}p_{s_{k+1}}}$ be the segment of T' such that p_{s_k} is the last position with $s_k \le i$ and $p_{s_{k+1}}$ is the first position with $s_{k+1} \ge i+1$. Consider Figure 4(a) for illustration. Since $\epsilon_t < \pi/2$, we can verify that p'_i and p'_{i+1} are located along $\overline{p_{s_k}p_{s_{k+1}}}$.

Let ψ_i be the angle formed by the two lines that pass through $\overline{p_i p_{i+1}}$ and $\overline{p_{s_k}p_{s_{k+1}}}$. Thus,

$$len(p'_i, p'_{i+1}|T') = d(p'_i, p'_{i+1}) = \cos(\psi_i)\cdot d(p_i, p_{i+1})$$
$$= \cos(\psi_i)\cdot len(p_i, p_{i+1}|T) \ge \cos(\epsilon_t)\cdot len(p_i, p_{i+1}|T)$$

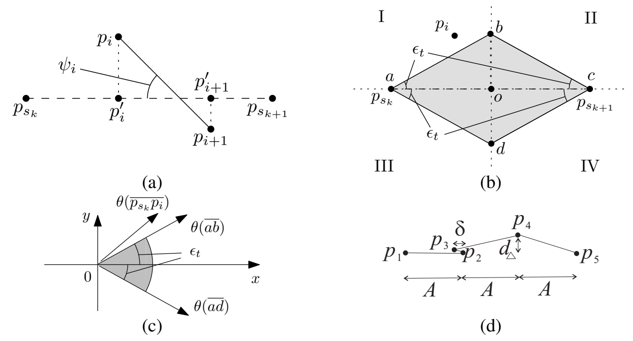


Figure 4: Proofs of Lemma 1, Lemma 2 and Lemma 3

which implies that $\cos(\epsilon_t) \le \frac{len(p'_i, p'_{i+1}|T')}{len(p_i, p_{i+1}|T)} \le 1$. Besides, since $speed(p'_i, p'_{i+1}|T') = len(p'_i, p'_{i+1}|T')/(t_{i+1} - t_i)$ and $speed(p_i, p_{i+1}|T) = len(p_i, p_{i+1}|T)/(t_{i+1} - t_i)$, we know $\cos(\epsilon_t) \le \frac{speed(p'_i, p'_{i+1}|T')}{speed(p_i, p_{i+1}|T)} \le 1$. □

Interestingly, DPTS gives an error bound on the position information (in addition to the length/speed information).

LEMMA 2 (BOUNDED POSITION ERROR). *Let T be a trajectory and T' be an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For each position p_i in T where $i \in [1, n]$, we have*

$$d(p_i, p'_i) \le 0.5\cdot\tan(\epsilon_t)\cdot L_{max}$$

where $p'_i = M_{\mathcal{C}}(p_i, T')$ and $L_{max} = \max_{k\in[1,m)} len(p_{s_k}, p_{s_{k+1}}|T')$. □

PROOF. Let $\overline{p_{s_k}p_{s_{k+1}}}$ be the segment of T' such that p_{s_k} is the last position with $p_{s_k} \le i$ and $p_{s_{k+1}}$ is the first position with $p_{s_{k+1}} \ge i$. We construct a rhombus $\diamond_{abcd}$ with four corners, namely a, b, c and d, such that a is at p_{s_k}, c is at $p_{s_{k+1}}$ and the angle between $\overline{ab}$ ($\overline{cb}$) and $\overline{ad}$ ($\overline{cd}$) is equal to $2\cdot\epsilon_t$. Consider Figure 4(b) for illustration where $\diamond_{abcd}$ is indicated by the shaded area. We claim that p_i is inside $\diamond_{abcd}$ which we prove by contradiction.

Assume that p_i is outside $\diamond_{abcd}$. We partition the plane into 4 parts with the two lines that pass through $\overline{ac}$ and $\overline{bd}$ as indicated by I, II, III and IV in Figure 4(b), where o is the intersection of the two lines. We consider 4 cases of which partition p_i is in. Without loss of generality, suppose p_i falls in part I.

Since p_i is outside $\diamond_{abcd}$, we know $\theta(\overline{p_{s_k}, p_i})$ falls outside range $[\theta(\overline{ad}), \theta(\overline{ab})]$. For illustration, consider Figure 4(c).

Consider the segments between p_{s_k} and p_i in T. For each such segment $\overline{p_h p_{h+1}}$ ($s_k \le h < i$), we denote by $\overrightarrow{p_h p_{h+1}}$ the vector from p_h to p_{h+1}. We know that the direction of each such vector falls in range $[\theta(\overline{ad}), \theta(\overline{ab})]$ since otherwise $\epsilon(T') > \epsilon_t$. As a result, we know $\theta(\overline{p_{s_k}p_i})$ falls in range $[\theta(\overline{ad}), \theta(\overline{ab})]$ since $\overrightarrow{p_{s_k}p_i} = \sum_{s_k \le h < i} \overrightarrow{p_h p_{h+1}}$. This, however, contradicts the fact that $\theta(\overline{p_{s_k}p_i})$ falls outside range $[\theta(\overline{ad}), \theta(\overline{ab})]$.

Thus, we know that p_i falls in $\diamond_{abcd}$. Therefore, we have

$$d(p_i, p'_i) \le d(b, o) = \tan(\epsilon_t)\cdot d(a, o) = 0.5\cdot\tan(\epsilon_t)\cdot d(a, c)$$
$$= 0.5\cdot\tan(\epsilon_t)\cdot len(p_{s_k}, p_{s_{k+1}}|T') \le 0.5\cdot\tan(\epsilon_t)\cdot L_{max}$$

which finishes the proof. □

Next, we show that existing position-preserving simplified trajectories do not have bounds on the direction information.

LEMMA 3 (UNBOUNDED DIRECTION ERROR). *Let T be a trajectory and T' be a (direction-based) ϵ_t-simplification of T with $\epsilon_t < \pi/2$. Let $T_{\mathcal{C}}$ be a (position-based) simplified trajectory of*

T such that $|T_{\mathbb{C}}| = |T'|$ and the error of $T_{\mathbb{C}}$ under the Closest Euclidean Distance is minimized. There exists a dataset such that $\epsilon(T_{\mathbb{C}}) \approx \pi$ and $\epsilon(T') \approx 0$. □

PROOF. We prove by constructing a problem instance as shown in Figure 4(d). $T = (p_1, p_2, p_3, p_4, p_5)$ is a trajectory, where p_1, p_2, p_3 and p_5 are located at a horizontal line and p_4 has its perpendicular distance from this line equal to a small real number $d_{\triangle}$. Besides, $d(p_2, p_3) = \delta$ where $\delta << d_{\triangle}$.

Suppose that we can only keep 4 positions in the simplified trajectory. In other words, we have to remove 1 position from the 5 positions. If we consider preserving the direction information, p_4 will be removed and thus $T' = (p_1, p_2, p_3, p_5)$. Thus, $\epsilon(T') = \epsilon(\overline{p_3p_5}) \approx 0$. If we consider preserving the position information, p_2 will be removed and thus $T_{\mathbb{C}} = (p_1, p_3, p_4, p_5)$. Hence, $\epsilon(T_{\mathbb{C}}) = \epsilon(\overline{p_1p_3}) = \triangle(\theta(\overline{p_1p_3}), \theta(\overline{p_2p_3})) \approx \pi$. □

3.2 Existing Trajectory Simplification

Many trajectory simplification techniques have been proposed. We categorize them by the main idea employed in the algorithm as follows. They are *Split* [17, 6], *Merge* [21, 18], *Greedy* [11, 17] and *Dead-Reckoning* [12].*Split* is an approach which finds a position in a given trajectory, according to the *heuristic* value of the position, to *split* the whole trajectory into two sub-trajectories and continues the process iteratively on each of the split sub-trajectories which cannot be approximated by a line segment connecting its start position and its end position. *Merge* is an approach which finds two adjacent segments in a given trajectory, according to the *heuristic* value computed from these two adjacent segments, discards the position p bridging these two segments, and create a segment connecting the non-bridging end position of one segment and the non-bridging end position of the other segment. It continues the process iteratively until discarding any position p violates the error tolerance. *Greedy* is an approach which finds a sequence of the greatest number of consecutive segments to be discarded and create a segment connecting the two end positions of this sequence iteratively until discarding any sequence of 2 consecutive segments violates the tolerance constraint. *Dead-Reckoning* is an online algorithm which reads each position sequentially and determines whether this position is discarded or not according to a *heuristic* criterion.

The aforementioned ideas of *Split*, *Merger* and *Greedy* can be adapted to our DPTS problem. The only change is to change the error measurement to our simplification error (Equation (2)). However, they have their drawbacks. First, they cannot return *optimal* solutions. Second, as shown in our experiments, they are not efficient compared with our proposed *Intersect* algorithm. Details of the adaptation can be found in our technical report [16].

Other related studies include [3] which studies the error bounds of several queries on the simplified trajectories with bounded simplification errors mainly measured by the position information, [5] which studies the trajectory simplification problem with the consideration of the shape and also the semantic meanings of the trajectory, [4] which introduces a multi-resolution *polygonal curve approximation* (also called *line simplification*) algorithm for trajectory simplification, and [10] which studies the trajectory simplification problem where the trajectories are constrained to a road network. None of these studies pay attention to the direction information for trajectory simplification.

4. FINDING OPTIMAL SOLUTION

A naive solution for the DPTS problem is to traverse each possible simplification of T with its simplification error at most ϵ_t and then to pick the one with the smallest size. Since the number of all

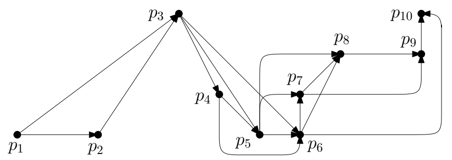


Figure 5: The graph G_{ϵ_t} constructed based on the running example when ϵ_t is set to be $\pi/4 = 0.785$

possible simplifications of a trajectory T is $2^{|T|-2}$, this solution is not feasible in practice. Alternatively, one may adopt a common dynamic programming (DP) technique for the DPTS problem. Unfortunately, the time complexity of this technique is cubic. For the sake of space, we include this DP algorithm in [16]. Instead, we propose a method called *SP* which is much faster and scalable.

Algorithm *SP* involves the following three steps.

- **Step 1 (Graph Construction):** It first constructs a graph based on the given trajectory.
- **Step 2 (Shortest Path Finding):** It computes a shortest path in this graph.
- **Step 3 (Solution Generation):** It finally returns the solution for DPTS according to the shortest path found.

In Step 1, it constructs a graph wrt ϵ_t, denoted by $G_{\epsilon_t}(V, E)$, as follows. For each position p_i of T where $1 \le i \le n$, it creates a vertex for p_i in V. For each pair of two positions (p_i, p_j) where $i < j$, it creates an edge (p_i, p_j) in E if $\epsilon(\overline{p_ip_j}) \le \epsilon_t$.

In Step 2, it finds the shortest path from p_1 to p_n in G_{ϵ_t} by a shortest path algorithm (e.g., a BFS search). Here, the *length* of a path is defined to be the number of edges involved along the path.

In Step 3, it generates the solution for DPTS according to the shortest path found. Note that all vertices involved in this shortest path correspond to all positions in the ϵ_t-simplification of T with the smallest size. Thus, if the ordering of the positions (or vertices) involved in the shortest path is "p_{s_1}-p_{s_2}-...-p_{s_m}", it returns the solution T' as $(p_{s_1}, p_{s_2}, ..., p_{s_m})$.

EXAMPLE 2 (ALGORITHM SP). Consider our running example in Figure 2. Suppose that $\epsilon_t = 0.785$. In Step 1 of the *SP* algorithm, we can construct graph G_{ϵ_t} accordingly as shown in Figure 5. In this figure, we construct a vertex for each position in T. Besides, for each pair of positions p_i and p_j where $i < j$, if $\epsilon(\overline{p_ip_j}) \le \epsilon_t$, we create an edge (p_i, p_j). Note that $\epsilon(\overline{p_ip_{i+1}}) = 0$ for each $i \in [1, n-1]$.

In Step 2, we can find the shortest path in this graph. It is easy to verify that p_1-p_3-p_6-p_{10} is the shortest path. Finally, in Step 3, we construct the solution of DPTS as (p_1, p_3, p_6, p_{10}). □

Let us analyze the time complexity of a *straightforward* implementation of algorithm *SP*. For Step 1, a straightforward solution for constructing G_{ϵ_t} is to try all possible pairs of (p_i, p_j) where $1 \le i < j \le n$ and to check whether $\epsilon(\overline{p_ip_j}) \le \epsilon_t$. Since there are $O(n^2)$ possible such pairs and the checking cost for each pair is $O(n)$, the time complexity of Step 1 is $O(n^3)$. For Step 2, a simple BFS could be adopted to find the shortest path from p_1 to p_n in G_{ϵ_t}, which takes $O(|V| + |E|)$ time. Since $|V| = O(n)$ and $|E| = O(n^2)$, we know that the cost of BFS is $O(n^2)$. Step 3 which returns the solution takes $O(n)$ time. As we can see, the time complexity of Step 1 (i.e., the graph construction) dominates those of Step 2 and Step 3. Thus, the overall time complexity of a straightforward implementation of algorithm *SP* is $O(n^3)$. Besides, the space complexity of *SP* is simply $O(|V| + |E|)$ which corresponds to the space cost of maintaining G_{ϵ_t}.

In the following, we propose two kinds of enhancement techniques in order to improve the efficiency of our *SP* algorithm. The

Algorithm 1 The *SP* algorithm with the practical enhancement

```
Input: A trajectory $T = (p_1, p_2, ..., p_n)$ and the error tolerance $\epsilon_t$
 1: $H_0 \leftarrow \{p_1\}$; $U \leftarrow \{p_2, p_3, ..., p_n\}$; $l \leftarrow 1$
 2: while true do
 3:     $H_l \leftarrow \emptyset$
 4:     //process the positions in $H_{l-1}$ and $U$ in a reversed order
 5:     for each $p_i$ in $H_{l-1}$ and each $p_j$ in $U$ where $i < j$ do
 6:         if $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ then
 7:             if $p_j = p_n$ then
 8:                 return the trajectory corresponding to the shortest
                    path from $p_1$ to $p_n$
 9:             $U \leftarrow U \backslash \{p_j\}$; $H_l \leftarrow H_l \cup \{p_j\}$
10:     $l \leftarrow l + 1$
```

first one is called the *practical enhancement* (Section 4.1) which is to improve the performance of the algorithm in a practical way. The second one is called the *complexity improvement* (Section 4.2) which is to improve the theoretical time complexity of the algorithm from cubic to quadratic with some properties.

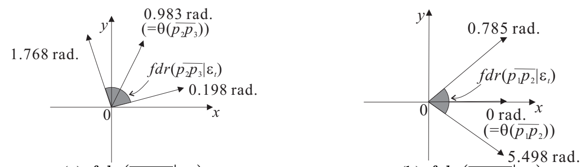
4.1 Practical Enhancement

The practical enhancement is to construct G_{ϵ_t} (in Step 1) *on the demand* of the BFS procedure used in Step 2. Specifically, the straightforward implementation has to materialize *all* possible edges in G_{ϵ_t} in Step 1 and then perform the BFS procedure in Step 2. Here, the enhancement constructs only *some* of the edges in G_{ϵ_t} which are needed in the BFS procedure. Since some other edges need not be constructed, the space consumption can be reduced and some computations can be also saved.

Given a position p in T and a non-negative integer l, p is said to be an *l-length position* if the length of the shortest path from p_1 to p in G_{ϵ_t} is equal to l. Given a non-negative integer l, we define the *l-length unique set*, denoted by H_l, to be the set of all l-length positions in T. For example, the 0-length unique set H_0 is $\{p_1\}$. Consider the BFS procedure starting from p_1 on G_{ϵ_t}. It first retrieves the set of positions which are the out-neighbors of p_1. This set corresponds to H_1. Then, starting from each position p in H_1, it retrieves the set of positions which are the out-neighbors of p and *have not been retrieved before*. This set corresponds to H_2. The above process continues from H_2 in the same manner until p_n is retrieved.

In view of the above discussion, we design our *SP* algorithm with this enhancement as follows. We maintain the l-length unique sets H_l ($l = 0, 1, 2, ...$) which store the positions retrieved by the BFS procedure and U for storing the remaining positions that have not been retrieved by the BFS procedure. We initialize H_0 to be $\{p_1\}$ and U to be $\{p_2, p_3, ..., p_n\}$. We then compute H_l based on H_{l-1} for $l = 1, 2, ...$ iteratively as follows. We start from each position p_i in H_{l-1}. For each position p_j in U, we compute $\epsilon(\overline{p_i p_j})$. If $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$, we further check whether p_j is p_n. If so, we stop the process since the shortest path from p_1 to p_n has been found; otherwise, we exclude p_j from U and include it in H_l. Besides, when processing the positions in H_{l-1} and U, we impose a *reversed* order, which corresponds to $p_n, p_{n-1}, ..., p_1$. The intuition is that we expect that p_n could be retrieved earlier in this way. We present our enhanced *SP* algorithm in Algorithm 1.

Complexity Analysis. The worst-case time complexity of the *SP* algorithm with the practical enhancement keeps the same as that of the straightforward implementation, i.e., it is still $O(n^3)$. However, in practice, with the practical enhancement, the *SP* algorithm is more efficient since some computations of $\epsilon(\overline{p_i p_j})$ are avoided, and it is also more scalable since there is no need to materialize G_{ϵ_t}. The space complexity of *SP* with the practical enhancement is simply $O(n)$ since it maintains each position once and does not materialize G_{ϵ_t} explicitly.



(a) $fdr(\overline{p_2 p_3}|\epsilon_t)$ (b) $fdr(\overline{p_1 p_2}|\epsilon_t)$

Figure 6: Illustration of $fdr(\overline{p_2 p_3}|\epsilon_t)$ and $fdr(\overline{p_1 p_2}|\epsilon_t)$

4.2 Complexity Improvement

The complexity improvement is to improve the time complexity of our *SP* algorithm from cubic to quadratic by using some properties for our algorithm. In this section, we focus on the complexity improvement based on the straightforward implementation for illustration. In Section 4.3, we describe how this complexity improvement can be incorporated with the practical enhancement.

As can be noticed, the cost of the straightforward implementation is dominated by the construction of graph G_{ϵ_t}. In this section, we propose a technique to reduce the cost of constructing the graph from $O(n^3)$ to $O(C \cdot n^2)$ time, where C is shown to be a small constant in most cases. The major idea of such an improvement is to reduce the time complexity of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ (in the graph construction step) from $O(n)$ to $O(C)$ by utilizing a new concept called "*feasible direction range*". Before we present the main idea, we first introduce some related concepts.

Given two angles θ_1 and θ_2 in [0, 2π), an *angular range*, represented in the form of $[\theta_1, \theta_2]$, is defined to be a set of all possible angles of a vector originated from the origin when it is rotated anti-clockwise from θ_1 to θ_2. For example, the shaded part in Figure 6(a) shows the angular range of $[0.198, 1.768]$, and the shaded part in Figure 6(b) shows the angular range of $[5.498, 0.785]$. Since the direction of $\overline{p_2 p_3}$ is 0.983 radian, we say that its direction is in $[0.198, 1.768]$ but not in $[5.498, 0.785]$. Similarly, since the direction of $\overline{p_1 p_2}$ is 0 radian, we say that its direction is in $[5.498, 0.785]$ but not in $[0.198, 1.768]$.

DEFINITION 1 (FEASIBLE DIRECTION RANGE). *Given a segment $\overline{p_h p_{h+1}}$ ($1 \leq h < n$) in T, the **feasible direction range** of $\overline{p_h p_{h+1}}$ wrt ϵ_t, denoted by $fdr(\overline{p_h p_{h+1}}|\epsilon_t)$, is defined to be the angular range in the form of $[\theta_1, \theta_2]$ with $\theta_1 = [(\theta(\overline{p_h p_{h+1}}) - \epsilon_t) \mod 2\pi]$ and $\theta_2 = [(\theta(\overline{p_h p_{h+1}}) + \epsilon_t) \mod 2\pi]$.* □

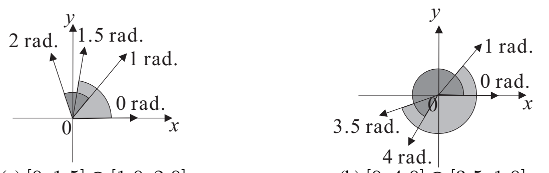
The feasible direction range of $\overline{p_h p_{h+1}}$ wrt ϵ_t corresponds to a set of all possible directions each of which has its angular difference from $\overline{p_h p_{h+1}}$ at most ϵ_t. We can write fdr as follows.

$$fdr(\overline{p_h p_{h+1}}|\epsilon_t) = [\theta(\overline{p_h p_{h+1}}) - \epsilon_t, \theta(\overline{p_h p_{h+1}}) + \epsilon_t] \mod 2\pi \tag{3}$$

Consider our running example. Suppose that ϵ_t is set to 0.785. Since the direction of $\overline{p_2 p_3}$ is 0.983 radian, $fdr(\overline{p_2 p_3}|\epsilon_t) = ([0.983 - 0.785, 0.983 + 0.785] \mod 2\pi) = [0.198, 1.768]$ (See Figure 6(a)). Similarly, since the direction of $\overline{p_1 p_2}$ is 0 radian, $fdr(\overline{p_1 p_2}|\epsilon_t) = ([0 - 0.785, 0 + 0.785] \mod 2\pi) = [5.498, 0.785]$ (See Figure 6(b)).

We denote by $T[i, j]$ the sub-trajectory of T that is between position p_i and position p_j ($1 \leq i < j \leq n$), i.e., $T[i, j] = (p_i, p_{i+1}, ..., p_j)$. We define the *feasible direction range* of a sub-trajectory $T[i, j]$ wrt ϵ_t, denoted by $fdr(T[i, j]|\epsilon_t)$, to be the *intersection* of the fdr's of the segments in $T[i, j]$. That is,

$$fdr(T[i, j]|\epsilon_t) = \cap_{i \leq h < j} fdr(\overline{p_h p_{h+1}}|\epsilon_t) \tag{4}$$



(a) $[0, 1.5] \cap [1.0, 2.0]$ (b) $[0, 4.0] \cap [3.5, 1.0]$

Figure 7: Illustration of intersection operations between two angular ranges

In our running example, $fdr(T[1,3]|\epsilon_r)$ is equal to $\cap_{1 \le h < 3} fdr(\overline{p_h p_{h+1}}|\epsilon_t) = fdr(\overline{p_1 p_2}|\epsilon_t) \cap fdr(\overline{p_2 p_3}|\epsilon_t) = [5.498, 0.785] \cap [0.198, 1.768] = [0.198, 0.785]$. Figures 6(a) and (b) illustrate this scenario.

In the following, we simply write $fdr(\overline{p_h p_{h+1}})$ $(fdr(T[i,j]))$ for $fdr(\overline{p_h p_{h+1}}|\epsilon_t)$ $(fdr(T[i,j]|\epsilon_t))$ if the context of ϵ_t is clear.

In some cases, an intersection between an angular range and another angular range results in a *single* angular range. In some other cases, this intersection operation results in *multiple* disjoint angular ranges. To illustrate, an intersection between $[0, 1.5]$ and $[1.0, 2.0]$ results in a single angular range $[1, 1.5]$ (as shown in Figure 7(a)). An intersection between $[0, 4.0]$ and $[3.5, 1.0]$ results in two angular ranges $[3.5, 4.0]$ and $[0, 1.0]$ (as shown in Figure 7(b)).

Thus, from Equation (4), since $fdr(T[i,j])$ involves multiple intersection operations of angular ranges, it may consist of multiple disjoint angular ranges. We denote by $||fdr(T[i,j])||$ the number of disjoint angular ranges in $fdr(T[i,j])$.

With the concept of "feasible direction range", we are now ready to describe how we can check whether $\epsilon(\overline{p_i p_j}) \le \epsilon_t$ efficiently.

LEMMA 4. *Let $T = (p_1, p_2, ..., p_n)$ be a trajectory. $\epsilon(\overline{p_i p_j}) \le \epsilon_t$ iff $\theta(\overline{p_i p_j})$ is in $fdr(T[i,j])$.* □

PROOF. "if": Assume that $\theta(\overline{p_i p_j})$ is in $fdr(T[i,j])$. It follows that $\theta(\overline{p_i p_j})$ is in $fdr(\overline{p_h p_{h+1}})$ and thus $\triangle(\theta(\overline{p_i p_j}), \theta(\overline{p_h p_{h+1}})) \le \epsilon_t$ for $i \le h < j$. Therefore, $\epsilon(\overline{p_i p_j}) = \max\{\triangle(\theta(\overline{p_h p_{h+1}}), \theta(\overline{p_i p_j}))|i \le h < j\} \le \epsilon_t$.

"only-if": this direction could be verified similarly. □

Lemma 4 suggests that checking whether $\epsilon(\overline{p_i p_j}) \le \epsilon_t$ is equivalent to checking whether $\theta(\overline{p_i p_j})$ is in $fdr(T[i,j])$. Suppose that $fdr(T[i,j])$ has been computed. Then, checking whether $\epsilon(\overline{p_i p_j}) \le \epsilon_t$ takes $O(||fdr(T[i,j])||)$ only (compared with $O(n)$ in the straightforward implementation).

Note that in some cases, $\epsilon(\overline{p_i p_j}) > \epsilon_t$ but $\epsilon(\overline{p_i p_{j+k}}) \le \epsilon_t$ where $j > i > 0$ and $k > 0$. By Lemma 4, we know that $\theta(\overline{p_i p_j})$ is not in $fdr(T[i,j])$ but $\theta(\overline{p_i p_{j+k}})$ is in $fdr(T[i, j+k])$. To illustrate, in our running example (Figure 2), if we set $\epsilon_t = \pi/4$, then $\epsilon(\overline{p_6 p_9}) > \epsilon_t$ but $\epsilon(\overline{p_6 p_{10}}) \le \epsilon_t$. By Lemma 4, $\theta(\overline{p_6 p_9})$ is not in $fdr(T[6,9])$ but $\theta(\overline{p_6 p_{10}})$ is in $fdr(T[6,10])$.

Now, we know that the checking step can be done in $O(||fdr(T[i,j])||)$. There are two remaining issues related to this checking step. The first issue is related to the size of $fdr(T[i,j])$. If this size is very large, the checking step is still expensive. Fortunately, we find that this size is usually a small constant. When $\epsilon_t \le \pi/2$, it is equal to 1. The second issue is how to compute $fdr(T[i,j])$ efficiently for each i and j where $i < j$.

4.2.1 Issue 1: Size of $fdr(T[i,j])$

LEMMA 5. *Let $T = (p_1, p_2, ..., p_n)$ be a trajectory and ϵ_t be the error tolerance. Then, given two integers i and j ($1 \le i < j \le n$), $||fdr(T[i,j])||$ is bounded by* $\min\{1 + \lfloor \frac{\epsilon_t}{(\pi - \epsilon_t)} \rfloor, j - i\}$. □

PROOF. We first give some concepts based on an angular range and provide a lemma which is used to prove Lemma 5.

We denote the *universe* angular range $[0, 2\pi)$ by $\mathcal{U}$, Given an angular range $[a, b]$, we denote its *complement* wrt $\mathcal{U}$, which is the angular range (b, a), by $[a, b]^c$. We define the *span* of an angular range $[a, b]$, denoted by $[a, b].span$, to be equal to $(b - a)($ mod $2\pi)$. We define the *span* of $fdr(T[i,j])$ $(1 \le i < j \le n)$, denoted by $fdr(T[i,j]).span$, to be equal to the sum of the spans of the disjoint angular ranges that are involved in $fdr(T[i,j])$.

LEMMA 6. *Let $[a, b]$ and $[a', b']$ be two angular ranges. $[a, b] \cap [a', b']$ involves two disjoint angular ranges iff $[a', b']^c$ (i.e., (b', a')) falls in $[a, b]$ completely.* □

PROOF. This could be verified easily by the fact that $[a, b] \cap [a', b']$ involves two disjoint angular ranges, namely $[a, b']$ and $[a', b]$, iff (b', a') falls in $[a, b]$ completely. □

Suppose $fdr(\overline{p_k p_{k+1}})$ is $[a_k, b_k]$ for $i \le k < j$. Then, $fdr(T[i,j])$ is equal to $\cap_{k=i}^{j-1} [a_k, b_k]$. Besides, we know $[a_k, b_k].span = 2\epsilon_t$ for $i \le k < j$ since $a_k = \theta(\overline{p_k p_{k+1}}) - \epsilon_t$ mod 2π and $b_k = \theta(\overline{p_k p_{k+1}}) + \epsilon_t$ mod 2π (Definition 1).

First, we prove $||fdr(T[i,j])|| \le j - i$ by induction of $k = j - i$. Base step: $k = 1$. The correctness is obvious since $fdr(T[i, i+1]) = [a_i, b_i]$ which involves one angular range only (i.e., $[a_i, b_i]$) and thus $||fdr(T[i, i+1])|| = 1$. Induction step: $||fdr(T[i, i+k)|| \le k$ implies $||fdr(T[i, i+k+1])|| \le k+1$. Assume $||fdr(T[i, i+k)|| = r$ $(r \le k)$. Specifically, let $[a'_1, b'_1], [a'_2, b'_2], ..., [a'_r, b'_r]$ be the r disjoint angular ranges involved in $fdr(T[i, i+k])$. Then, $fdr(T[i, i+k+1])$, which is equal to $fdr(T[i, i+k]) \cap [a_{i+k}, b_{i+k}]$, corresponds to r intersections, $[a'_h, b'_h] \cap [a_{i+k}, b_{i+k}]$ for $1 \le h \le r$, each two of which are disjoint. Among these r intersections, we show that at most one involves two disjoint angular ranges by contradiction. Assume that there exists h_1 and h_2 $(i \le h_1 \ne h_2 \le r)$ such that both $[a'_{h_1}, b'_{h_1}] \cap [a_{i+k}, b_{i+k}]$ and $[a'_{h_2}, b'_{h_2}] \cap [a_{i+k}, b_{i+k}]$ involve two disjoint angular ranges. According to Lemma 6, we know that (b_{i+k}, a_{i+k}) is in both $[a'_{h_1}, b'_{h_1}]$ and $[a'_{h_2}, b'_{h_2}]$, which leads to a contradiction since $[a'_{h_1}, b'_{h_1}]$ and $[a'_{h_2}, b'_{h_2}]$ are disjoint. As a result, $fdr(T[i, i+k_1])$ involves at most $r+1$ disjoint angular ranges. That is, $||fdr(T[i, i+k+1])|| \le r+1 \le k+1$.

Second, we prove $||fdr(T[i,j])|| \le 1 + \lfloor \frac{\epsilon_t}{\pi - \epsilon_t} \rfloor$. We compute $fdr(T[i, i+1]), fdr(T[i, i+2]), ..., fdr(T[i,j])$ sequentially based on the following equation.

$$\begin{aligned} fdr(T[i, i+k+1]) &= fdr(T[i, i+k]) \cap [a_{i+k}, b_{i+k}] \\ &= fdr(T[i, i+k]) \backslash (b_{i+k}, a_{i+k}) \end{aligned} \quad (5)$$

We have two cases regarding Equation 5. Case 1: $||fdr(T[i, i+k+1])|| = ||fdr(T[i, i+k])|| + 1$. In this case, according to Lemma 6, (b_{i+k}, a_{i+k}) is in one of the disjoint angular ranges that are involved in $fdr(T[i, i+k+1])$. Then, we deduce that $fdr(T[i, i+k+1]).span = fdr(T[i, i+k]).span - (2\pi - 2\epsilon_t)$ since $(a_{i+k} - b_{i+k})$ mod $2\pi = 2\pi - 2\epsilon_t$. Case 2: $||fdr(T[i, i+k+1)|| = ||fdr(T[i, i+k)||$. In this case, we have $fdr(T[i, i+k+1]).span \le fdr(T[i, i+k]).span$ since the span is non-increasing after an intersection operation. In view of the above two cases, we conclude that the *increase* of the number of disjoint angular ranges by 1 (in Case 1 only) is due to the *decrease* of the span by $(2\pi - 2\epsilon_t)$. Since at the beginning, $||fdr(T[i, i+1])|| = 1$ and $fdr(T[i, i+1]).span = 2\epsilon_t$, $||fdr(T[i, i+k])||$ has its greatest value equal to $(1 + \lfloor \frac{2\epsilon_t}{2\pi - 2\epsilon_t} \rfloor)$. □

According to Lemma 5, $||fdr(T[i,j])||$ is usually bounded by a small constant. Let $C = \min\{1 + \lfloor \frac{\epsilon_t}{(\pi - \epsilon_t)} \rfloor, j - i\}$. In particular, when $\epsilon_t \le \pi/2$, $\lfloor \frac{\epsilon_t}{(\pi - \epsilon_t)} \rfloor$ is equal to 0. In this case, $||fdr(T[i,j])||$ is exactly equal to $\min\{1, j - i\} = 1$ (since $j > i$).

4.2.2 *Issue 2: How to Compute* $fdr(T[i,j])$ *Efficiently*

A straightforward method to compute $fdr(T[i,j])$ is to compute $fdr(\overline{p_h p_{h+1}})$ $(i \leq h < j)$ *independently* and then to intersect these fdr's. This straightforward method, nevertheless, incurs the cost of $\Omega(n)$ on average. Since we have $\Theta(n^2)$ instances of $fdr(T[i,j])$, computing all instances of $fdr(T[i,j])$ with this method incurs the total cost of $\Omega(n^3)$. Fortunately, this method could be improved significantly since it involves a lot of redundant work. A better method only takes $O(C)$ instead of $\Omega(n)$ to compute $fdr(T[i,j])$ based on the following *incremental property*.

Given two integers i and j where $1 \leq i < j < n$,

$$fdr(T[i,j+1]) = fdr(T[i,j]) \cap fdr(\overline{p_j p_{j+1}}) \qquad (6)$$

Suppose that the content of $fdr(T[i,j])$ is known, since the total number of angular ranges in $fdr(\overline{p_j p_{j+1}})$ is 1, we can compute $fdr(T[i,j+1])$ in $O(C)$ time. Note that C is the greatest number of angular ranges in $fdr(T[i,j])$ and the intersection operation between two intervals could be finished in $O(1)$ time.

Thus, we propose to compute $fdr(T[i,j])$ $(1 \leq i < j \leq n)$ using the incremental property. Specifically, it involves n rounds.

- At round 1, it computes $fdr(T[h,h+1])$ (i.e., $fdr(\overline{p_h p_{h+1}})$) for $1 \leq h < n$.
- At round r $(r > 1)$, it computes $fdr(T[h,h+r])$ for $1 \leq h \leq n-r$. Specifically, it computes $fdr(T[h,h+r])$ by intersecting $fdr(T[h,h+r-1])$ (which has been maintained at round $r-1$) and $fdr(\overline{p_{h+r-1} p_{h+r}})$. Note that this operation takes $O(C)$ time.

Complexity Analysis. The time complexity of the above method for computing $fdr(T[i,j])$'s for $1 \leq i < j \leq n$ is $O(C \cdot n^2)$ since it involves n rounds and each round incurs the cost of $O(C \cdot n)$. Since there are $O(n^2)$ times of checking whether $\epsilon(\overline{p_i p_{i+1}}) \leq \epsilon_t$ and each checking can be done in $O(C)$ time with the $fdr(T[i,j])$ information, the time complexity of the *SP* algorithm with the complexity improvement is $O(C \cdot n^2)$. Besides, the above method of computing $fdr(T[i,j])$'s has its space complexity of $O(C \cdot n)$ since at each round, it is sufficient to maintain $O(n)$ $fdf(T[i,j])$'s each of which involves $O(C)$ intervals. Since the *SP* algorithm with the complexity improvement materializes G_{ϵ_t} explicitly, we know that its space complexity is $O(C \cdot n + |V| + |E|)$, where V (E) is the vertex (edge) set of G_{ϵ_t}.

4.3 Combining The Two Enhancements

The practical enhancement involves the construction of only *some* edges, which means that we just need to perform the checking of $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ for *some* pairs of positions (p_i, p_j) only where $1 \leq i < j \leq n$. However, the cost of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ is $O(n)$ which is expensive. In contrast, the complexity improvement reduces the cost of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ from $O(n)$ to $O(C)$, but the undesired part is that it always computes $fdr(T[i,j])$ for *all* pairs of positions (p_i, p_j).

In this part, we propose to unify the good aspects of both the practical enhancement and the complexity improvement and at the same time, to avoid their undesired aspects. Our strategy is to maintain $fdr(T[i,j])$ $(1 \leq i < j \leq n)$ (i.e., the idea of complexity improvement) on the demand of the BFS process (i.e., the idea of the practical enhancement). Specifically, when checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ $(1 \leq i < j \leq n)$, we have two cases. Case 1: $fdr(T[i,j])$ has been computed. In this case, the checking could be finished in $O(C)$ time. Case 2: $fdr(T[i,j])$ has not been computed. In this case, we recursively resort to $fdr(T[i,j-1])$ for computing $fdr(T[i,j])$. This recursive process stops either in Case 1 or $fdr(T[i,i+1])$ is acquired. Note that $fdr(T[i,i+1])$ could be computed in $O(1)$ time by Equation (3).

This version of *SP* algorithm enjoys the benefit of the practical enhancement since it checks whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ $(1 \leq i < j \leq n)$ on demand of the BFS process and does not materialize G_{ϵ_t} explicitly, and it also enjoys the benefit of the complexity improvement since it adopts the "feasible direction range" concept for checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$, which is fast.

It could be verified that the worse-case time complexity of the *SP* algorithm with both the practical enhancement and the complexity improvement is $O(C \cdot n^2)$ since it computes only a sub-set of all possible $fdr(T[i,j])$'s.

Besides, the space complexity of this version of *SP* is $O(C \cdot n)$ (it is sufficient to maintain for each i the computed $fdr[i,j]$ with the *largest* j among all *computed* $fdr[i,j]$'s throughout the execution of the algorithm since each $fdr[i,j]$ is enquired at most once and note that it does not materialize G_{ϵ_t}).

5. FINDING APPROXIMATE SOLUTION

According to the discussion in Section 4, the time complexity of an optimal algorithm for DPTS is at least quadratic. This, however, is not scalable enough when the datasets involves millions of positions. In this section, we develop an approximate algorithm called *Intersect* for the DPTS problem, which runs in linear time and gives a certain degree of quality guarantee.

Before we describe *Intersect*, we first give a concept of "feasibility" used in the algorithm. Given an error tolerance ϵ_t and two integers i and j where $1 \leq i < j \leq n$, $\overline{p_i p_j}$ is said to be ϵ_t-*feasible* iff $fdr(T[i,j]|\epsilon_t)$ is non-empty. With this concept, we have the following property.

LEMMA 7 (FEASIBILITY). *Given an error tolerance* ϵ_t *and two integers* i *and* j *where* $1 \leq i < j \leq n$, *if* $\overline{p_i p_j}$ *is* $\frac{\epsilon_t}{2}$*-feasible, then* $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$.

PROOF SKETCH. Since $\overline{p_i p_j}$ is $\frac{\epsilon_t}{2}$-feasible, $fdr(T[i,j]|\epsilon_t/2)$ is non-empty. We deduce that for any two segments between p_i and p_j in T, $\overline{p_h p_{h+1}}$ and $\overline{p_{h'} p_{h'+1}}$ where $i \leq h < h' < j$, $\triangle(\theta(\overline{p_h p_{h+1}}), \theta(\overline{p_{h'} p_{h'+1}})) \leq \epsilon_t$. We further deduce that $\theta(\overline{p_i p_j})$ is in $fdr(T[i,j]|\epsilon_t)$ because $\theta(\overline{p_i p_j})$ is equal to the angle of a *vector* which is equal to the sum of all *vectors* between p_i and p_j in T (i.e., $\sum_{k=i}^{j-1} \overrightarrow{p_k p_{k+1}}$ where $\overrightarrow{p_k p_{k+1}}$ is a vector from p_k to p_{k+1} for each $k \in [1, j-1]$) in the two-dimensional space (i.e., the x-coordinate and the y-coordinate). Here, the angle of a vector is defined to be the angle of an anticlockwise rotation from the positive x-axis to this vector. By Lemma 4, $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$. The complete proof could be found in [16]. □

Specifically, *Intersect* has the following steps. Let T' be a variable storing the simplified trajectory to be returned. Let e be a variable storing the position *index* of the last position in T'. Let h be a variable storing the position index of the position in T being processed. Initially, *Intersect* initializes T' to be (p_1), and then sets e to be 1 (since p_1 is currently the last position in T'). Then, it reads each of the remaining positions sequentially. It sets h to 2 (since p_2 is the position in T to process next). It proceeds with an iterative step as follows. Whenever $h \leq n$ and $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible, it increments h by 1. It terminates this iterative step when either (1) $h > n$ or (2) h has *just* been incremented to a value such that $\overline{p_e p_h}$ is not $\frac{\epsilon_t}{2}$-feasible. For both stopping conditions, we know that $\overline{p_e p_{h-1}}$ is $\frac{\epsilon_t}{2}$-feasible and thus by Lemma 7, we have $\epsilon(\overline{p_e p_{h-1}}) \leq \epsilon_t$. Thus, p_{h-1} is appended to T'. Then, e is set to $h-1$. It repeats the above iterative step whenever $h \leq n$. At the end, it returns T'. The pseudo-code of *Intersect* is shown in Algorithm 2.

With Lemma 7, it is easy to verify that the trajectory returned by *Intersect* is an ϵ_t-simplification of T.

Algorithm 2 The *Intersect* Algorithm

```
Input: A trajectory $T = (p_1, p_2, ..., p_n)$; an error tolerance $\epsilon_t$
Output: An $\epsilon_t$-simplification of $T'$
1: $T' \leftarrow (p_1)$; $e \leftarrow 1$; $h \leftarrow 2$
2: while $h \le n$ do
3:     while $h \le n$ and $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible do
4:         increment $h$ by 1
5:     append $p_{h-1}$ to $T'$; $e \leftarrow h - 1$
6: return $T'$
```

LEMMA 8. *Let T' be the output of the* Intersect *algorithm in Algorithm 2. Then, T' is an ϵ_t-simplification of T.* □

Intersect not only scans the data once only and returns an ϵ_t-simplification of T at the end, but also provides a certain degree of guarantee on the size of the simplified trajectory.

LEMMA 9 (SIZE BOUND). *Let T' be the output of the* Intersect *algorithm in Algorithm 2. We have $|T'| \le |T''|$, where T'' is the $\epsilon_t/2$-simplification of T with the minimum size.* □

PROOF SKETCH. Let $T = (p_1, ..., p_n)$, $T' = (p_{s_1}, ..., p_{s_m})$ and $T'' = (p_{r_1}, ..., p_{r_l})$. By definition, we have $s_1 = r_1 = 1$ and $s_m = r_l = n$. Note that $n = |T|, m = |T'|$ and $l = |T''|$. Our proof has two steps. First, we prove that $fdr(T[r_k, r_{k+1}]|\epsilon_t/2)$ is non-empty for $1 \le k < l$. This could be verified by the fact that T'' is an $\epsilon_t/2$-simplification. Second, we prove $m \le l$ by contradiction which main idea is that if $m > l$, we show $s_k \ge r_k$ for $k = 1, 2, ..., l$ by induction, which, however, results in that $s_m > n$ contradicting $s_m = n$. The complete proof could be found in [16]. □

Complexity Analysis. We know that variable h is incremented whenever line 4 of Algorithm 2 is executed. Both the stopping condition of the outer while-loop (line 2) and one of the stopping conditions of the inner while-loop (line 3) are "$h \le n$". We conclude that there are $O(n)$ times to execute the steps in line 3, line 4 and line 5. The step in line 4 and the step in line 5 take $O(1)$ time. In the following, we show that the step in line 3 also takes O(1) time. Thus, the time complexity of *Intersect* is $O(n)$.

The remaining issue is to derive the time complexity of the step in line 3. In line 3, checking whether $h \le n$ can be done in $O(1)$ time. However, a straightforward implementation of checking whether $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible (line 3) (or equivalently checking whether $fdr(T[e, h]|\epsilon_t/2)$ is non-empty) is expensive. This is because as described in Section 4.2.2, computing $fdr(T[e, h]|\epsilon_t/2)$ from scratch is expensive. By using the same technique in Section 4.2.2, we can perform this checking operation in $O(1)$ time. Specifically, we introduce a variable called fdr to store the content of $fdr(T[e, h]|\epsilon_t/2)$. We have the following two changes in the algorithm due to this variable. Firstly, between line 2 and line 3, we insert a statement that "fdr $\leftarrow fdr(\overline{p_e p_h}|\epsilon_t/2)$". Note that $h = e + 1$ at this moment, and the time complexity of this statement is $O(C')$ where $C' = \min\{1 + \lfloor \frac{\epsilon_t/2}{(\pi - \epsilon_t/2)} \rfloor, h - e\}$ (by using Lemma 5). Secondly, in the inner while-loop, just after line 4, we insert a statement that "fdr $\leftarrow$ fdr $\cap fdr(\overline{p_{h-1} p_h}|\epsilon_t/2)$". Note that the time complexity of this statement is $O(C')$. With these two changes, checking whether $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible (line 3) is equivalent to checking whether the current content of fdr is non-empty because the current content of fdr is equal to $fdr(T[e, h]|\epsilon_t/2)$ (by Equation (6)). Furthermore, since ϵ_t is at most π and thus $\epsilon_t/2$ is at most $\pi/2$, we deduce that $C' = O(1)$. We conclude that the time complexity of the step in line 3 is $O(1)$.

	# of trajectories	total # of positons	average # of positions per trajectory	directional difference between two adjacent segments (Mean, S.D.)
Deer	32	20,065	627	(1.669, 0.948)
Elk	33	47,204	1,430	(1.647, 0.984)
Hurricane	570	17,736	31	(0.213, 0.300)
Geolife	17,621	24,876,978	1,412	(0.364, 0.615)
T-Drive	10,359	17,740,902	1,713	(0.657, 0.803)

Table 1: Real datasets

The space complexity of *Intersect* is $O(n)$ (which corresponds to the memory usage for storing the simplified trajectory T').

6. EMPIRICAL STUDIES

6.1 Datasets and Algorithms

We used 5 real datasets in our experiments, namely Deer, Elk, Hurricane, Geolife and T-Drive. Deer and Elk[1] are two animal movement datasets which contain the radio-telemetry locations of deers in 1995 and elks in 1993, respectively. Hurricane[2] contains the trajectories of the Atlantic hurricanes from year 1950 to year 2004. These three datases (i.e., Deer, Elk and Hurricane) are benchmark datasets for trajectory clustering [15]. 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 [24, 23]. The statistics of these datasets are summarized in Table 1.

We study 5 optimal algorithms with the following notions. DP is the dynamic programming algorithm and SP is the straightforward implementation of the *SP* algorithm. SP-prac (SP-theo) is the *SP* algorithm with the practical enhancement (complexity improvement) only and SP-both is the one with all enhancements. Besides, we study 4 approximate algorithms, Split, Merge, Greedy and Intersect. The first three are the adaptations of the existing trajectory simplification methods and the forth is proposed in this paper.

All algorithms were implemented in C/C++ and run on a Linux platform with a 2.66GHz machine and 4GB RAM.

6.2 Relevance to Existing Studies

In this section, we conducted experiments to show how DPTS is relevant to existing studies.

6.2.1 Bounds of DPTS wrt Existing Measurements

In this part, we verify the theoretical bounds of the length (speed) error and the position error of DPTS as introduced in Section 3.1.2.

We vary the tolerance ϵ_t on $\{0.2, 0.4, 0.6, 0.8, 1\}$. The results about length (speed) errors are shown in Figure 8(a), where the "length (speed) ratio" is defined to be $\min_{i \in [1,n)}\{len(p'_i, p'_{i+1}|T')/len(p_i, p_{i+1}|T)\}$ where p'_i is the estimated position of p_i on T' for $i \in [1, n]$. Thus, the larger this ratio is, the more accurate the length (speed) information of the simplified trajectory is. Note that the length ratio and the speed ratio (with respect to a segment in the original trajectory) are exactly the same since the speed is equal the length divided by the time difference between the time stamps of the two end-points of the segment, and the time difference in the original trajectory is kept to be the same as the time difference in the simplified trajectory. We observe that the theoretical bound of the length (speed) ratio is usually good (e.g., it is about 0.92 when $\epsilon_t = 0.4$).

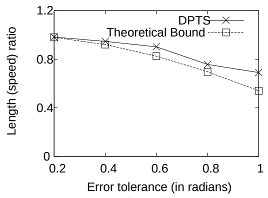
[1]http://www.fs.fed.us/pnw/starkey/data/tables/

[2]http://weather.unisys.com/hurricane/atlantic/

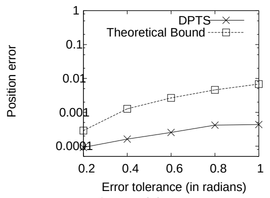
[3]http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

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

The results about the position error are shown in Figure 8(b). We observe that the empirical position error is usually significantly smaller than the theoretical bound (by near to one order of magnitude). Besides, when ϵ_t increases, the increase in the position error of DPTS becomes smaller. When ϵ_t becomes large, the position error of DPTS keeps quite stable.



(a) Length and speed error



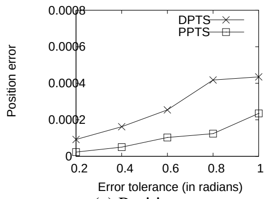
(b) position error

Figure 8: Verification of theoretical error bounds (Geolife)

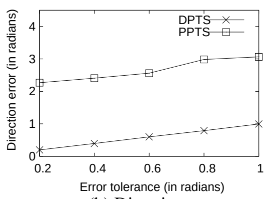
6.2.2 DPTS vs. PPTS

In this part, we want to compare DPTS with Position-Preserving Trajectory Simplification (PPTS) in terms of two measurements, namely the position error and the direction error. In Section 3, we show that the position error of DPTS is *bounded* (Lemma 2) while the direction error of PPTS is *un-bounded* (Lemma 3). We study *how worst the position error of DPTS compared with PPTS is* and *how worst the direction error of PPTS compared with DPTS is*.

We adopt the *Douglas-Peucker* algorithm [6] for PPTS which is the most popular existing algorithm for PPTS [17, 3, 8], and we use our *SP* algorithm for DPTS. We vary ϵ_t for DPTS. For a fair comparison, we enforce that the simplified trajectories from DPTS and PPTS have the same size. The results are shown in Figure 9(a) for position errors and in Figure 9(b) for direction errors. Consider Figure 9(a). It could be noticed that though the position errors of DPTS are usually larger than those of PPTS, the difference is small. For example, the ratio of the position errors is between 1.85 to 3. Consider Figure 9(b). We observe that the direction errors of PPTS are significantly larger than that of DPTS. For example, when $\epsilon_t = 0.2$, the ratio is more than 10. Besides, the direction errors of PPTS are greater than 2, a value greater than $\pi/2$, even with a small value of ϵ_t and is nearly to π, the greatest possible direction error, with a medium value of ϵ_t, which implies that PPTS can hardly preserves the direction information. In conclusion, our DPTS preserves the direction information by its nature and also the position information to a certain degree, but PPTS preserves the position information only but not the direction information.



(a) Position error



(b) Direction error

Figure 9: Comparison with existing PPTS (Geolife)

6.2.3 An Application Study (Trajectory Clustering)

In Section 6.2.2, we compared DPTS and PPTS with their *favor-metrics* (i.e., the position error, favoring PPTS, and the direction error, favoring DPTS). In this section, we compare DPTS with PPTS with a *neutral* metric, the clustering quality, for a real-life application, trajectory clustering.

The main idea is as follows. Let $\mathcal{D}$ be a set of raw trajectories. We perform DPTS (PPTS) on each trajectory in $\mathcal{D}$ and obtain a set of simplified trajectories, denoted by $\mathcal{D}^d$ ($\mathcal{D}^p$). Then, we perform a clustering procedure on each of these two sets of trajectories and obtain the corresponding clustering results. We regard the clustering results based on $\mathcal{D}$ as ground truth and measure the qualities of the clustering results on $\mathcal{D}^d$ and $\mathcal{D}^p$. We verify DPTS by showing that the clustering results on $\mathcal{D}^d$ are consistently better (closer to the ground truth) than those on $\mathcal{D}^p$.

Consider the clustering procedure on $\mathcal{D}$ first. The clustering results based on $\mathcal{D}$ (i.e., the cluster membership of each trajectory) could be represented by a binary matrix $M_{n_t \times n_c}$, where n_t is the number of trajectories in the set and n_c is the number of resulting clusters. Note that under some clustering mechanisms such as the one in [15], a trajectory could belong to multiple clusters. For each trajectory $T \in \mathcal{D}$, its cluster membership could be represented by an n_c-dimensional binary vector. For each pair of trajectories T_1 and T_2 in $\mathcal{D}$, we measure the similarity between T_1 and T_2 by the Euclidean distance between T_1's cluster membership (which is a vector) and T_2's cluster membership (which is a vector). If the distance is below a pre-set threshold σ, we regard T_1 and T_2 to be *similar*; otherwise, we regard T_1 and T_2 to be *dissimilar*. Thus, based on the clustering results on $\mathcal{D}$, we can always obtain a *similarity matrix* which indicates for each pair of two trajectories in $\mathcal{D}$ whether they are similar or not. Let S be such a similarity matrix corresponding to $\mathcal{D}$. The distance threshold σ for deciding whether two trajectories are similar is set to 0.5 by default (all distances are normalized to [0, 1]).

For a specific trajectory $T \in \mathcal{D}$, we denote its simplification in $\mathcal{D}^d$ and $\mathcal{D}^p$ by T^d and T^p, respectively.

Similarly, we perform the same clustering procedure on $\mathcal{D}^d$ and $\mathcal{D}^p$ as we did on $\mathcal{D}$ and obtain their corresponding similarity matrices, denoted by S^d and S^p, respectively.

We measure the quality of the clustering on $\mathcal{D}^d$ (and the clustering on $\mathcal{D}^p$) as follows. For a pair of two trajectories T_1 and T_2 in $\mathcal{D}$, we have 4 cases. Case 1: T_1 and T_2 are similar (wrt S) and T_1^d and T_2^d (the simplified trajectories of T_1 and T_2 under DPTS) are similar (wrt S^d). In this case, we have an occurrence of true positive (TP). Case 2: T_1 and T_2 are dissimilar (wrt S) and T_1^d and T_2^d are similar (wrt S^d). In this case, we have an occurrence of false positive (FP). Similarly, Case 3 and Case 4 correspond to the occurrences of false negative (FN) and true negative (TN), respectively. We adopt three measures in our experiments for measuring the clustering results. The first is called *Rand*, which is defined to be $(|TP| + |TN|)/(|TP| + |FP| + |FN| + |TN|)$ where $|\cdot|$ denotes the number of occurrences. The second and the third are defined to be $|TP|/(|TP| + |FP|)$ and $|TN|/(|TN| + |FN|)$, respectively. The larger the measure is, the better the clustering is.

For the trajectory clustering procedure, we adopt the *TRACLUS* algorithm [15] and the *CATS* algorithm [9]. According to [9], existing trajectory clustering algorithms fall in two categories. The first category includes those algorithms which take each trajectory *as a whole* for clustering while the second category includes the algorithms which use *sub-trajectories* for clustering. *CATS* is the state-of-the-art in the first category and *TRACLUS* is the state-of-the-art in the second category [9].

For the PPTS procedure, again, we adopt the popular *Douglas-Peucker* algorithm.

We vary the error tolerance ϵ_t with the values of 0.2, 0.4, 0.6, 0.8 and 1 for DPTS. Figure 10(a), (b) and (c) show the results about the Rand measure, the measure of $|TP|/(|TP| + |FP|)$ and the measure of $|TN|/(|TN|+|FN|)$ on the Deer dataset, respectively.

We observe that the clustering based on the simplified trajectories returned by DPTS is consistently better than that based on the simplified trajectories returned by PPTS. This might be explained by the fact that the direction information is heavily used in the trajectory clustering algorithms and the direction information loss due

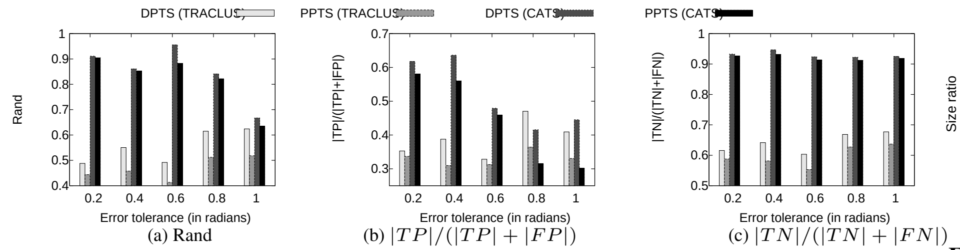

(a) Rand (b) $|TP|/(|TP|+|FP|)$ (c) $|TN|/(|TN|+|FN|)$

Figure 10: Trajectory Clustering Study (the Deer dataset)

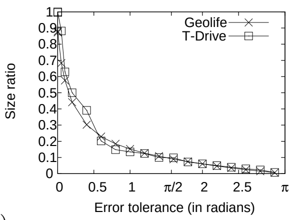


Figure 11: Compression rate

to DPTS is bounded while that due to PPTS is un-bounded.

6.3 Performance Study of the Optimal Algs

In the part, we study the effects of 2 factors, namely the data size (i.e., $|T|$) and the error tolerance (i.e., ϵ_t), on the performance of the optimal algorithms. We use 2 measures, namely the running time and the memory.

Effect of $|T|$. The values used for $|T|$ are around $2k$ $4k$, $6k$, $8k$ and $10k$ (ϵ_t is fixed to be 1). For each setting of $|T|$, we select a set of 10 trajectories each of which has its size near to this value and run DPTS on each of these trajectories. Then, we average the experimental results on these trajectories (this policy is used throughout our experiments without specification). Figure 12 show the results on Geolife. According to these results, SP-both is the fastest while DP is the slowest due to its high time complexity. Besides, the complexity improvement helps to reduce the running time dramatically (e.g., SP-theo is faster than SP by 2-3 orders of magnitude). This could be easily explained by the fact that with the complexity improvement, the cost of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ is reduced from $O(n)$ to $O(C)$ (C is a small constant). Though the practical enhancement improves the time efficiency a little, it helps to reduce the memory significantly (e.g., the memory occupied by SP-theo is 1-3 orders of magnitude larger than that occupied by SP-both and the difference increases when $|T|$ increases on Geolife).

The experimental results on T-Drive are similar. Due to page limit, we put these results in the full version of this paper [16].

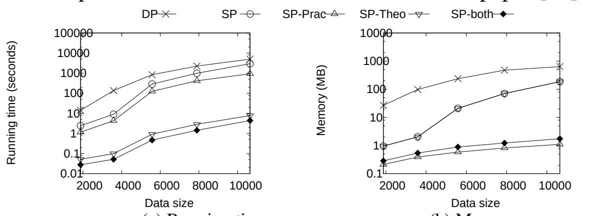

(a) Running time (b) Memory

Figure 12: Effect of data size $|T|$ (Geolife)

Effect of ϵ_t. The values used for ϵ_t are 0.2, 0.4, 0.6, 0.8 and 1 in radians ($|T|$ is fixed to be $5k$). Figure 13 shows the results on Geolife. According to these results, ϵ_t affects the SP algorithms only. Specifically, the running times of all SP algorithms increase slightly when ϵ_t becomes larger. This is because a larger ϵ_t usually results in G_{ϵ_t} with more edges and thus the BFS process on G_{ϵ_t} needs more time.

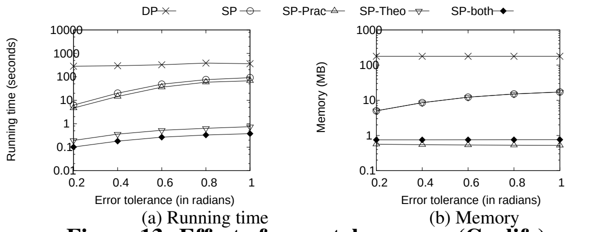

(a) Running time (b) Memory

Figure 13: Effect of error tolerance ϵ_t (Geolife)

Compression Rate. We also study the effect of ϵ_t on the *size ratio* which is defined to be equal to $\sum_{T' \in \mathcal{D}'} |T'| / \sum_{T \in \mathcal{D}} |T|$, where $\mathcal{D}$ is the set of raw trajectories and $\mathcal{D}'$ is the set of the corresponding simplified trajectories. Note that a smaller size ratio means a higher compression rate. The results are shown in Figure 11. We have the following observations. First, the size ratio decreases significantly when we increase the tolerance from 0 slightly. This is good since it implies that under DPTS, the trajectory data could be simplified significantly with a small error. Second, we observe that the size ratio is strictly smaller than 1 (e.g., it is about 0.9 for the Geolife datasets) even if the error tolerance is set to be 0. This implies that the real-life trajectories usually involve a certain degree of redundancy and could be simplified without incurring any error.

Scalability Test. Figure 14 shows the results of the scalability test on the optimal algorithms. We only show the results of SP-theo and SP-both since the other optimal algorithms are not scalable on large datasets due to their expensive time complexities. According to these results, both SP-theo and SP-both are scalable to large trajectory datasets with millions of positions, and SP-both runs slightly faster than SP-theo. It is noted that SP-both occupies significantly less memory than SP-theo and thus SP-both is more scalable than SP-theo. This is because SP-both does not materialize G_{ϵ_t} explicitly while SP-theo does.

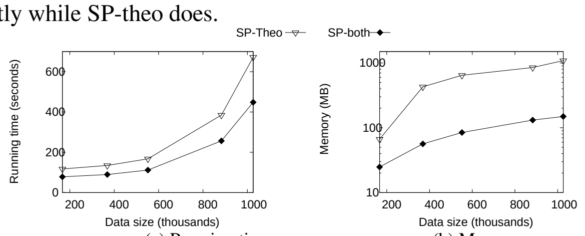

(a) Running time (b) Memory

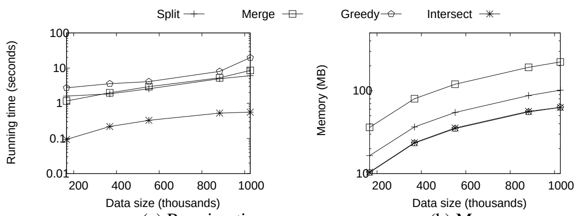
Figure 14: Scalability test (Geolife)

6.4 Performance Study of the Approx. Algs

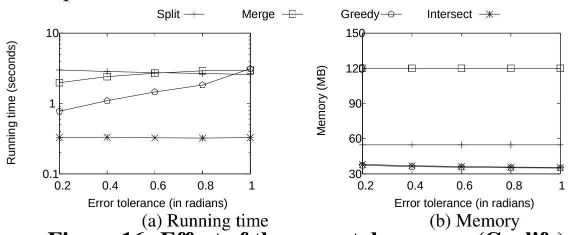
In this part, we study the effects of $|T|$ and ϵ_t on the approximate algorithms. We use 3 measures, namely the running time, the memory and the approximation error. The approximation error of an approximate algorithm is defined to be $|T'|/|T^*|$, where T' is the simplified trajectory returned by the approximate algorithm on a given raw trajectory and T^* is the simplified trajectory returned by an optimal algorithm on the same raw trajectory.

Effect of $|T|$. The values used for $|T|$ are around $200k$, $400k$, $600k$, $800k$ and $1000k$ (ϵ_t is fixed to be 1). Figure 15 shows the results. According to these results, Intersect is the fastest, which is at least 1 order of magnitude faster than other approximate algorithms. This is because Intersect runs in linear time while the other algorithms run in quadratic time in the worst case [16]. Besides, Intersect occupies the least memory and Greedy occupies slightly more memory than Intersect (though the difference in Figure 15(b) is not obvious).

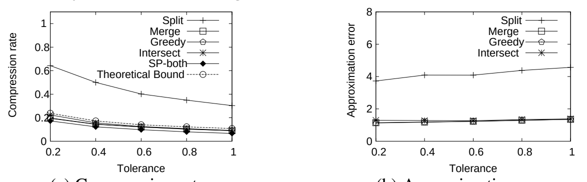
Effects of ϵ_t. Figure 16 shows the effects of ϵ_t on the approximate algorithms, where we vary ϵ_t with 0.2, 0.4, 0.6, 0.8 and 1 ($|T|$ is fixed to be $500k$). According to these results, Greedy runs faster than Split and Merge with small ϵ_t's. This is because with a smaller ϵ_t, it is less likely that a long sequence of consecutive segments could be approximated with one segment and thus the cost


(a) Running time (b) Memory
Figure 15: Effect of data size $|T|$ (Geolife)

of checking the error of the segment linking the start position and the end position is small.


(a) Running time (b) Memory
Figure 16: Effect of the error tolerance ϵ_t (Geolife)

Compression Rate & Approximation Error. Figure 17(a) shows the results on the compression rates of the approximate algorithms. We also show the theoretical bound of the size the trajectory returned by Intersect (Lemma 9). Figure 17(b) shows the results on the approximate errors of the approximate algorithms. Both these results verify our Intersect algorithm.

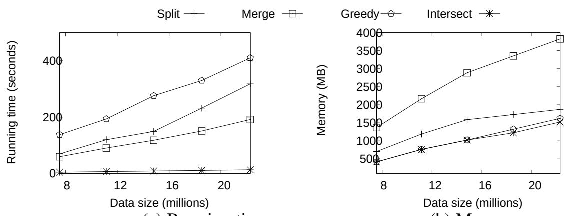

(a) Compression rate (b) Approximation error
Figure 17: Compression rates and approximate errors of the approximate algoritms (Geolife)

Scalability Test. The largest trajectory in our real datasets contains around 2M positions only. In order to generate larger trajectories, we concatenate multiple trajectories in ascending order of their time stamps into one. Figure 18 shows the results of the scalability test on the approximate algorithms. According to these results, Intersect is very fast on large datasets with more than 20M positions. For example, Intersect runs in 13.3s on the trajectory with 24,876,978 positions. In contrast, the running times of other approximate algorithms increase much faster when $|T|$ increases.

7. CONCLUSION

In this paper, we propose direction-preserving trajectory simplification, which has not been studied in the literature, as a novel alternative to the traditional position-preserving trajectory simplification. We propose an optimal algorithm called *SP* and an approximate algorithm called *Intersect*. We conducted experiments to show the efficiency and the scalability of our proposed methods. There are many possible future directions. Firstly, it is interesting to study how to extend our algorithms when each segment is associated with a weight. Secondly, it is good to study how to simplify trajectories when positions are associated with labels (e.g., restaurants and gasoline stations).

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 grants DAG12EG07-7 and FSGRF13EG27, and the research of H. V. Jagadish is supported in part by NSF grant IIS-1250880.


(a) Running time (b) Memory
Figure 18: Scalability test (Geolife)

8. REFERENCES

[1] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB'05*, pages 853–864.

[2] S. Brakatsoulas, D. Pfoser, and N. Tryfona. Modeling, storing and mining moving object databases. In *IDEAS'04*, pages 68–77.

[3] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.

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

[5] Y. Chen, K. Jiang, Y. Zheng, C. Li, and N. Yu. Trajectory simplification method for location-based social networking services. In *Proceedings of the 2009 International Workshop on Location Based Social Networks*, pages 33–40. ACM, 2009.

[6] 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.

[7] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *SIGMOD'07*, pages 330–339.

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

[9] C.-C. Hung, W.-C. Peng, and W.-C. Lee. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. *The VLDB Journal*, pages 1–24, 2011.

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

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

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

[13] J. G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE'08*, pages 140–149.

[14] J. G. Lee, J. Han, X. Li, and H. Gonzalez. Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. *PVLDB'08*, 1(1):1081–1094, 2008.

[15] J. G. Lee, J. Han, and K. Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD'07*, pages 593–604.

[16] C. Long, R. C.-W. Wong, and H. Jagadish. Direction-preserving trajectory simplification (technical report). In *http://www.cse.ust.hk/~raywong/paper/trajectory.pdf*, 2013.

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

[18] 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.

[19] D. Patel, C. Sheng, W. Hsu, and M. L. Lee. Incorporating duration information for trajectory classification. In *ICDE'12*, pages 1132–1143.

[20] N. Pelekis, I. Kopanakis, G. Marketos, I. Ntoutsi, G. Andrienko, and Y. Theodoridis. Similarity search in trajectory databases. In *TIME'07*, pages 129–140.

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

[22] M. Singh, Q. Zhu, and H. Jagadish. Swst: A disk based index for sliding window spatio-temporal data. In *ICDE'12*, pages 342–353.

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

[24] 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.