Velocity Vector Preserving Trajectory Simplification

Guanzhi Wang*[1], Zhenmei Shi*[1], Cheng Long[2], Ya Gao[3], Raymond Chi-Wing Wong[4]
[1,3,4]Hong Kong University of Science and Technology, [2]Nanyang Technological University
{gwangaj, zshiad}@connect.ust.hk[1], c.long@ntu.edu.sg[2]
gaoyarosy@gmail.com[3], raywong@cse.ust.hk[4]

ABSTRACT

A trajectory contains both positional and temporal information. Despite many trajectory simplification algorithms being proposed, not many works focus on both positional and temporal information. In this paper, we propose velocity-vector-based error measurement which is closely related to speed and direction information, and also introduce the notion of Velocity Vector Preserving Trajectory Simplification (VVPTS). We present a linear-space optimal algorithm with $O(n^2 \log n)$ time complexity, and another approximate linear-time linear-space algorithm with a theoretically bounded compression rate. We present extensive analytic and empirical studies in our measurement and two algorithms. Notably, our VVPTS algorithms have a quality guarantee under existing error metrics and have a good scalability performance.

1. INTRODUCTION

GPS-enabled devices generate large amounts of spatio-temporal data of moving objects, which can be used in various applications. A trajectory is composed of a series of points containing both positional and temporal information. To relieve storage burden and fasten data processing, it is often necessary to simplify such trajectory with a subset of its points while ensuring that the error or the size of the result has a specified upper bound. There are two types of trajectory simplification problems. *min-# problem* aims to find a simplification whose simplification error does not exceed a given error tolerance and has the minimum size. *min-ϵ problem* aims to find a simplification whose size does not exceed a given threshold and has the minimum simplification error. In this paper, we consider a common situation where the storage capacity is very limited and as long as the error of the simplified trajectory is under a predefined threshold, we would focus on reducing its size. Therefore, min-# problem is our major concern. A trajectory is different from a polygonal line since it incorporates the time dimension. From positional and temporal information, the velocity of the movement can be derived. Velocity information is of vital importance to a wide range of applications, including traffic analysis.

This paper studies trajectory simplification which preserves velocity vector information. The reminder of this paper is organized as follows. Section 2 defines the min-# velocity vector preserving trajectory simplification (min-# VVPTS) problem. Section 3 introduces some of the popular trajectory simplification error measurements and algorithms and discusses how they are related to the newly defined velocity-vector-based error measurement and algorithms of trajectory simplification under this new metric. Section 4 and Section 5 develop algorithms to find the optimal and approximate solutions to the min-# VVPTS problem, respectively. Section 6 gives empirical studies for the error metrics and algorithms concerned in this paper. Section 7 summarizes the work.

*Equal contribution

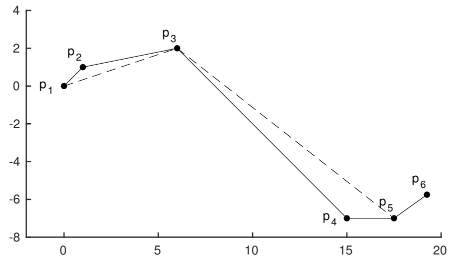


Figure 1: An example trajectory

2. PROBLEM DEFINITION

A *trajectory* of a moving object consists of a sequence of points with their positions in a two dimensional space and time stamps. A point p_i in such a trajectory is represented by (x_i, y_i, t_i) where t_i is the time when the point is recorded and (x_i, y_i) is the coordinate of the moving object in the 2D Euclidean space at time t_i. $T = (p_1, p_2, \dots, p_n)$ represents a trajectory T with n ordered points $(t_1 < t_2 < \dots < t_n)$. A *simplification* of trajectory T can be represented by $T_s = (p_{s_1}, p_{s_2}, \dots, p_{s_m})$ where $1 = s_1 < s_2 \dots < s_m = n$. The number of vertices, or the *size* of T_s, denoted by $|T_s|$, does not exceed the size of the original trajectory, i.e., $|T_s| = m \leq |T| = n$. A segment is defined by two consecutive points on a trajectory. $p_i p_{i+1}$ $(i \in [1, n))$ is the i^{th} segment of T and $p_{s_k} p_{s_{k+1}}$ $(k \in [1, m))$ is the k^{th} segment of T_s. The *sub-trajectory* of T from p_i to p_j $(1 \leq i < j \leq n)$, denoted by $T[i, j]$, refers to the portion $(p_i, p_{i+1}, \dots, p_j)$ of T. The segment $p_{s_k} p_{s_{k+1}}$ of T_s is a simplification of the sub-trajectory of T from p_{s_k} to $p_{s_{k+1}}$. Figure 1 shows a trajectory $T = (p_1, p_2, p_3, p_4, p_5, p_6)$. Its simplification should preserve the starting point p_1 and ending point p_6 and remove some of the points in between. $T_s = (p_1, p_3, p_5, p_6)$ is a simplification of T. $|T_s| = 4$, $s_1 = 1$, $s_2 = 3$, $s_3 = 5$ and $s_4 = 6$. $p_1 p_2$ is a segment of T. $p_1 p_3$ is not a segment of T but a segment of T_s since the two end points are not consecutive on T but consecutive on T_s. $p_1 p_3$ on T_s is a simplification of the sub-trajectory $T[1, 3]$, namely (p_1, p_2, p_3) on T.

Velocity of a moving object is the rate of change of the object's position, which reveals both speed and direction of the movement. The velocity of a segment $p_i p_{i+1}$, denoted by $\vec{V}(p_i p_{i+1})$, equals the displacement in 2D Euclidean space divided by the time interval from p_i to p_{i+1}, i.e., $\vec{V}(p_i p_{i+1}) = \overrightarrow{p_i p_{i+1}} / \Delta T(p_i p_{i+1})$

Notation	Description
$p_i = (x_i, y_i, t_i)$	a point on (x_i, y_i) at time stamp t_i
$T = (p_1, p_2, \dots, p_n)$	a trajectory with n ordered points
$T_s = (p_{s_1}, p_{s_2}, \dots, p_{s_m})$	a simplification of trajectory T
$T[i, j]$	the sub-trajectory of T from p_i to p_j
$\|T\|$	the size of trajectory T
$p_i p_{i+1}$	the i^{th} segment of T
$\overrightarrow{p_i p_j}$	the displacement from p_i to p_j in 2D Euclidean space
$\theta(p_i p_j)$	the direction of $\overrightarrow{p_i p_j}$
$\theta(\overrightarrow{AB})$	the direction of $\overrightarrow{AB}$
$\vec{V}(p_i p_j)$	the average velocity of $T[i, j]$
$\Delta T(p_i p_j)$	the time interval from p_i to p_j
V_{i-j}	the point representing $\vec{V}(p_i p_j)$ in 2D Euclidean space
$V_x(p_i p_j)$, $V_y(p_i p_j)$	the x-coordinate and y-coordinate of V_{i-j}
E_v, E_{cd}, E_{sed}, E_d	velocity-vector-based, closest Euclidean distance, synchronous Euclidean distance and direction-based trajectory simplification error measurements
$opt(T, \epsilon_t)$	the optimal solution of the min-# VVPTS problem whose input trajectory is T and error tolerance is ϵ_t
$dist(A, B)$	distance between point A and point B in 2D Euclidean space
$\epsilon_v(p_i p_j)$	the simplification error of potential segment $p_i p_j$ under E_v
$\epsilon_{cd}(p_i p_j)$	the simplification error of potential segment $p_i p_j$ under E_{cd}
$\epsilon_{sed}(p_i p_j)$	the simplification error of potential segment $p_i p_j$ under E_{sed}
$\epsilon_d(p_i p_j)$	the simplification error of potential segment $p_i p_j$ under E_d
$\epsilon_v(T_s)$	the simplification error of T_s under E_v
$I_{i-(j-1)}$	the feasible velocity area of potential segment $p_i p_j$
D_i	the feasible velocity area of segment $p_i p_{i+1}$
$I_{i\&j}$	the intersection of D_i and D_j
$Arc_k(I_{i-j})$	the arc which is the part of the boundary of I_{i-j} on the circumference of D_k
$Arc_k(I_{i\&j})$	the arc which is the part of the boundary of $I_{i\&j}$ on the circumference of D_k
$Arc_k(I_{i-j}).P_1$, $Arc_k(I_{i-j}).P_2$	the two end points of $Arc_k(I_{i-j})$
$Arc_k(I_{i-j}).\theta_1$, $Arc_k(I_{i-j}).\theta_2$	the directions of vectors from $V_{k-(k+1)}$ to the two end points of $Arc_k(I_{i-j})$
$Arc_k(I_{i-j}).K_1$, $Arc_k(I_{i-j}).K_2$	the indexes of the disks on which the end points of $Arc_k(I_{i-j})$ falls, $K_1, K_2 \in [i, k) \cup (k, j]$
R_i	reference point for the preprocessing of checking whether V_{i-j} belongs to $I_{i-(j-1)}$ for $j \in [i, n)$
$Arc_k(I_{i-j}).\alpha_1$, $Arc_k(I_{i-j}).\alpha_2$	the directions of vectors from R_i to the two end points of $Arc_k(I_{i-j})$
$\min V_x(T[i, j])$, $\max V_y(T[i, j])$	the minimum and maximum x component of velocities of each segment on $T[i, j]$
$\min V_y(T[i, j])$, $\max V_y(T[i, j])$	the minimum and maximum y component of velocities of each segment on $T[i, j]$
$\Delta V_x(T[i, j])$, $\Delta V_y(T[i, j])$	the ranges of the x and y component of velocities of each segment on $T[i, j]$

Table 1: Notations

where $\overrightarrow{p_i p_{i+1}} = (x_{i+1} - x_i, y_{i+1} - y_i)$ and $\Delta T(p_i p_{i+1}) = t_{i+1} - t_i$. Similarly, the average velocity of a sub-trajectory between p_i and p_j, $\vec{V}(p_i p_j) = \overrightarrow{p_i p_j} / \Delta T(p_i p_j) = (x_j - x_i, y_j - y_i)/(t_j - t_i)$. Same as [8], the *direction* of the sub-trajectory from p_i to p_j, denoted by $\theta(p_i p_j)$, is defined to be the angle of anticlockwise rotation from positive x-axis to $\overrightarrow{p_i p_j}$ and has a range of $[0, 2\pi)$. In the following discussion, the direction of a vector refers to the angle of anticlockwise rotation from the positive x-axis to that vector. A velocity can be decomposed into two linearly independent components, namely a component of $\vec{V}(p_i p_j)$ along the x-axis and a component of $\vec{V}(p_i p_j)$ along the y-axis. The components of $\vec{V}(p_i p_j)$ along the x-axis and the y-axis are computed as $\vec{V}(p_i p_j)\cos\theta(p_i p_j)$ and $\vec{V}(p_i p_j)\sin\theta(p_i p_j)$ respectively. Velocity $\vec{V}(p_i p_j)$ can be represented by point V_{i-j} whose coordinate is $(V_x(p_i p_j), V_y(p_i p_j))$ in a 2D Euclidean space, $V_x(p_i p_j) = \|\vec{V}(p_i p_j)\|\cos\theta(p_i p_j)$ and $V_y(p_i p_j) = \|\vec{V}(p_i p_j)\|\sin\theta(p_i p_j)$. Consider the example trajectory T shown in Figure 1, the unit of x and y coordinates is meter and the unit of time is second, the data of each point are as follows. $p_1 = (0, 0, 0)$, $p_2 = (1, 1, 1)$, $p_3 = (6, 2, 2)$, $p_4 = (15, -7, 5)$, $p_5 = (17.5, -7, 6)$, $p_6 = (19.25, -5.75, 6.5)$. By the definitions above, it can be computed that $\vec{V}(p_2 p_3) = (5, 1)$ where $V_x(p_2 p_3) = 5$ and $V_y(p_2 p_3) = 1$. $\vec{V}(p_2 p_3)$ can be represented by point V_{2-3} whose coordinate is $(5, 1)$. The speed and direction of $\vec{V}(p_2 p_3)$, denoted by $\|\vec{V}(p_2 p_3)\|$ and $\theta(p_2 p_3)$, are $\sqrt{26}$ and 11.310 degree, respectively. Similarly, $\vec{V}(p_1 p_3) = (3, 1)$ where $V_x(p_1 p_3) = 3$ and $V_y(p_1 p_3) = 1$. $\vec{V}(p_1 p_3)$ can be represented by point V_{1-3} whose coordinate is $(3, 1)$.

We developed a *velocity-vector-based error measurement E_v*. For a trajectory T and its simplification $T_s = (p_{s_1}, p_{s_2}, \dots, p_{s_m})$, the *simplification error* of a segment $p_{s_k} p_{s_{k+1}}$ on T_s under E_v, denoted by $\epsilon_v(p_{s_k} p_{s_{k+1}})$, is defined to be the greatest distance between $V_{s_k - s_{k+1}}$ and $V_{i-(i+1)}$ for $i \in [s_k, s_{k+1})$. Recall that $V_{s_k - s_{k+1}}$ is the point representing the velocity of $p_{s_k} p_{s_{k+1}}$ and $V_{i-(i+1)}$ is the point representing the velocity of $p_i p_{i+1}$. For $i \in [s_k, s_{k+1})$, $p_i p_{i+1}$ are segments of the sub-trajectory $T[s_k, s_{k+1}]$, which is simplified by segment $p_{s_k} p_{s_{k+1}}$ of T_s.

$$\epsilon_v(p_{s_k} p_{s_{k+1}}) = \max_{s_k \le i < s_{k+1}} dist(V_{s_k - s_{k+1}}, V_{i-(i+1)}), where$$
$$dist(V_{s_k - s_{k+1}}, V_{i-(i+1)}) = \sqrt{\Delta V^2(p_{s_k} p_{s_{k+1}}, p_i p_{i+1})}$$
$$= \sqrt{\Delta V_x^2(p_{s_k} p_{s_{k+1}}, p_i p_{i+1}) + \Delta V_y^2(p_{s_k} p_{s_{k+1}}, p_i p_{i+1})},$$
$$\Delta V_x(p_{s_k} p_{s_{k+1}}, p_i p_{i+1}) = V_x(p_{s_k} p_{s_{k+1}}) - V_x(p_i p_{i+1}) \quad and$$
$$\Delta V_y(p_{s_k} p_{s_{k+1}}, p_i p_{i+1}) = V_y(p_{s_k} p_{s_{k+1}}) - V_y(p_i p_{i+1})$$

In our example, since $\vec{V}(p_1 p_2) = (1, 1)$, $\vec{V}(p_2 p_3) = (5, 1)$ and $\vec{V}(p_1 p_3) = (3, 1)$, the distance between V_{1-3} and V_{1-2} and the distance between V_{1-3} and V_{2-3} are 2. $\epsilon_v(p_1 p_3) = 2$. This velocity-vector-based error measurement allows us to quantify the difference in velocities in a way which is independent of the coordinate system used. That is, the directions along which the velocity is decomposed do not affect the value of simplification error. The simplification error of T_s under E_v, denoted by $\epsilon_v(T_s)$, is defined to be the greatest simplification error of a segment on T_s under E_v. $\epsilon_v(T_s) = \max_{1 \le k < m} \epsilon_v(p_{s_k} p_{s_{k+1}})$. Given a non-negative real number ϵ_t, T_s is called an *ϵ_t-simplification* of T if the simplification error of T_s does not exceed ϵ_t, i.e., $\epsilon(T_s) \le \epsilon_t$. The *min-# velocity vector preserving trajectory simplification problem (VVPTS)* is defined as follows.

PROBLEM 1. *Given a trajectory T and an error tolerance ϵ_t ($\epsilon_t \ge 0$), the **min-# velocity vector preserving trajectory simplification problem** is to find an ϵ_t-simplification of T under E_v with minimum size.* □

3. PREVIOUS WORK

3.1 Existing Error Measurements

Different metrics have been developed to evaluate error of the simplified trajectory. *Closest Euclidean distance* (CD) and *synchronous Euclidean distance* (SED) are two prevalent ones [4, 2, 10]. For a trajectory $T = (p_1, p_2, \dots, p_n)$, $T[i, j]$ $(1 \le i < j \le n)$ denotes the sub-trajectory of T from p_i to p_j. For a simplification of T, $T_s = (p_{s_1}, p_{s_2}, \dots, p_{s_m})$, the simplification error of segment $p_{s_k} p_{s_{k+1}}$ $(k \in [1, m))$ under closest Euclidean distance error measurement E_{cd}, denoted by $\epsilon_{cd}(p_{s_k} p_{s_{k+1}})$, is defined to be the maximum of the smallest Euclidean distance between segment $p_{s_k} p_{s_{k+1}}$ and each point on sub-trajectory $T[s_k, s_{k+1}]$. E_{cd} does not take into account the time dimension of the trajectory. SED error measurement considers temporal as well as spatial information. SED error measurement introduces the concept of *temporally synchronized position.* For any point $p_i = (x_i, y_i, t_i)$ on the sub-trajectory $T[s_k, s_{k+1}]$, its approximated temporally synchronized position on T_s is $p'_i = (x'_i, y'_i, t'_i)$, where $x'_i = x_{s_k} + \frac{t_i - t_{s_k}}{t_{s_{k+1}} - t_{s_k}}(x_{s_{k+1}} - x_{s_k})$, $y'_i = y_{s_k} + \frac{t_i - t_{s_k}}{t_{s_{k+1}} - t_{s_k}}(y_{s_{k+1}} - y_{s_k})$ and $t'_i = t_i$. The simplification error of segment $p_{s_k}p_{s_{k+1}}$ under SED error measurement E_{sed}, denoted by $\epsilon_{sed}(p_{s_k}p_{s_{k+1}})$, is defined to be the maximum of the distances between each point p_i on $T[s_k, s_{k+1}]$ and its temporally synchronized position p'_i. According to the definitions, the computations of $\epsilon_{cd}(p_{s_k}p_{s_{k+1}})$ and $\epsilon_{sed}(p_{s_k}p_{s_{k+1}})$ take $O(s_{k+1} - s_k)$ time. *Local integral square error* (LISE) and *local integral square synchronous Euclidean distance* (LSSD) are variations of closest Euclidean distance and SED with improved computational efficiency [2]. The simplification error of segment $p_{s_k}p_{s_{k+1}}$ under LISE and LSSD, denoted by $\epsilon_{lise}(p_{s_k}p_{s_{k+1}})$ and $\epsilon_{lssd}(p_{s_k}p_{s_{k+1}})$, are defined as follows.

$$\epsilon_{lise}(p_{s_k}p_{s_{k+1}}) = \sum_{i=s_k}^{s_{k+1}} D^2(p_i, p_{s_k}p_{s_{k+1}})$$

$$\epsilon_{lssd}(p_{s_k}p_{s_{k+1}}) = \sum_{i=s_k}^{s_{k+1}} SED^2(p_i, p_{s_k}p_{s_{k+1}})$$

Here, $D(p_i, p_{s_k}p_{s_{k+1}})$ denotes the closest Euclidean distance between point p_i and segment $p_{s_k}p_{s_{k+1}}$, and $SED(p_i, p_{s_k}p_{s_{k+1}})$ denotes the distance between p_i and its temporally synchronized position on $p_{s_k}p_{s_{k+1}}$. It has been proved that after $O(n)$ time pre-calculation of some accumulative terms, LISE and LSSD errors of any segment of T_s can be obtained in constant time.

There are several other error metrics for trajectory or polygonal path simplification. Long *et al.* dpts proposed *direction-based error measurement E_d*. The simplification error of segment $p_{s_k}p_{s_{k+1}}$ under E_d, denoted by $\epsilon_d(p_{s_k}p_{s_{k+1}})$, is defined to be the maximum of the *angular differences* between the direction of $p_{s_k}p_{s_{k+1}}$ and the direction of each segment on $T[s_k, s_{k+1}]$. Here, the angular difference between θ_1 and θ_2 is defined to be the minimum of $|\theta_1 - \theta_2|$ and $2\pi - |\theta_1 - \theta_2|$. The time cost of computing $\epsilon_d(p_{s_k}p_{s_{k+1}})$ is $O(s_{k+1} - s_k)$. Bose *et al.* [1] proposed three area-based error measurements. There are applications (e.g., land boundary mapping) in which area distortion is a primary concern but the time complexities of finding optimal solution of trajectory simplification problem under area-based error measurements are usually high. Gudmnundsson *et al.* [5] introduced *distance preserving polygonal path approximation.* The simplification error of segment $p_{s_k}p_{s_{k+1}}$ under the distance-based error measurement E_{dist}, denoted by $\epsilon_{dist}(p_{s_k}p_{s_{k+1}})$, is the ratio of $\delta(p_{s_k}, p_{s_{k+1}})$ to $dist(p_{s_k}, p_{s_{k+1}})$, where $\delta(p_{s_k}p_{s_{k+1}})$ is the sum of the lengths of all segments on $T[s_k, s_{k+1}]$. The time complexities of optimal min-# and min-ϵ trajectory simplification under E_{dist} are $O(n^2)$ and $O(n^2 \log n)$. Preservation of distance is meaningful for the representations of roads and rivers.

We now discuss the relationship between E_v and some existing error measurements. E_v gives certain guarantees on the simplification errors under E_{cd} and E_{sed}. These guarantees can be expressed in terms of the error tolerance under E_v, the distance and time interval between two consecutive points on the simplified trajectory. The proof of Lemma 1 and Lemma 2 could be found in Section A and B in the appendix.

LEMMA 1. *Let T be a trajectory and T_s be an ϵ_t-simplification of T under E_v. For each segment $p_{s_k}p_{s_{k+1}}$ of T_s, $\epsilon_{cd}(p_{s_k}p_{s_{k+1}}) \le dist(p_{s_k}, p_{s_{k+1}})/2 + \epsilon_t \Delta T(p_{s_k}p_{s_{k+1}})$.* □

LEMMA 2. *Let T be a trajectory and T_s be an ϵ_t-simplification of T under E_v. For each segment $p_{s_k}p_{s_{k+1}}$ of T_s, $\epsilon_{sed}(p_{s_k}p_{s_{k+1}}) < dist(p_{s_k}, p_{s_{k+1}}) + \epsilon_t \Delta T(p_{s_k}p_{s_{k+1}})/2$.* □

Trajectory simplifications under E_{cd} and E_{sed}, however, do not limit velocity error. Consider a trajectory $T = (p_1, p_2, \ldots, p_n)$ where all p_i ($i \in [1, n]$) are on a straight line in 2D Euclidean space. For any pair of i, j ($1 \le i < j \le n$), regardless of the speeds of each segment on $T[i, j]$ and the average speed of p_ip_j, $\epsilon_{cd}(p_ip_j)$ always is zero. That is, though located on the same straight line, the distance between any two V_{i-j} for $1 \le i < j \le n$ can be infinitely large and thus, $\epsilon_v(p_ip_j)$ can be unbounded when $\epsilon_{cd}(p_ip_j)$ is zero. Similarly, consider another trajectory $T = (p_1, p_2, \ldots, p_n)$ where all p_i ($i \in [1, n]$) are on a straight line in 2D Euclidean space and p_1, p_n are two end points of the line. For any pair of i, j ($1 \le i < j \le n$), $\epsilon_{sed}(p_ip_j)$ is not greater than $\|p_1 - p_n\|$. However, the distance between any two V_{i-j} for $1 \le i < j \le n$ can be infinitely large and therefore, $\epsilon_v(p_ip_j)$ can be unbounded.

Whether VVPTS offers error bounds on direction and distance information depends on the velocities of each segment of the input trajectory and the error tolerance set.

LEMMA 3. *Let T be a trajectory and T_s be a ϵ_t-simplification of T under E_v. For each segment $p_{s_k}p_{s_{k+1}}$ of T_s, $\epsilon_d(p_{s_k}p_{s_{k+1}})$ will be unbounded if ϵ_t is greater than the smallest speed of segments on $T[s_k, s_{k+1}]$. $\epsilon_d(p_{s_k}p_{s_{k+1}}) \le \arcsin(\epsilon_t / \min_{s_k \le i < s_{k+1}} \|\vec{V}(p_ip_{i+1})\|)$ if ϵ_t is smaller than the speeds of all segments on $T[s_k, s_{k+1}]$.* □

LEMMA 4. *Let T be a trajectory and T_s be a ϵ_t-simplification of T under E_v. For each segment $p_{s_k}p_{s_{k+1}}$ of T_s, $\epsilon_{dist}(p_{s_k}p_{s_{k+1}})$ will be unbounded if ϵ_t is greater than the speeds of all segments on $T[s_k, s_{k+1}]$. If ϵ_t is smaller than the greatest speed of segments on $T[s_k, s_{k+1}]$, $\epsilon_{dist}(p_{s_k}p_{s_{k+1}}) \le 1/(1 - \epsilon_t / \max_{s_k \le i < s_{k+1}} \|\vec{V}(p_ip_{i+1})\|)$.* □

The proofs of Lemma 3 and 4 are trivial.

3.2 Existing Simplification Algorithms

Ying *et al.* [13] proposed Velocity-Preserving Trajectory Simplification (VPTS) to minimize both geometric and velocity error. Actually, the velocity-based error measurement defined in VPTS is an equivalent deformation of E_{dist}, so the compression of VPTS ignores the temporal information. Lin *et al.* [7] proposed the Adaptive Trajectory Simplification (ATS) algorithm to preserve the position feature and the velocity feature from the given trajectories. The ATS algorithm partitions the trajectories into velocity-preserving segments by grouping the velocity values into several intervals. This simplification indeed is derived from the position preserving simplification approach on each segment. Moreover, ATS and VPTS only focused on the speed component of the velocity, but the direction information was not considered. On the other hand, Long *et al.* [8] introduced Direction-Preserving Trajectory Simplification (DPTS) with its focus on direction component of the velocity. DPTS has a bad performance under E_v as shown in Experiment 6.2, because it does not take into account the time dimension of the trajectory. Different from previous problems, our target is to simplify a trajectory while preserving both speed and direction components of the velocity, more specifically, velocity vector.

Algorithms have been developed to find the optimal and approximate solutions of the min-# and min-ϵ trajectory simplification problems.

Optimal Algs. *Graph-based approach* is one of the popular algorithms to compute the optimal solution of the min-# problem [8]. It has two major steps, namely directed acyclic graph construction and shortest path computation. Let $\epsilon(p_ip_j)$ be the simplification error of segment p_ip_j ($i \le j$). For min-# problem with input trajectory $T = (p_1, p_2, \ldots, p_n)$ and error tolerance ϵ_t, in the graph construction step, a graph with n vertices V_i ($i \in [1,n]$) is created. There should be an edge between two vertices V_i and V_j ($1 \le i < j \le n$) if the simplification error of $p_i p_j$ is within the error tolerance, i.e., $\epsilon(p_i p_j) \le \epsilon_t$. A path from V_i to V_j on the graph corresponds to a ϵ_t-simplification of the sub-trajectory from p_i to p_j. In the shortest path search step, breadth-first search can be applied. The search starts at p_1 and ends at p_n. The shortest path obtained corresponds to a solution T_s of the min-# trajectory simplification problem. p_i ($i \in [1,n]$) is a point of T_s if and only if V_i is on the corresponding shortest path from V_1 to V_n. How this graph-based approach can be adapted to solve the min-# VVPTS problem will be introduced in Section 4.

Approx. Algs. Many trajectory simplification ideas can be adapted to our VVPTS problem to return approximate solutions.Two popular algorithms are greedy [6, 9] and split [9, 3].

Greedy is an approach which sequentially scans the trajectory $T = (p_1, p_2, \ldots, p_n)$ once from the starting point to the ending point and iteratively finds the largest number of consecutive segments which can be discarded and links the two end points of this sub-trajectory until reaching the last point. The time complexity of greedy is $O(n^2)$, because, in the worst case, the simplified trajectory T_s contains only starting and ending points of T. It has to compute $\epsilon_v(p_1p_3), \epsilon_v(p_1p_4), \epsilon_v(p_1p_5), \ldots, \epsilon_v(p_1p_n)$. Since computing $\epsilon_v(p_ip_j)$ takes $O(j-i)$ time, the total time complexity is $O(n^2)$.

Split is an approach which finds a segment in a given trajectory $T = (p_1, p_2, \ldots, p_n)$, where the segment has the largest error under E_v, to split the whole trajectory into two sub-trajectories and recursively continues this process on each sub-trajectories until the sub-trajectory can be simplified by a line segment linking its start point and end point.The time complexity of split is $O(n^2)$, because, in the worst case, it always chooses the second point to split the trajectory.

A *heuristic* algorithm to approximately solve the min-# VVPTS problem in a more efficient way will be introduced in Section 5.

4. FINDING OPTIMAL SOLUTION

4.1 Graph-based Approach

Graph-based approach introduced in Section 3.2 can be directly adapted to obtain the optimal solution of the min-# VVPTS problem. We further illustrate this algorithm by running it on the example trajectory T shown in Figure 1. The coordinate in the 2D Euclidean space and time stamp of each point on the example trajectory can be found in Section 2. The error tolerance ϵ_t is set to be 3. In the graph construction step, a directed acyclic graph with 6 vertices V_i ($i = 1,2,\ldots,6$) is constructed. Each vertex V_i corresponds to a point p_i of T. $\vec{V}(p_ip_j)$ is computed for each pair of points p_i and p_j ($1 \le i < j \le 6$). Based on these computed velocities, $\epsilon_v(p_ip_j)$ is computed for each pair of p_i and p_j. If $\epsilon_v(p_ip_j) \le 3$, we will construct an edge between V_i and V_j. Note that $\epsilon_v(p_ip_{i+1}) = 0$ for $i \in [1,6)$. Computation shows that $\epsilon_v(p_1p_3) = 2$, $\epsilon_v(p_3p_5) = 2.281$ and $\epsilon_v(p_4p_6) = 1.795$. Other simplification errors computed are greater than the error tolerance 3. Therefore, the edges of the graph are (V_1,V_2), (V_1,V_3), (V_2,V_3), (V_3,V_4), (V_3,V_5), (V_4,V_5), (V_4,V_6) and (V_5,V_6). The resulted graph is shown in Figure 2. In the shortest path search step, a shortest path from V_1 to V_6 on the graph is found by breadth-first search. The shortest path found, (V_1,V_3,V_5,V_6), suggests one of the optimal solutions to the problem $T_s = (p_1,p_3,p_5,p_6)$.

Complexity Analysis. In the graph construction step, for each pair of points p_i and p_j ($1 \le i < j \le n$), the simplification error

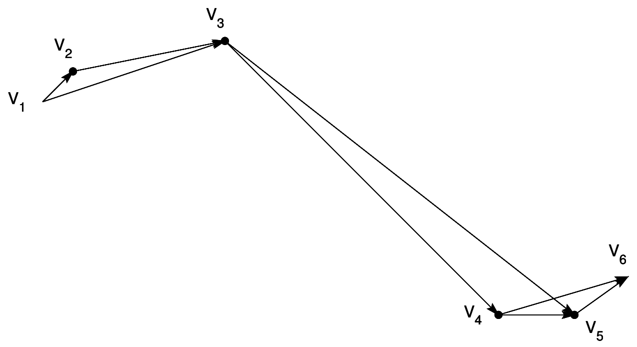


Figure 2: Directed acyclic graph for VVPTS with example trajectory and error tolerance 3

$\epsilon_v(p_ip_j)$ has to be computed. According to the definition of E_v, the time complexity of obtaining $\epsilon_v(p_ip_j)$ is $O(j-i)$ since we need to compute $dist(V_{i-j}, V_{k-(k+1)})$ for each $k \in [i,j)$. Thus, the graph construction step takes $O(n^3)$ time. The shortest path search step takes $O(|V|+|E|)$ time where $|V|$ and $|E|$ denote the number of vertices and the number of edges on the graph. Thus, the time complexity of the naive implementation of the graph-based approach is $O(n^3)$. The algorithm takes $O(|V|+|E|)$ space. Since $|V| = O(n)$ and $|E| = O(n^2)$, the space complexity is $O(n^2)$.

An improvement of the graph-based approach will be introduced in Section 4.2 which proposes a method to reduce the time complexity and space complexity.

4.2 Complexity Improvement

According to the time complexity analysis, graph construction is the dominant step of the graph-based approach. Reducing the time cost of simplification error computation could reduce the time cost of graph construction. The main idea of our method is that when checking whether $\epsilon_v(p_ip_j)$ ($1 \le i < j \le n$) is within ϵ_t, instead of comparing V_{i-j} against each $V_{k-(k+1)}$ for $k \in [i,j)$, we maintain a *feasible velocity area* $I_{i-(j-1)}$ which is defined by $V_{k-(k+1)}$ for $k \in [i,j)$ and ϵ_t, and check whether V_{i-j} falls inside this area. The definition of feasible velocity area is as follows.

DEFINITION 1. *For a segment p_ip_{i+1} ($1 \le i < n$) of T, its* ***feasible velocity area****, denoted by I_{i-i}, is defined to be a disk whose center is $V_{i-(i+1)}$ and radius is ϵ_t. For a potential segment p_ip_j ($1 \le i < j \le n$) of T_s, its* ***feasible velocity area****, denoted by $I_{i-(j-1)}$, is defined to be the intersection of the feasible velocity areas of all p_kp_{k+1} for $k \in [i,j)$, that is, $I_{i-(j-1)} = \bigcap_{k=i}^{j-1} I_{k-k}$.* □

For concise representation, the feasible velocity area I_{i-i} of segment p_ip_{i+1} ($1 \le i < n$) of the original trajectory is denoted by D_i, i.e., D_i is the disk whose center is $V_{i-(i+1)}$ and radius is ϵ_t. The feasible velocity areas D_i of all segments p_ip_{i+1} ($i \in [1,6)$) on the example trajectory T (see Figure 1) are shown in Figure 3. For a potential segment p_2p_6 of T_s, I_{2-5} is the common intersection of D_2, D_3, D_4 and D_5. It can be observed from Figure 3 that the boundaries of I_{2-5} consist of parts of the circumferences of D_2, D_3 and D_5. Note that p_2p_6 cannot be a segment of T_s since our previous computation shows that $\epsilon_v(p_2p_6) > \epsilon_t = 3$. However, in the graph-based approach with complexity improvement, we do not know whether $\epsilon_v(p_2p_6)$ is within the error tolerance before the feasible velocity area of p_2p_6 is obtained.

LEMMA 5. *The simplification error of p_ip_j ($1 \le i < j \le n$) under E_v is within ϵ_t if and only if V_{i-j} falls inside the feasible velocity area of segment p_ip_j.* □

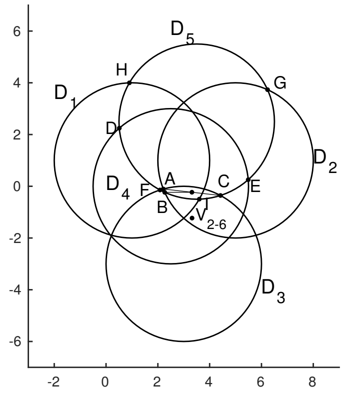


Figure 3: Feasible velocity areas of segments on the example trajectory

PROOF. "if": Assume that V_{i-j} falls inside $I_{i-(j-1)}$. According to the definition of feasible velocity area, V_{i-j} falls inside every D_k for $k \in [i, j)$. Since D_k is a disk with center $V_{k-(k+1)}$ and radius ϵ_t, the distance between V_{i-j} and $V_{k-(k+1)}$ will not exceed ϵ_t. Therefore, $\epsilon_v(p_i p_j) = \max_{i \le k < j} dist(V_{i-j}, V_{k-(k+1)}) \le \epsilon_t$.
"only if": Assume that $\epsilon_v(p_i p_j) \le \epsilon_t$. $dist(V_{i-j}, V_{k-(k+1)}) \le \epsilon_t$ for $k \in [i, j)$. Since D_k is a disk with center $V_{k-(k+1)}$ and radius ϵ_t, V_{i-j} falls inside every D_k for $k \in [i, j)$. According to the definition of feasible velocity area, V_{i-j} falls inside the $I_{i-(j-1)}$. □

According to Lemma 5, checking whether $\epsilon_v(p_i p_j)$ is within ϵ_t is equivalent to checking whether V_{i-j} falls inside $I_{i-(j-1)}$. To construct a complete directed acyclic graph, we need to check whether V_{i-j} belongs to $I_{i-(j-1)}$ for each pair of i, j ($1 \le i < j \le n$). During this process, $O(n^2)$ feasible velocity area computations and point in region checks need to be performed. In Section 4.2.1, we introduce the data structure to store feasible velocity area. In Section 4.2.2, we present an efficient method of obtaining all feasible velocity areas required for the graph construction. In Section 4.2.3, we propose an efficient method of performing point in region check and illustrate how the feasible velocity area computations and point in region checks are combined under $O(n^2 \log n)$ computations.

4.2.1 Data Structure of Feasible Velocity Area

I_{i-j} ($1 \le i < j < n$) can be described by a set of arcs $Arc_k(I_{i-j})$ ($k \in [i, j]$) forming its boundary. $Arc_k(I_{i-j})$ denotes the arc which is a part of the boundary of I_{i-j} and a part of the circumference of D_k. We introduce some notations for the non-empty $Arc_k(I_{i-j})$. $Arc_k(I_{i-j}).P_1$ and $Arc_k(I_{i-j}).P_2$ denote the end points of $Arc_k(I_{i-j})$. Suppose $Arc_k(I_{i-j})$ is an arc from A to B in anticlockwise direction on the circumference of D_k. $Arc_k(I_{i-j}).P_1 = A$ and $Arc_k(I_{i-j}).P_2 = B$. $Arc_k(I_{i-j}).\theta_1$ and $Arc_k(I_{i-j}).\theta_2$ denote the direction of the vector from the center of D_k to A and the direction of the vector from the center of D_k to B, respectively. If $Arc_k(I_{i-j}).\theta_1 = \theta_a$ and $Arc_k(I_{i-j}).\theta_2 = \theta_b$, we can say $Arc_k(I_{i-j}) = [\theta_a, \theta_b]$. $Arc_k(I_{i-j}).K_1$ and $Arc_k(I_{i-j}).K_2$ denote the indexes of the disks whose circumferences intersect with the circumference of D_k at the end points of $Arc_k(I_{i-j})$. Suppose $Arc_k(I_{i-j}) \ne [0, 2\pi]$, $Arc_k(I_{i-j}).\theta_1 \ne Arc_k(I_{i-j}).\theta_2$, $Arc_k(I_{i-j}).P_1$ is one of the intersection points of the circumferences of D_k and D_{k_1} and $Arc_k(I_{i-j}).P_2$ is one of the intersection points of the circumferences of D_k and D_{k_2}. $Arc_k(I_{i-j}).K_1$ is the smallest k_1 and $Arc_k(I_{i-j}).K_2$ is the smallest k_2 ($k_1, k_2 \in [i, k) \cup (k, j]$). When $Arc_k(I_{i-j}) = [0, 2\pi]$ or $Arc_k(I_{i-j}).\theta_1 = Arc_k(I_{i-j}).\theta_2$, $Arc_k(I_{i-j})$ $.K_1 = Arc_k(I_{i-j}).K_2 = k$. Take I_{2-5} in Figure 3 as an example.

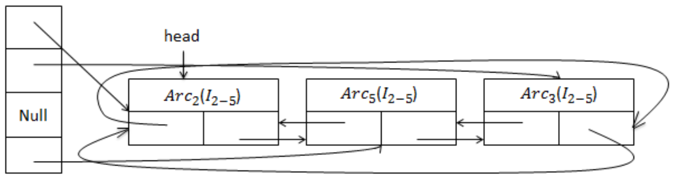


Figure 4: Linked list and reference array after computing I_{2-5}

The arcs which form the boundary of I_{2-5} and thus define I_{2-5} are $Arc_2(I_{2-5})$, $Arc_3(I_{2-5})$ and $Arc_5(I_{2-5})$. $Arc_3(I_{2-5})$ is the arc from C to A in anticlockwise direction on the circumference of D_3. $Arc_3(I_{2-5}).P_1 = C$ and $Arc_3(I_{2-5}).P_2 = A$. $Arc_3(I_{2-5}).\theta_1 = \theta(\overrightarrow{P_{3-4}C}) = 61.80$ and $Arc_3(I_{2-5}).\theta_2 = \theta(\overrightarrow{P_{3-4}A}) = 105.25$. Thus, $Arc_3(I_{2-5}) = [61.80, 105.25]$. C is one of the intersection points of the circumferences of D_3 and D_5, and A is one of the intersection points of the circumferences of D_3 and D_2. $Arc_3(I_{2-5}).K_1 = 5$ and $Arc_3(I_{2-5}).K_2 = 2$.

For a feasible velocity area I_{i-j}, we construct a circular doubly linked list to store its boundary. Each node of the list corresponds to an arc. In the node, the data field stores the arc information including its P_1, P_2, θ_1, θ_2 and K_1, K_2. The next link and previous link are references to the nodes which correspond to the next and previous arcs in anticlockwise direction of the arc represented by the current node on the boundary of I_{i-j}, respectively. For quick access to the nodes in a list, an array whose elements are references to the list nodes will be maintained. If the left-to-right computation illustrated in Section 4.2.2 is adopted, when computing I_{i-j}, the size of the array will be $n - i$ and the k^{th} ($k \in [1, j - i + 1]$) element of the array is a reference to the node corresponding to $Arc_{i+k-1}(I_{i-j})$ in the list. Consider I_{2-5} of our example trajectory. After the computation of I_{2-5} (the computation of feasible velocity areas will be introduced in Section 4.2.2), the circular doubly linked list and the reference array are shown in Figure 4.

4.2.2 Left-to-Right Computation

For an input trajectory $T = (p_1, p_2, \ldots, p_n)$, the feasible velocity areas required for graph construction are shown in Table 2. $I_{i-(i+1)}$ ($1 \le i < n$) is the intersection of two disks and can be obtained in constant time. Since $I_{i-j} = I_{i-(j-1)} \cap D_j$, I_{i-j} can be calculated from known $I_{i-(j-1)}$. I_{i-j} is obtained by "cutting" $I_{i-(j-1)}$ with $Arc_j(I_{i-j})$. Regarding our example trajectory, suppose I_{2-4} whose boundary consists of $Arc_2(I_{2-4})$, $Arc_4(I_{2-4})$ and $Arc_3(I_{2-4})$, and $Arc_5(I_{2-5})$ which is the arc from B to C in anticlockwise direction on the circumference of D_5 are known. Intuitively, I_{2-5} can be obtained by cutting I_{2-4} with $Arc_5(I_{2-5})$. Since $Arc_5(I_{2-5}).K_1 = 2$ and $Arc_5(I_{2-5}).K_2 = 3$. $Arc_2(I_{2-4})$ and $Arc_3(I_{2-4})$ are cut and $Arc_5(I_{2-5})$ becomes a part of the boundary of the new feasible velocity area. How this cutting process can be strictly implemented will be introduced in this section. Note that in actual implementation, the data of $I_{i-(j-1)}$ will be updated to the data of I_{i-j}, which means that $I_{i-(j-1)}$ will not be maintained when I_{i-j} is available. Therefore whether V_{i-j} falls inside $I_{i-(j-1)}$ may need to be decided before the computation of I_{i-j}. An efficient approach to perform point in region check will be introduced in Section 4.2.3.

This section focuses on the computation of feasible velocity area. As is introduced, we would like to compute the unknown I_{i-j} from known $Arc_j(I_{i-j})$ and $I_{i-(j-1)}$. Our first step is to make related arcs which may be used in this process available. Since we need to compute I_{i-j} for all pairs of i, j ($1 \le i < j < n$) for graph construction, the potentially useful arcs are $Arc_j(I_{i-j})$

$I_{1-2}, I_{1-3}, I_{1-4}, \ldots, I_{1-(n-2)}, I_{1-(n-1)}$
$I_{2-3}, I_{2-4}, \ldots, I_{2-(n-2)}, I_{2-(n-1)}$
$I_{3-4}, \ldots, I_{3-(n-2)}, I_{3-(n-1)}$
$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$
$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$
$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$
$I_{(n-2)-(n-1)}$

Table 2: Feasible velocity areas required for graph construction (from the last row to the first row, from left to right in each row)

Disk	Arcs
D_2	$Arc_2(I_{1-2})$
D_3	$Arc_3(I_{1-3})$, $Arc_3(I_{2-3})$
D_4	$Arc_4(I_{1-4})$, $Arc_4(I_{2-4})$, $Arc_4(I_{3-4})$
......	......
D_{n-2}	$Arc_{n-2}(I_{1-(n-2)})$, $Arc_{n-2}(I_{2-(n-2)})$, ..., $Arc_{n-2}(I_{(n-3)-(n-2)})$
D_{n-1}	$Arc_{n-1}(I_{1-(n-1)})$, $Arc_{n-1}(I_{2-(n-1)})$, ..., $Arc_{n-1}(I_{(n-3)-(n-1)})$, $Arc_{n-1}(I_{(n-2)-(n-1)})$

Table 3: Arcs computed before the computation of feasible velocity area (from the last column to the first column)

for $1 \le i < j < n$ (see Table 3). For each arc, we record its P_1, P_2, θ_1, θ_2 and K_1, K_2. The potentially useful arcs which are on the circumference of the same disk can be calculated by intersect operation. For $k \in (1, n)$, $Arc_k(I_{(k-1)-k})$ involves only two disks and can be obtained in constant time. Then, $Arc_k(I_{i-k})$ where i takes the value from $k-2$ to 1 can be calculated in sequence. Here we introduce two extra notations. $I_{i\&j}$ denotes the common intersection of D_i and D_j. $Arc_k(I_{i\&j})$ ($k = i$ or $k = j$) denotes the arc which is a part of the boundary of $I_{i\&j}$ and a part of the circumference of D_k. The computation of potentially useful arcs is based on the equation $Arc_k(I_{i-k}) = Arc_k(I_{(i+1)-k}) \cap Arc_k(I_{i\&k})$. In a special case where D_i and D_j overlap, $Arc_j(I_{i\&j} = Arc_i(I_{i\&j})$ $= [0, 2\pi]$. $Arc_k(I_{i\&k})$ involves only two disks and can be obtained in constant time. The intersect operation takes constant time. Thus, if $Arc_k(I_{(i+1)-k})$ is known, $Arc_k(I_{i-k})$ can be computed in constant time. From known $Arc_k(I_{(k-1)-k})$, we can compute $Arc_k(I_{(k-2)-k})$, $Arc_k(I_{(k-3)-k})$, ..., $Arc_k(I_{1-k})$ in sequence by intersect operations and the total time cost is $O(k)$. The time complexity of computing all arcs required is $O(n^2)$. Consider back the example trajectory. The fourth row of the arc table, i.e., the potentially useful arcs on the circumference of D_5, can be computed as follows: $Arc_5(I_{4-5}) = \widehat{DE}$, $Arc_5(I_{3-5}) = Arc_5(I_{4-5}) \cap Arc_5(I_{3\&5}) = \widehat{FC}$, $Arc_5(I_{2-5}) = Arc_5(I_{3-5}) \cap Arc_5(I_{2\&5}) = \widehat{BC}$, $Arc_5(I_{1-5}) = Arc_5(I_{2-5}) \cap Arc_5(I_{1\&5}) = \widehat{BI}$.

The second step is to compute the feasible velocity areas in Table 2. If we take a closer look at Table 2 and Table 3 simultaneously, we can find that all related arcs used by i^{th} $(1 \le i < n-1)$ row of Table 2 are those from i^{th} $(1 \le i < n-1)$ column of Table 3. Therefore, we only need to keep an array of size $n-2$ to save information of arcs which are from i^{th} column of Table 3 when we compute i^{th} row of Table 2. The order of computation for Table 2 is from the last row to the first row, from left to right in each row. At the same time, we update the array of related arcs accordingly.

As is introduced in Section 4.2.1, a circular doubly linked list and a reference array will be used to compute and store a feasible velocity area. For each row of the feasible velocity area table, we keep updating the same circular doubly linked list and the reference array as we compute the feasible velocity areas from left to right. At the beginning of the i^{th} $(1 \le i < n-1)$ row, the process of computing $I_{i-(i+1)}$ and initiating the linked list and reference array is as follows.
(1) If $Arc_{i+1}(I_{i-(i+1)})$ is empty, $I_{i-(i+1)}$ will be empty and no linked list and reference array will be constructed.
(2) If $Arc_{i+1}(I_{i-(i+1)})$ is non-empty, compute $Arc_i(I_{i-(i+1)})$ and construct an array with $n-i$ elements.
(a) If $Arc_{i+1}(I_{i-(i+1)}).\theta_1 = Arc_{i+1}(I_{i-(i+1)}).\theta_2$ or $Arc_{i+1}(I_{i-(i+1)})$ $= [0, 2\pi]$, $I_{i-(i+1)}$ will be a point or a disk. Construct a circular doubly linked list with one node. The data field of the node stores the information of $Arc_i(I_{i-(i+1)})$. The first element of the reference array is a reference to the node of $Arc_i(I_{i-(i+1)})$.
(b) If $Arc_{i+1}(I_{i-(i+1)}).\theta_1 \neq Arc_{i+1}(I_{i-(i+1)}).\theta_2$ and $Arc_{i+1}(I_{i-(i+1)})$ $\neq [0, 2\pi]$, construct a circular doubly linked list with two nodes. The data fields of the first and second nodes store the information of $Arc_i(I_{i-(i+1)})$ and $Arc_{i+1}(I_{i-(i+1)})$, respectively. The first two elements of the reference array are references to the node of $Arc_i(I_{i-(i+1)})$ and the node of $Arc_{i+1}(I_{i-(i+1)})$, respectively.

This initiation takes constant time. The process of computing I_{i-j} from known $I_{i-(j-1)}$ $(i+1 < j < n)$ is as follows.
(1) If the current linked list is empty, i.e., $I_{i-(j-1)}$ is empty, I_{i-j} will be empty. There will be no update to the linked list and reference array.
(2) If the current linked list is a point, check whether this point is inside D_j.
(a) If the point is inside D_j, I_{i-j} will still be a point.
(b) If the point is outside D_j, I_{i-j} will be empty.
(3) If the current linked list is non-empty and not a point, check $Arc_j(I_{i-j})$ from the arc table.
(a) If $Arc_j(I_{i-j})$ is empty, $I_{i-(j-1)}$ will be either completely inside or completely outside D_j. Find any point which is an end point of the arc represented by any node in the current linked list and check it against D_j. If the point belongs to D_j, $I_{i-j} = I_{i-(j-1)}$ and there will be no update to the linked list and reference array. If the point does not belong to D_j, I_{i-j} will be empty, delete all nodes in the linked list and set all elements of the reference array to null.
(b) If $Arc_j(I_{i-j}) = [0, 2\pi]$, $I_{i-j} = I_{i-(j-1)}$ will be a disk and there will be no update to the linked list and reference array.
(c) If $Arc_j(I_{i-j}).\theta_1 = Arc_j(I_{i-j}).\theta_2$, i.e., $Arc_j(I_{i-j})$ is a point, $I_{i-(j-1)}$ will be either completely inside D_j or overlap with D_j at one point. Find any point which is inside $I_{i-(j-1)}$. If such point does not belong to D_j, I_{i-j} will become a point. Save this point. Delete all nodes in the linked list and set all elements of the reference array to null. If such point belongs to D_j, $I_{i-j} = I_{i-(j-1)}$ and there will be no update to the linked list and reference array.
(d) If non-empty $Arc_j(I_{i-j}).\theta_1 \neq Arc_j(I_{i-j}).\theta_2$ and $Arc_j(I_{i-j}) \neq$ $[0, 2\pi]$, build a node for $Arc_j(I_{i-j})$ with its previous and next links pointing to itself and update the $j-i+1^{th}$ element of the reference array to the address of the node. Let $p_1 = Arc_j(I_{i-j}).P_1$, $p_2 = Arc_j(I_{i-j}).P_2$, $k_1 = Arc_j(I_{i-j}).K_1$, $k_2 = Arc_j(I_{i-j}).K_2$. Get from the $k_1 - i + 1^{th}$ and the $k_2 - i + 1^{th}$ elements of the reference array the node of $Arc_{k_1}(I_{i-j})$ and the node of $Arc_{k_2}(I_{i-j})$. Starting from the node of $Arc_{k_1}(I_{i-j})$, traverse the doubly linked list through the next links of the nodes until $Arc_{k_2}(I_{i-j})$ is reached, delete all nodes encountered between the nodes of $Arc_{k_1}(I_{i-j})$ and $Arc_{k_2}(I_{i-j})$ (exclusively) and update corresponding elements in the reference array to null. Update the P_2, θ_2, K_2 and next link of the node of $Arc_{k_1}(I_{i-j})$ to p_1, the direction of $\overrightarrow{V_{k_1-(k_1+1)}p_1}$, j and the address of the node of $Arc_j(I_{i-j})$. Update the P_1, θ_1, K_1 and previous link of the node of $Arc_{k_2}(I_{i-j})$ to p_2, the direction

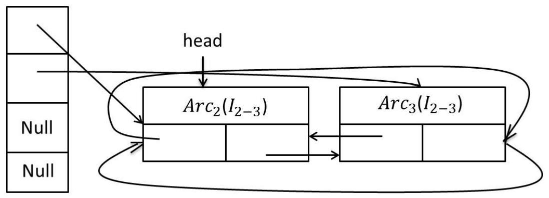


Figure 5: Linked list and reference array after computing I_{2-3}

of $\overrightarrow{V_{k_2-(k_2+1)}p_2}$, j and the address of the node of $Arc_j(I_{i-j})$. Update previous and next links of the node of $Arc_j(I_{i-j})$ to the addresses of the nodes of $Arc_{k_1}(I_{i-j})$ and $Arc_{k_2}(I_{i-j})$, respectively.

We continue to use the example trajectory to illustrate how a row of the feasible velocity area table is computed. Suppose the arc table is already available and we are now computing the second row of the feasible velocity area table. The process of computing I_{2-3} and initiating the linked list and reference array is as follows: since $Arc_3(I_{2-3})$ is non-empty, compute $Arc_2(I_{2-3})$ and construct an array with 4 elements. Since $Arc_3(I_{2-3}).\theta_1 \neq Arc_3(I_{2-3}).\theta_2$ and $Arc_3(I_{2-3}) \neq [0,2\pi]$, construct a circular doubly linked list with 2 nodes. The data fields of the first and second nodes store the information of $Arc_2(I_{2-3})$ and $Arc_3(I_{2-3})$, respectively. The first and second elements of the reference array are references to the node of $Arc_2(I_{2-3})$ and the node of $Arc_3(I_{2-3})$, respectively. The linked list and reference array after this step are shown in Figure 5.

We then compute I_{2-4} from I_{2-3} and $Arc_4(I_{2-4})$: since the current linked list is non-empty, $Arc_4(I_{2-4}).\theta_1 \neq Arc_4(I_{2-4}).\theta_2$ and $Arc_4(I_{2-4}) \neq [0,2\pi]$, build a node for $Arc_4(I_{2-4})$ with its previous and next links pointing to itself and update the third element of the reference array to the address of the node. Let $p_1 = Arc_4(I_{2-4}).P_1$ and let $p_2 = Arc_4(I_{2-4}).P_2$. $Arc_4(I_{2-4}).K_1 = 2$ and $Arc_4(I_{2-4}).K_2 = 3$, get from the first and second elements of the reference array the node of $Arc_2(I_{2-4})$ and the node of $Arc_3(I_{2-4})$. Since the next link of the node of $Arc_2(I_{2-4})$ is a reference to the node of $Arc_3(I_{2-4})$, there will be no delete operation. Update the P_2, θ_2, K_2 and next link of the node of $Arc_2(I_{2-4})$ to p_1, $\theta(\overrightarrow{V_{2-3}p_1})$, 4 and the address of the node of $Arc_4(I_{2-4})$. Update the P_1, θ_1, K_1 and previous link of the node of $Arc_3(I_{2-4})$ to p_2, $\theta(\overrightarrow{V_{3-4}p_2})$, 4 and the address of the node of $Arc_4(I_{2-4})$. Update the previous and next links of the node of $Arc_4(I_{2-4})$ to the addresses of the nodes of $Arc_2(I_{2-4})$ and $Arc_3(I_{2-4})$, respectively. The linked list and reference array after computing I_{2-4} are shown in Figure 6.

The computation of I_{2-5} from I_{2-4} and $Arc_5(I_{2-5})$ is similar: since the current linked list is non-empty, $Arc_5(I_{2-5}) \neq [0,2\pi]$ and $Arc_5(I_{2-5}).\theta_1 \neq Arc_5(I_{2-5}).\theta_2$, build a node for $Arc_5(I_{2-5})$ with its previous and next links pointing to itself and update the fourth element of the reference array to the address of the node. $Arc_5(I_{2-5}).P_1 = B$, $Arc_5(I_{2-5}).P_2 = C$, $Arc_5(I_{2-5}).K_1 = 2$ and $Arc_5(I_{2-5}) = 3$. Get from the first and second elements of the reference array the node of $Arc_2(I_{2-5})$ and the node of $Arc_3(I_{2-5})$. The next link of the node of $Arc_2(I_{2-5})$ is a reference to the node of $Arc_4(I_{2-5})$, and the next link of the node of $Arc_4(I_{2-5})$ is a reference to the node of $Arc_3(I_{2-5})$. Thus, delete the node of $Arc_4(I_{2-5})$ and update the third element of the reference array to null. Update the P_2, θ_2, K_2 and next link of the node of $Arc_2(I_{2-5})$ to B, $\theta(\overrightarrow{V_{2-3}B})$, 5 and the address of the node of $Arc_5(I_{2-5})$. Update the P_1, θ_1, K_1 and previous link of the node of $Arc_3(I_{2-5})$ to C, $\theta(\overrightarrow{V_{3-4}C})$, 5 and the address of the node of $Arc_5(I_{2-5})$. Update the previous and next links of the node of $Arc_5(I_{2-5})$ to the addresses of the nodes of $Arc_2(I_{2-5})$ and $Arc_3(I_{2-5})$, respectively. The linked list and reference array after computing I_{2-5} are shown in Figure 4.

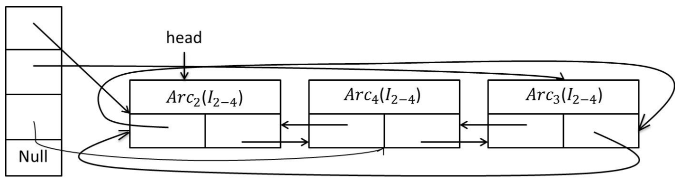


Figure 6: Linked list and reference array after computing I_{2-4}

LEMMA 6. *For* $1 \leq i \leq k \leq j < n$, *if* $Arc_k(I_{i-j})$ *is empty*, $Arc_k(I_{i-t})$ *for any* $t \in (j,n)$ *will also be empty.* □

PROOF. If no arc on the circumference of D_k forms a part of the boundary of non-empty I_{i-j}, D_k should contain I_{i-j}. Since $I_{i-t} = I_{i-j} \cap I_{j-t}$, I_{i-j} contains I_{i-t}. It can be derived that D_k contains I_{i-t}. No arc on the circumference of D_k will become a part of the boundary of I_{i-t}. That is, $Arc_k(I_{i-t})$ will be empty. □

Complexity Analysis. As is analyzed before, the time cost of getting all potentially useful arcs is $O(n^2)$. Regarding the computation of feasible velocity areas, if the operations of deleting nodes from the linked list and setting corresponding elements in the reference array to null are not counted, getting I_{i-j} from known $I_{i-(j-1)}$ will take constant time. During the computation of the i^{th} row of feasible velocity area table, according to Lemma 6, if the node of arc on the circumference of a disk D_k is deleted from the linked list, it will not be inserted back. This indicates that the node of arc from each disk will be removed from the linked list at most once, and each element in the reference array will be set to null at most once. Thus, when computing the i^{th} row of the feasible area table, the total time cost as well as the time cost of deleting nodes from the linked list and setting elements in the reference array to null are $O(n-i)$. The time complexity of getting all feasible velocity areas required is $O(n^2)$. In terms of space complexity, $O(n)$ space is required to store the information of all related arcs which are from a specific column of Table 3. We do not need to "remember" all computed feasible velocity areas, $O(n)$ space is required to store the current feasible velocity area. The space cost is therefore $O(n)$ during the computation of arc table and feasible velocity areas.

4.2.3 Point in Convex Region Check

Since a feasible velocity area is either a disk or the intersection of disks, it is convex. To decide whether a point falls inside a convex region bounded by n circular arcs, the naive approach is to test the point against each arc, which takes $O(n)$ time. Shamos[11] proposed a method to decide whether a point is interior or exterior to a convex polygon with n sides in $O(\log n)$ time after $O(n)$ time preprocessing. The preprocessing chooses any point R which is interior to the polygon, and divides the plane into n sectors by drawing n rays through the vertices of the polygon originating at R. After the preprocessing, given a point P, the sector P belongs to can be decided in $O(\log n)$ time by binary search. Suppose P belongs to sector S, whether P is interior to the polygon can be determined by testing P against the side of the polygon which is contained in S, and this takes constant time. Therefore, after spending $O(n)$ time to perform the preprocessing on a convex polygon with n sides, checking whether a point is interior to that polygon takes $O(\log n)$. We prove that this method can also solve the inclusion problem of convex region. Two issues need to be addressed when adapting it to construct the directed acyclic graph desired in VVPTS problem since instead of having a fixed convex region, we have a distinct feasible velocity area $I_{i-(j-1)}$ for each point V_{i-j} $(1 \le i < j < n)$: first, the preprocessing has to be reusable; second, the data structure has to allow efficient search and update.

Section 4.2.2 introduces how the i^{th} $(1 \le i < n)$ row of feasible velocity areas in Table 2 can be computed in $O(n-i)$ time, after the $O(n)$ time computation of arcs in Table 3. The aim of computing I_{i-j} is to decide whether $V_{i-(j+1)}$ falls inside I_{i-j}. We would like to make the preprocessing on I_{i-j} for $j \in (i,n)$ reusable for the i^{th} row of the feasible velocity area table. To achieve this goal, for the i^{th} row of the feasible velocity area table, the reference point R_i is fixed. That is, the same R_i is used in the test of $V_{i-(j+1)}$ against I_{i-j} for all $j \in (i,n)$. R_i should be interior to every non-empty feasible velocity area in the i^{th} row. To get R_i, we perform the left-to-right computation of the i^{th} row of Table 2, stop at the smallest non-empty feasible velocity area and define R_i to be any point which is inside this convex region. R_i can be decided in constant time when the smallest non-empty feasible velocity area I_{i-j} is known. If I_{i-j} is a point, R_i will be that point. If I_{i-j} is not a point, we may define R_i to be the middle point of a straight line whose end points are the end points of an arc which is part of the boundary of I_{i-j}. When the arc table is available, the time cost of getting R_i is dominated by the time cost of computing the smallest feasible velocity area in the i^{th} row of Table 2, which is $O(n-i)$. Consider the second raw of the feasible velocity area table of the example trajectory, to get R_2, firstly, I_{2-3}, I_{2-4} and I_{2-5} are computed by the left-to-right method. The smallest non-empty feasible velocity area is I_{2-5} and R_2 can be defined as the middle point of the line whose end points are A and C. The coordinate of R_2 is $(3.31, 0.23)$. As is shown in Figure 3, 3 rays which originate at R_2 and pass through A, C or B divide the plane into 3 sectors.

After R_i is fixed, when checking whether $V_{i-(j+1)}$ falls inside I_{i-j}, the plane is divided into sectors by rays which originate at R_i and pass through the end points of the arcs forming the boundary of I_{i-j}. The sectors can be stored in a binary search tree to facilitate efficient search and update. A sector which contains non-empty $Arc_k(I_{i-j})$ $(k \in [i,j])$ is stored in one or two nodes of the tree. Let $p_1 = Arc_k(I_{i-j}).P_1$ and $p_2 = Arc_k(I_{i-j}).P_2$. $Arc_k(I_{i-j}).\alpha_1$ denotes the direction of $\overrightarrow{R_i p_1}$ and $Arc_k(I_{i-j}).\alpha_2$ denotes the direction of $\overrightarrow{R_i p_2}$. Let $\alpha_{k_1} = Arc_k(I_{i-j})\ .\alpha_1$ and let $\alpha_{k_2} = Arc_k(I_{i-j}).\alpha_2$. If $\alpha_{k_1} \le \alpha_{k_2}$, the sector of $Arc_k(I_{i-j})$ will be stored in one node whose key is the interval of angle $[\alpha_{k_1}, \alpha_{k_2}]$. If $\alpha_{k_1} > \alpha_{k_2}$, i.e., the x-axis crosses the sector of $Arc_k(I_{i-j})$, the sector will be stored in two nodes whose keys are $[\alpha_{k_1}, 2\pi]$ and $[0, \alpha_{k_2}]$. The keys of nodes in the tree cover $[0, 2\pi]$ and have no overlap with each other except on the boundaries. The node also stores the index of the disk to which the arc of the sector belongs. For quick access to the nodes in the binary tree, an array whose elements are references to the tree nodes will be maintained. When testing $V_{i-(j+1)}$ against I_{i-j}, the size of the array will be $2(n-i)$ and the $2k-1^{th}$ and $2k^{th}$ elements $(k \in [1, j-i+1])$ are reference to the nodes of $Arc_{i+k-1}(I_{i-j})$ in the tree. Consider the example trajectory, Figure 3 shows the partition of the plane when deciding whether V_{2-6} is interior to I_{2-5}. Figure 7 shows the corresponding binary search tree and reference array.

Now we are ready to discuss the initiation and update of the binary search tree and its reference array in the left-to-right computation of feasible velocity area and test of point against the area. For each row of the feasible velocity area table, we keep updating the same binary search tree and reference array as we test $V_{i-(j+1)}$ against I_{i-j} from $j = i+1$ to $j = n-1$. Note that the binary search tree and tree reference aim to facilitate point in region checks while the circular doubly linked list and list reference array illustrated in Section 4.2.2 aim to compute feasible velocity areas. Since the data structures do not maintain computed feasible velocity areas, the computation of feasible velocity area and point in region check need to be done simultaneously after the reference point of the corresponding row is available. In this section, we present the operations on the binary search tree and tree reference array. Assume that the information of potentially useful arcs (see Table 3) is available. For $i \in [1, n-1)$, after obtaining R_i, at the beginning of the i^{th} row of the feasible velocity area table, the process of initiating the binary search tree and its reference array is as follows.

(1) If $Arc_{i+1}(I_{i-(i+1)})$ is empty, $I_{i-(i+1)}$ will be empty and no binary search tree and reference array will be constructed.

(2) If $Arc_{i+1}(I_{i-(i+1)})$ is not empty, compute $Arc_i(I_{i-(i+1)})$ and construct a reference array with $2(n-i)$ elements.

(a) If $Arc_{i+1}(I_{i-(i+1)}) = [0, 2\pi]$, $I_{i-(i+1)} = D_i$ will be a disk. Construct a binary search tree with two nodes. The keys are $[\alpha, 2\pi]$ and $[0, \alpha]$ where $\alpha = Arc_{i+1}(I_{i-(i+1)}).\alpha_1 = Arc_{i+1}(I_{i-(i+1)})\ .\alpha_2$. The disk index of the nodes is i. The first and second elements of the reference array are references to the two nodes.

(b) If $Arc_{i+1}(I_{i-(i+1)}).\theta_1 = Arc_{i+1}(I_{i-(i+1)}).\theta_2$, $I_{i-(i+1)}$ will be a point. Construct a binary search tree with one node. The key of the node is $[\alpha, \alpha]$, $\alpha = Arc_{i+1}(I_{i-(i+1)})\ .\alpha_1 = Arc_{i+1}(I_{i-(i+1)}).\alpha_2$. The disk index of the node is i. The first element of the reference array is a reference to the only node in the current tree.

(c) If $Arc_{i+1}(I_{i-(i+1)}).\theta_1 \ne Arc_{i+1}(I_{i-(i+1)}).\theta_2$ and $Arc_{i+1}(I_{i-(i+1)}) \ne [0, 2\pi]$. Build a binary search tree with two or three nodes corresponding to the whole or part of the two sectors which contain $Arc_i(I_{i-(i+1)})$ and $Arc_{i+1}(I_{i-(i+1)})$ respectively. The keys of the nodes are intervals of angles whose boundary can be 0, 2π $Arc_{i+1}(I_{i-(i+1)}).\alpha_1$ or $Arc_{i+1}(I_{i-(i+1)}).\alpha_2$. The disk indexes of the nodes are i and $i+1$. The first two elements of the reference array are references to nodes with disk index i, and the third and fourth elements are references to the nodes with disk index $i+1$.

This initiation takes constant time. When $I_{i-(j-1)}$ is known, after performing binary search to test V_{i-j} against $I_{i-(j-1)}$, the process of updating the binary search tree and its reference array from $I_{i-(j-1)}$ to I_{i-j} $(i+1 < j < n)$ is as follows.

(1), (2)The condition and the operation is consistent with 4.2.2.

(3) If the current linked list is non-empty and not a point, check $Arc_j(I_{i-j})$ from the arc table.

If non-empty $Arc_j(I_{i-j}) \ne [0, 2\pi]$ and $Arc_j(I_{i-j}).\theta_1 \ne Arc_j(I_{i-j})\ .\theta_2$, let $p_1 = Arc_j(I_{i-j}).P_1$, $p_2 = Arc_j(I_{i-j}).P_2$, $k_1 = Arc_j(I_{i-j}).K_1$, $k_2 = Arc_j(I_{i-j}).K_2$. Get from the $k_1 - i + 1^{th}$ and the $k_2 - i + 1^{th}$ elements of the list reference array the node of $Arc_{k_1}(I_{i-j})$ and the node of $Arc_{k_2}(I_{i-j})$ in the current linked list. Starting from the node of $Arc_{k_1}(I_{i-j})$, traverse the linked list through the next links of the nodes until $Arc_{k_2}(I_{i-j})$ is reached. For any list node of $Arc_k(I_{i-j})$ encountered between the nodes of $Arc_{k_1}(I_{i-j})$ and $Arc_{k_2}(I_{i-j})$ (exclusive, i.e., $k \ne k_1$ and $k \ne k_2$), delete the tree nodes with disk index k and update corresponding elements in the tree reference array to null. Update the keys of the tree nodes with index k_1 from $[Arc_{k_1}(I_{i-(j-1)}).\alpha_1, Arc_{k_1}(I_{i-(j-1)}).\alpha_2]$ or $[Arc_{k_1}(I_{i-(j-1)}).\alpha_1, 2\pi]$ and $[0, Arc_{k_1}(I_{i-(j-1)}).\alpha_2]$ to $[Arc_{k_1}(I_{i-j})\ .\alpha_1, Arc_{k_1}(I_{i-j}).\alpha_2]$ or $[Arc_{k_1}(I_{i-j}).\alpha_1, 2\pi]$ and $[0, Arc_{k_1}(I_{i-j}).\alpha_2]$ where $Arc_{k_1}(I_{i-j}).\alpha_1 = Arc_{k_1}(I_{i-(j-1)}).\alpha_1$ and $Arc_{k_1}(I_{i-j}).\alpha_2$

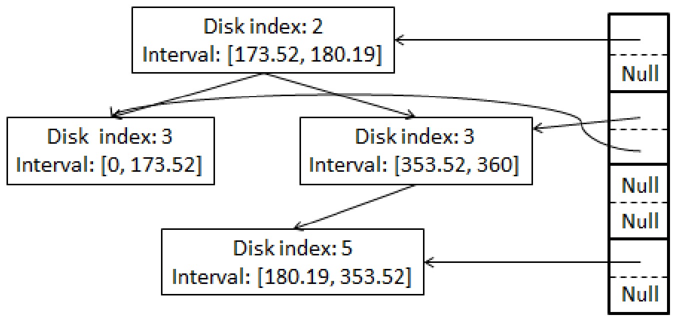


Figure 7: Binary search tree and reference array when solving the inclusion problem of I_{2-5} (unit: degree)

is the direction of $\overrightarrow{R_i p_1}$. Update the keys of the tree nodes with index k_2 symmetrically where $Arc_{k_2}(I_{i-j}).\alpha_2 = Arc_{k_2}(I_{i-(j-1)}).\alpha_2$ and $Arc_{k_2}(I_{i-j}).\alpha_1$ is the direction of $\overrightarrow{R_i p_2}$. The update may involve delete of nodes. Note that $Arc_j(I_{i-j}).\alpha_1$ and $Arc_j(I_{i-j}).\alpha_2$ are the directions of $\overrightarrow{R_i p_1}$ and $\overrightarrow{R_i p_2}$, respectively. If $Arc_j(I_{i-j}).\alpha_1 \le Arc_j(I_{i-j}).\alpha_2$, build a node with key $[Arc_j(I_{i-j}).\alpha_1, Arc_j(I_{i-j}).\alpha_2]$; else, construct two tree nodes with keys $[Arc_j(I_{i-j}).\alpha_1, 2\pi]$ and $[0, Arc_j(I_{i-j}).\alpha_2]$. The disk index of the nodes is j. The $2(j-i)+1^{th}$ and the $2(j-i+1)^{th}$ elements of the tree reference array are references to the newly constructed tree nodes. Insert the newly constructed tree nodes into the tree.

Complexity Analysis. For $i \in [1, n-1)$, during the left-to-right computation of feasible velocity areas I_{i-j} for $j \in (i, n)$, there are at most $j-i+2$ nodes in the binary search tree. The time cost of deleting a node from the tree and inserting a node into the tree are $O(\log(j-i))$. The time cost of updating an element in the reference array and constructing a new node are $O(1)$. When computing I_{i-j} from $I_{i-(j-1)}$, after relevant nodes have been removed, the update of keys of tree nodes will not cause the violation of the binary search tree property since the spread of the angular interval (i.e., the key) of the nodes involved will be reduced. Thus the time cost of changing a key is $O(1)$. According to Lemma 6, the nodes with index k ($k \in [i, n)$) will be constructed, inserted into the tree and deleted from the tree at most once, and the corresponding elements of the tree reference array will be updated at most twice. Therefore, when computing the i^{th} row of Table 2, regarding the binary search tree and its reference array, the time cost of delete and insert operations are $O((n-i)\log(n-i))$, the time cost of updating the reference array, constructing new nodes and updating keys of nodes are $O(n-i)$ and the total time cost is $O((n-i)\log(n-i))$. The time complexity of getting all feasible velocity areas and corresponding trees and tree reference arrays is therefore $O(n^2 \log n)$. The space cost of the binary search tree and tree reference array is $O(n)$.

After the binary search tree of I_{i-j} is constructed, checking whether $V_{i-(j+1)}$ falls inside I_{i-j} involves the following steps: (1) Compute the direction of $\overrightarrow{R_i V_{i-(j+1)}}$. (2) Search on the tree of I_{i-j} for the node whose interval of angle contains the computed direction in Step (1). (3) Get the disk index k ($k \in [i, j]$) of the node found in Step (2) and test $V_{i-(j+1)}$ against the D_k. If $V_{i-(j+1)}$ belongs to D_k, $V_{i-(j+1)}$ will fall inside I_{i-j}; else, $V_{i-(j+1)}$ will be outside I_{i-j}. For example, Figure 3 shows the partition of the plane when deciding whether V_{2-6} is interior to I_{2-5} and Figure 7 shows the corresponding binary search tree and reference array. Suppose R_2 is set to be $(3.31, 0.23)$ according to previous computation. From the data provided in Section 2, the coordinate of V_{2-6} is computed to be $(3.32, -1.23)$. The direction of $\overrightarrow{R_2 V_{2-6}}$ is computed to be 270.32. The node on the tree whose interval of angle contains 270.32 degree has an index 5. We then check V_{2-6} against D_5. Since $dist(V_{5-6}, V_{2-6}) = 3.73 > \epsilon_t = 3$, it can be concluded that V_{2-6} falls outside I_{2-5}. Instead of checking V_{2-6} against each of D_2, D_3, D_4 and D_5, we perform a binary search and check V_{2-6} only against D_5. The time complexity is reduced. The time complexity of checking whether $V_{i-(j+1)}$ falls inside known I_{i-j} is $O(\log(j-i))$. The time complexity of performing all points in known convex region tests is $O(n^2 \log n)$.

4.2.4 Construction of Points Linkage and Complexity

Algorithm 1 Graph construction with complexity improvement

```
Input: Trajectory $T = (p_1, p_2, ..., p_n)$ and error tolerance $\epsilon_t$
Output: Graph $G$
  // Step 1
  $p_n.step = 0$, $p_{n-1}.step = 1$, $p_{n-1}.next = p_n$
  for $i \leftarrow n-2$ to 1 do
    // Step 2
    compute $Arc_i(I_{i-i+1})$ and $Arc_{i+1}(I_{i-i+1})$
    for $j \leftarrow i+2$ to $n-1$ do
      compute $Arc_j(I_{i-j})$
    $p_i.step = p_{i+1}.step + 1$, $p_i.next = p_{i+1}$
    // Step 3
    $j \leftarrow i+1$
    compute $I_{i-j}$, initiate the cdll and lra
    while $j < n-1$ and $I_{i-j} \ne \emptyset$ do
      $j \leftarrow j+1$
      compute $I_{i-j}$, update cdll and lra
    find reference point $R_i$
    release cdll
    // Step 4
    $j \leftarrow i+1$
    compute $I_{i-j}$, initiate the cdll, bst, lra and tra
    while $j < n$ and $I_{i-j} \ne \emptyset$ do
      check if $V_{i-(j+1)}$ is interior to $I_{i-j}$ by binary search
      if $V_{i-(j+1)}$ is interior to $I_{i-j}$ and $p_{j+1}.step + 1 < p_i.step$
      then
        $p_i.step = p_{j+1}.step + 1$
        $p_i.next = p_{j+1}$
      $j \leftarrow j+1$
      if $j < n$ then
        compute $I_{i-j}$, update cdll, bst, lra and tra
```

The computation of potentially useful arcs and feasible velocity areas are introduced in Section 4.2.2. The solution to the inclusion problem of feasible velocity area are described in Section 4.2.3. Complexity improvement is an integration of these major components. The steps of graph construction with complexity improvement to solve the min-# VVPTS problem of an input trajectory with n points are summarized as follows.

- **Step 1** Initiate the graph. Each point has a step number which signifies the minimum steps it takes to traverse to the last point of the trajectory. Each point also has a pointer pointing to its next point, which is used for linkage construction.
- **Step 2** Compute the potentially useful arcs in Table 3. (see Section 4.2.2)
- **Step 3** For each $i \in [1, n-1)$, compute I_{i-j} for $j \in (i, n)$ by updating the circular doubly linked list (*cdll*) and list reference array (*lra*) (see Section 4.2.2) to obtain the reference point R_i (see Section 4.2.3).

- **Step 4** For each $i \in [1, n-1)$, for $j \in (i, n)$, compute I_{i-j} while maintaining the circular doubly linked list (*cdll*), binary search tree (*bst*), list reference array (*lra*) and tree reference array (*tra*), check whether $V_{i-(j+1)}$ is interior to I_{i-j} by binary search (see Section 4.2.3) and construct a corresponding link if $V_{i-(j+1)}$ is interior to I_{i-j} and $p_{j+1}.step+1 < p_i.step$, then set $p_i.step$ to $p_{j+1}.step+1$ and set $p_i.next$ to p_{j+1}.

Complexity Analysis. The pseudo-code is given in Algorithm 1. Step 1 takes $O(1)$ time, Step 2 and 3 take $O(n^2)$ time and Step 4 takes $O(n^2 \log n)$ time. Thus the graph construct takes $O(n^2 \log n)$ time in total. Solving the min-# VVPTS problem by the graph-based approach with complexity improvement takes $O(n^2 \log n)$ time. The space complexity is $O(n)$ to maintain a column of the arc table and a row of the feasible velocity areas.

5. FINDING APPROXIMATE SOLUTION

The costs of the trajectory simplification algorithm can be reduced if the output is not required to be optimal. In this section, a heuristic algorithm which has linear time and space complexities and produces an output whose size has an upper bound will be proposed. Our heuristic algorithm is based on the following observation of the relationship between the velocities of segments on the original trajectory and the velocities of segments on a simplification of the original trajectory. The proof of Lemma 7 could be found in Section C in the appendix.

LEMMA 7. *For the sub-trajectory $T[i, j]$ ($1 \le i < j \le n$) of $T = (p_1, p_2, \ldots, p_n)$, let $\Delta V_x(T[i, j])$ and $\Delta V_y(T[i, j])$ be the range of the x and y component of velocities of each segment from p_i to p_j. i.e., $\Delta V_x(T[i, j]) = \max_{i \le k<j} V_x(p_k p_{k+1}) - \min_{i \le k<j} V_x(p_k p_{k+1})$ and $\Delta V_y(T[i, j]) = \max_{i \le k<j} V_y(p_k p_{k+1}) - \min_{1 \le k<j} V_y(p_k p_{k+1})$. If $\Delta V_x^2(T[i, j]) + \Delta V_y^2(T[i, j]) \le \epsilon_t^2$, then $\epsilon_v(p_i p_j) \le \epsilon_t$.* □

Our heuristic algorithm is designed as follows. For input trajectory $T = (p_1, p_2, \ldots, p_n)$ and error tolerance ϵ_t, define T_s to be the variable storing the approximate solution of min-# VVPTS and initialize T_s to be (p_1). Initialize the starting point and temporary ending point of the first segment in T_s to be p_1 and p_2. For current starting point p_i and temporary ending point p_j ($1 \le i < j \le n$)
(1) If $j = n$, finalize the ending point of the current segment to p_j, append p_j to T_s and return T_s.
(2) If $j < n$, if $\Delta V_x^2(T[i, j+1]) + \Delta V_y^2(T[i, j+1]) \le \epsilon_t^2$, set the temporary ending point to p_{j+1}; if $\Delta V_x^2(T[i, j+1]) + \Delta V_y^2(T[i, j+1]) > \epsilon_t^2$, finalize the ending point of the current segment to p_j, append p_j to T_s and set the starting point and temporary ending point to be p_j and p_{j+1}, respectively. Repeat this iterative process.
The pseudo-code is shown in Algorithm 2. By Lemma 7, $\epsilon_v(T_s) \le \epsilon_t$ and thus T_s is a ϵ_t-simplification of the input trajectory.
Complexity Analysis. The initialization steps of the algorithm (i.e., the first 4 lines) take constant time. In the while-loop, both condition (1) (line 6 to 9) and condition (2) (line 10 to 21) take constant time. Condition (1) will be executed once and condition (2) will be executed $n-2$ times. Therefore, the time complexity of Algorithm 2 is $O(n)$. The space complexity is also $O(n)$.

The output of this heuristic algorithm has an upper bound on its size which can be expressed by the size of the optimal solution to min-# VVPTS problem with a smaller error tolerance. The proof of Lemma 8 could be found in Section D in the appendix.

LEMMA 8. *Let T_s be the output of the new heuristic algorithm when the error tolerance is ϵ_t and let T_r be the output of the optimal algorithm for min-# VVPTS when the error tolerance is $\frac{\sqrt{2}}{4}\epsilon_t$. $|T_s| \le |T_r|$.* □

Algorithm 2 Heuristic algorithm for min-# VVPTS

Input: Trajectory $T = (p_1, p_2, \ldots, p_n)$ and error tolerance ϵ_t
Output: The ϵ_t-simplification of T

```
1: // initialization
2: $T_s \leftarrow (p_1)$; $i \leftarrow 1$; $j \leftarrow 2$
3: $\max V_x(T[i,j]), \min V_x(T[i,j]) \leftarrow V_x(p_i p_j)$
4: $\max V_y(T[i,j]), \min V_y(T[i,j]) \leftarrow V_y(p_i p_j)$
5: while true do
6:    // condition (1)
7:    if j=n then
8:       append $p_j$ to $T_s$
9:       return $T_s$
10:   // condition (2)
11:   $\max V_x(T[i,j+1]) \leftarrow \max(\max V_x(T[i,j]), V_x(p_j p_{j+1}))$
12:   $\min V_x(T[i,j+1]) \leftarrow \min(\min V_x(T[i,j]), V_x(p_j p_{j+1}))$
13:   $\max V_y(T[i,j+1]) \leftarrow \max(\max V_y(T[i,j]), V_y(p_j p_{j+1}))$
14:   $\min V_y(T[i,j+1]) \leftarrow \min(\min V_y(T[i,j]), V_y(p_j p_{j+1}))$
15:   if $\Delta V_x^2(T[i,j+1]) + \Delta V_y^2(T[i,j+1]) \le \epsilon_t^2$ then
16:      $j \leftarrow j+1$
17:   else
18:      append $p_j$ to $T_s$
19:      $i \leftarrow j$; $j \leftarrow j+1$
20:      $\max V_x(T[i,j]), \min V_x(T[i,j]) \leftarrow V_x(p_i p_j)$
21:      $\max V_y(T[i,j]), \min V_y(T[i,j]) \leftarrow V_y(p_i p_j)$
```

6. EMPIRICAL STUDIES

We use two datasets, namely GeoLife [18, 16, 17] and T-Drive [14, 15]. GeoLife dataset stores the outdoor movements collected by 182 people by GPS loggers and GPS phones in over 5 years. T-Drive dataset stores the movements of 10357 taxis in one week. The statistics of the two datasets are summarized in Table 4. In the whole Section 6, the unit of E_v is $10^{-4} \times degree/second$, where $degree$ is the degree of latitude and longitude.

We study two optimal algorithms, namely direct implementation of graph-based approach and graph-based approach with complexity improvement (VVPTS). We also study three approximate algorithms, including popular greedy, split algorithms and the newly proposed heuristic algorithm. All algorithms are implemented in C++ and the experiments are run on a Linux platform with a 2.66GHz machine and 48GB RAM.

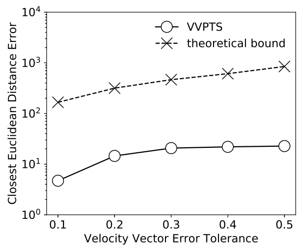
	# of trajectories	total # of points	average # of positions per trajectory
GeoLife	17,621	24,876,978	1,412
T-Drive	10,359	17,740,902	1,713

Table 4: Real datasets

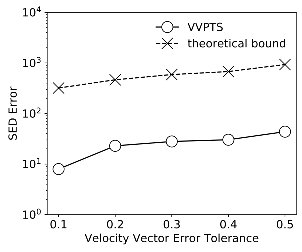
Due to page limit, the experimental results on T-Drive can be found in the full version of this paper [12]. The experimental results on T-Drive are similar as GeoLife. Sample running code can be found at *http://github.com/zhmeishi/VVPTS/* .

6.1 Error Measurements Comparison

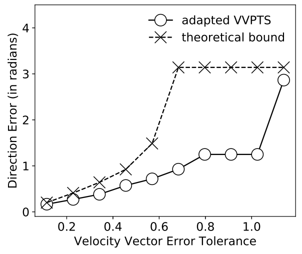
In this part, for a simplified trajectory whose error under E_v is bounded, we study its error under other common error metrics.
Closest Euclidean Distance Error Measurement. The theoretical bound (Lemma 1), which is $\max_{1 \le k<m}[dist(p_{s_k}, p_{s_{k+1}})/2 + \epsilon_t \Delta T(p_{s_k} p_{s_{k+1}})]$ for T_s whose size is m, and the actual error are computed. We vary the tolerance ϵ_t on $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. The results are shown in Figure 8(a). We observe that the empirical closest Euclidean distance error is significantly smaller than the theoretical bound and becomes stable when ϵ_t is large.



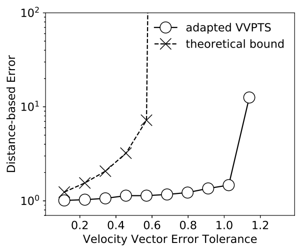
(a) CD error



(b) SED error



(c) Direction-based error



(d) Distance-based error

Figure 8: Verification of theoretical error bounds (GeoLife)

SED Error Measurement. The theoretical bound (Lemma 2), which is $\max_{1 \le k < m}[dist(p_{s_k}, p_{s_{k+1}}) + \epsilon_t \Delta T(p_{s_k} p_{s_{k+1}})/2]$ for T_s whose size is m, and the actual error are computed. We vary the tolerance ϵ_t on $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. The results are shown in Figure 8(b). Similar with the closest Euclidean distance error, the empirical SED error is also significantly smaller than the theoretical bound.

Adaptive VVPTS. The theoretical bound of the direction-based error and the distance-based error are related to the smallest speed of segments of the trajectory. However, it is quite often that the smallest speed becomes zero (two adjacent points coincide with each other) and the upper-bounds do not exist. Thus, we adapt the optimal VVPTS algorithm to preserve those adjacent points which are too close to each other. The adaptation is operated as follows.

(1) For a burst of consecutive static segments (speed is zero), only keep the start point and the end point of this sub-trajectory.
(2) For all segments with a smaller speed than the given velocity-vector-based error but larger than zero, keep those segments.
(3) Split the trajectory by the end points of the kept segments mentioned before and simplify each sub-trajectory by VVPTS.
(4) Merge the simplified sub-trajectories and kept segments to be the final simplified trajectory.

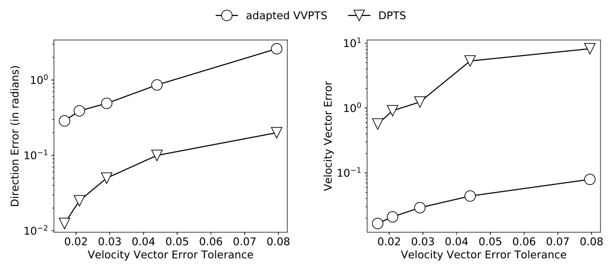
Direction-based Error Measurement. We first compute the smallest speed of segments on T. Then, ϵ_t is set to be smaller than the smallest speed. The theoretical bound (see Lemma 3), which is $\arcsin(\epsilon_t / \min_{1 \le i < n} \|\vec{V}(p_i p_{i+1})\|)$, and the actual error are computed. ϵ_t is varied from small to large up to the value of the smallest speed of segments on T. After that, ϵ_t is set to be greater than the smallest speed. With increasing ϵ_t, the theoretical bound becomes π (unbounded) and the corresponding direction error of T_s is computed. The results are shown in Figure 8(c). We can see when the theoretical bound becomes π, the direction error is still small when ϵ_t is not too large.

Distance-based Error Measurement. After the greatest speed of segments on T is computed, ϵ_t is first set to be smaller than the greatest speed. The theoretical bound (see Lemma 4), which is $1/(1 - \epsilon_t / \min_{1 \le k < m} \max_{s_k \le i < s_{k+1}} \|\vec{V}(p_i p_{i+1})\|)$, and the actual error are computed. ϵ_t is varied from small to large up to the value of the greatest speed of segments on T. Then, ϵ_t is set to be greater than the greatest speed. With increasing ϵ_t, the theoretical bound becomes $+\infty$ and the corresponding distance error of T_s is computed. The results are shown in Figure 8(d). Similarly, we can see even if the theoretical bound becomes $+\infty$ (unbounded), the distance-based error is still in a small scale until ϵ_t is larger than 1.2.

6.2 VVPTS vs. DPTS

In this section, we compare the adapted VVPTS introduced in Section 6.1 with DPTS. In Section 6.1, we show that the direction error of VVPTS is bounded when ϵ_t is smaller than the smallest speed. Following the comparison method used by [8], we do the comparison in terms of the direction error and the velocity vector error, and enforce that the simplified trajectories from the adapted VVPTS and DPTS have the same size.

We vary ϵ_t for VVPTS and the results are shown in Figure 9. From Figure 9(a), we can see that the ratio of direction errors is between 8.6 and 23.0. However, the ratio of velocity vector errors is between 34.2 and 121.1 as shown in Figure 9(b), which is significantly larger than that of VVPTS.



(a) Direction error (b) Velocity vector error

Figure 9: Comparison with existing DPTS (GeoLife)

6.3 Performance of Algorithms

In this part, time cost, space cost and size ratio of algorithms with different input trajectory size and error tolerance are examined. For the newly proposed heuristic algorithm, the theoretical bound of its output (see Lemma 8) is verified.

Effect of $|T|$. Sizes of input trajectories are set to 2,000, 4,000, 6,000, 8,000 and 10,000 for optimal algorithms and are set to 20,000, 40,000, 60,000, 80,000 and 100,000 for approximate algorithms. ϵ_t is fixed to 0.5. First, two optimal min-# VVPTS algorithms, namely the graph-based approach and the approach with complexity improvement (VVPTS) are run. The results are shown in Figure 10. The VVPTS spends less time and less memory than the graph-based approach. Second, we test three approximate algorithms, namely greedy, split, and the newly proposed heuristic algorithm. The results are shown in Figure 11. From the experiment results, the new heuristic algorithm is the fastest. Split algorithm is faster than the greedy algorithm. The greedy algorithm takes much more time than the other two approximate algorithms. The three approximate algorithms have the same linear space cost.

Effect of ϵ_t. We vary the tolerance ϵ_t on $\{0.1, 0.2, 0.4, 0.8, 1.6\}$ and fix the size of input trajectory to be 50,000. The results are shown in Figure 12(a) and Figure 12(b). ϵ_t affects the graph-based approach and the greedy approach greatly since a larger ϵ_t would make it more likely that a long sequence of consecutive segments could be approximated with one segment, which makes the cost of checking the error of that long sequence larger. For the split, the time cost decreases as ϵ_t increases, because the recursion depth is less when ϵ_t becomes larger.

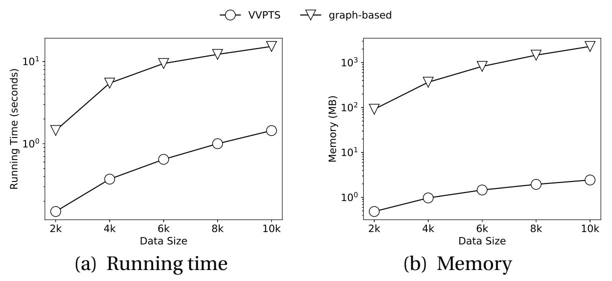

(a) Running time (b) Memory

Figure 10: Effect of data size $|T|$ on optimal algs (GeoLife)

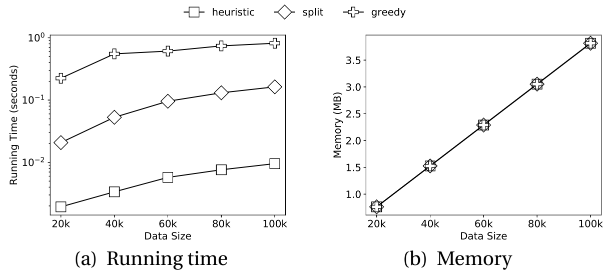

(a) Running time (b) Memory

Figure 11: Effect of data size $|T|$ on approx. algs (GeoLife)

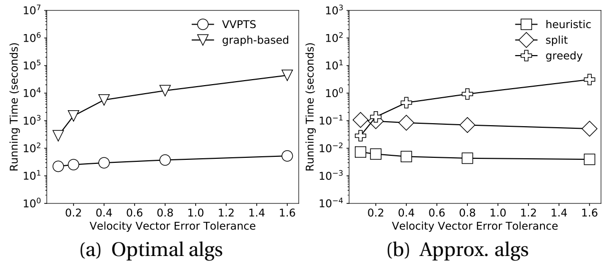

(a) Optimal algs (b) Approx. algs

Figure 12: Effect of error tolerance ϵ_t on algs (GeoLife)

Compression Rate. The *size ratio*, which is defined to be $|T_s|/|T|$ where T_s is the simplification of T by different algorithms are compared. The results are shown in Figure 13(a). The size ratio decreases with increasing error tolerance. The size ratios of approximate algorithms are larger than the optimal algorithms. The theoretical bound (see Lemma 8) and actual value of the size of simplified trajectory by the new heuristic algorithm are also verified in Figure 13(a).

Approximation Error. We define T' to be the simplified trajectory returned by the approximate algorithm on a given raw trajectory and T^* to be the simplified trajectory returned by an optimal algorithm on the same raw trajectory. The approximation error of an approximate algorithm is defined to be $|T'/T^*|$. The results are shown in Figure 13(b). The heuristic algorithm has a smaller approximation error than the split algorithm. The greedy algorithm has the smallest approximation error. This is because for a trajectory without frequent abrupt velocity variations, local optimal solutions would be close to the global optimal solution.

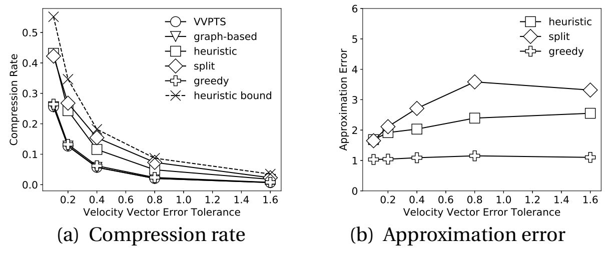

(a) Compression rate (b) Approximation error

Figure 13: Compression rate and approximation errors (GeoLife)

Scalability Test. For the direct implementation of the graph-based optimal approach, when the size of input is set to be 50,000, the space cost is already 56GB, which is larger than 48GB RAM, and thus it's not scalable. To study the scalability of VVPTS, sizes of input trajectories are increased to around 200,000, 400,000, 600,000, 800,000 and 1,000,000. The time complexities ($O(n^2 \log n)$) and space complexities ($O(n)$) of VVPTS are verified and the results are shown in Figure 14. To study the scalability of the approximate algorithms, sizes of the input trajectories are increased to around 2,000,000, 4,000,000, 6,000,000, 8,000,000 and 10,000,000. The time complexities ($O(n^2)$ for greedy, $O(n^2)$ for split, and $O(n)$ for the new heuristic algorithm) are verified and the results are shown in Figure 15. The new heuristic algorithm is the fastest among three approximate algorithms.

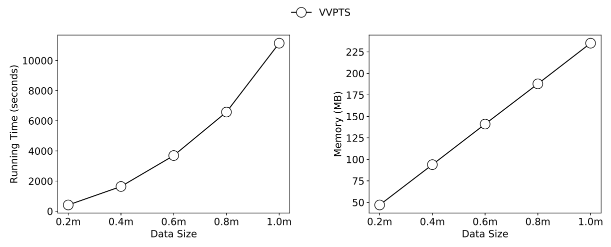

(a) Running time (b) Memory

Figure 14: Scalability test on VVPTS (GeoLife)

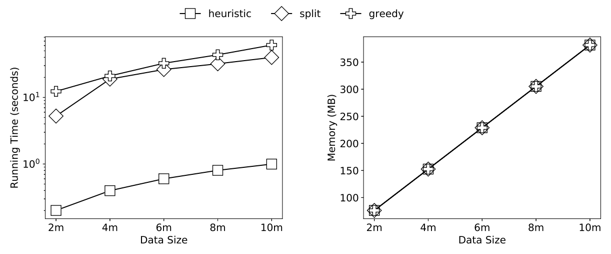

(a) Running time (b) Memory

Figure 15: Scalability test on approx. algs (GeoLife)

7. CONCLUSION

This paper proposes the idea of simplifying a trajectory while preserving its velocity information. A velocity-vector-based trajectory simplification error measurement E_v is defined and its relationships with some of the existing error measurements are studied. We show that the optimal solution to the min-# VVPTS problem can be obtained using $O(n^3)$ time and $O(n^2)$ space by a direct adaption of graph-based approach. We develop an algorithm to reduce the time cost to $O(n^2 \log n)$ and the space cost to $O(n)$ and thoroughly describe its data structure (i.e., a doubly linked list with its reference array and a binary search tree with its reference array) and corresponding operations. We present an algorithm with linear time and space costs which produces an approximate solution to the min-# VVPTS problem and analytically prove that the size of the approximate solution has a certain guarantee. We conduct experiments on real datasets (i.e., Geolife and T-Drive) to empirically prove the measurements relationships and the theoretical bound of the compression rate for the approximate algorithm. We also compared our adapted VVPTS algorithm with DPTS in the experiments and test the efficiency and the scalability of our two proposed methods.

8. REFERENCES

[1] P. Bose, S. Cabello, O. Cheong, J. Gudmundsson, M. van Kreveld, and B. Speckmann. Area-preserving approximations of polygonal paths. *Journal of Discrete Algorithms*, 4(4):554–566, 2006.

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

[3] 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.

[4] J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. *Computational geometry*, 42(9):825–841, 2009.

[5] J. Gudmundsson, G. Narasimhan, and M. Smid. Distance-preserving approximations of polygonal paths. *Computational Geometry*, 36(3):183–196, 2007.

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

[7] C.-Y. Lin, C.-C. Hung, and P.-R. Lei. A velocity-preserving trajectory simplification approach. In *Technologies and Applications of Artificial Intelligence (TAAI), 2016 Conference on*, pages 58–65. IEEE, 2016.

[8] C. Long, R. C.-W. Wong, and H. Jagadish. Direction-preserving trajectory simplification. *Proceedings of the VLDB Endowment*, 6(10):949–960, 2013.

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

[10] J. Muckell, P. W. Olsen Jr, J.-H. Hwang, C. T. Lawson, and S. Ravi. Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica*, 18(3):435–460, 2014.

[11] M. I. Shamos. Geometric complexity. In *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 224–233. ACM, 1975.

[12] G. Wang, Z. Shi, C. Long, Y. Gao, and R. C.-W. Wong. Velocity vector preserving trajectory simplification (technical report). In *http://zhmeishi.github.io/paper/VVPTS.pdf*, 2019.

[13] J. J.-C. Ying and J.-H. Su. On velocity-preserving trajectory simplification. In *Asian Conference on Intelligent Information and Database Systems*, pages 241–250. Springer, 2016.

[14] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM, 2011.

[15] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108. ACM, 2010.

[16] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on gps data. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 312–321. ACM, 2008.

[17] Y. Zheng, X. Xie, and W.-Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

[18] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th international conference on World wide web*, pages 791–800. ACM, 2009.

APPENDIX

A. PROOF OF LEMMA 1

PROOF. Let $\epsilon_{cd}(p_{s_k}p_{s_{k+1}}) = dist(p_i, p'_i)$ $(i \in [s_k, s_{k+1}])$ where p'_i is the position on $p_{s_k}p_{s_{k+1}}$ which has the smallest Euclidean distance from p_i. $p_ip'_i$ is perpendicular to $p_{s_k}p_{s_{k+1}}$.

$$dist^2(p_i, p'_i) + dist^2(p_{s_k}, p'_i) = dist^2(p_{s_k}, p_i)$$
$$dist^2(p_i, p'_i) + dist^2(p'_i, p_{s_{k+1}}) = dist^2(p_i, p_{s_{k+1}}), thus$$
$$2dist^2(p_i, p'_i) = dist^2(p_{s_k}, p_i) + dist^2(p_i, p_{s_{k+1}}) - [dist^2(p_{s_k}, p'_i)+ dist^2(p'_i, p_{s_{k+1}})]$$
$$2dist^2(p_i, p'_i) \le [\textstyle\sum_{h=s_k}^{s_{k+1}-1} dist(p_h, p_{h+1})]^2 - dist^2(p_{s_k}, p_{s_{k+1}})/2$$
$$dist^2(p_i, p'_i) \le [\textstyle\sum_{h=s_k}^{s_{k+1}-1} (\|\vec{V}(p_hp_{h+1})\|\Delta T(p_hp_{h+1}))]^2/2 - dist^2(p_{s_k}, p_{s_{k+1}})/4$$
$$Since \quad \epsilon_v(p_{s_k}p_{s_{k+1}}) \le \epsilon_t, \|\vec{V}(p_{s_k}p_{s_{k+1}})\| \ge \|\vec{V}(p_hp_{h+1})\| - \epsilon_t$$
$$dist^2(p_i, p'_i) \le [(\|\vec{V}(p_{s_k}p_{s_{k+1}})\| + \epsilon_t)\Delta T(p_{s_k}p_{s_{k+1}})]^2/2 - dist^2(p_{s_k}, p_{s_{k+1}})/4$$
$$dist^2(p_i, p'_i) \le dist^2(p_{s_k}, p_{s_{k+1}})/4 + \epsilon_t^2\Delta T^2(p_{s_k}p_{s_{k+1}})/2+ dist(p_{s_k}, p_{s_{k+1}})\epsilon_t\Delta T(p_{s_k}p_{s_{k+1}})$$
$$dist^2(p_i, p'_i) \le [dist(p_{s_k}, p_{s_{k+1}})/2 + \epsilon_t\Delta T(p_{s_k}p_{s_{k+1}})]^2 - \epsilon_t^2\Delta T^2(p_{s_k}p_{s_{k+1}})/2$$
$$Therefore \quad dist(p_i, p'_i) \le dist(p_{s_k}, p_{s_{k+1}})/2 + \epsilon_t\Delta T(p_{s_k}p_{s_{k+1}})$$

We complete the proof. □

B. PROOF OF LEMMA 2

PROOF. Let $\epsilon_{sed}(p_{s_k}p_{s_{k+1}}) = dist(p_i, p'_i)$ $(i \in [s_k, s_{k+1}])$ where p'_i is the temporally synchronized position of p_i on T_s.

$$dist(p_i, p'_i) < \textstyle\sum_{h=s_k}^{i-1} dist(p_h, p_{h+1}) + dist(p_{s_k}, p'_i)$$
$$dist(p_i, p'_i) < \textstyle\sum_{h=i}^{s_{k+1}-1} dist(p_h, p_{h+1}) + dist(p'_i, p_{s_{k+1}})$$
$$2dist(p_i, p'_i) < \textstyle\sum_{h=s_k}^{s_{k+1}-1} dist(p_h, p_{h+1}) + dist(p_{s_k}, p_{s_{k+1}})$$
$$2dist(p_i, p'_i) < [\|\vec{V}(p_{s_k}p_{s_{k+1}})\| + \epsilon_t]\Delta T(p_{s_k}p_{s_{k+1}}) + dist(p_{s_k}, p_{s_{k+1}})$$
$$Therefore \quad dist(p_i, p'_i) < dist(p_{s_k}, p_{s_{k+1}}) + \epsilon_t\Delta T(p_{s_k}p_{s_{k+1}})/2$$

We complete the proof. □

C. PROOF OF LEMMA 7

PROOF. Assume that $\Delta V_x^2(T[i,j]) + \Delta V_y^2(T[i,j]) \le \epsilon_t^2$. Let $\max V_x(T[i,j]) = \max_{i\le k<j} V_x(p_kp_{k+1})$, $\min V_x(T[i,j]) = \min_{i\le k<j} V_x(p_kp_{k+1})$, $\max V_y(T[i,j]) = \max_{i\le k<j} V_y(p_kp_{k+1})$ and $\min V_y(T[i,j]) = \min_{i\le k<j} V_y(p_kp_{k+1})$. Since $V_x(p_ip_j) = [\sum_{i\le k<j} V_x(p_kp_{k+1})\Delta t(p_kp_{k+1})]/\Delta t(p_ip_j)$, $\min V_x(T[i,j]) \le V_x(p_ip_j) \le \max V_x(T[i,j])$. Similarly, $\min V_y(T[i,j]) \le V_y(p_ip_j) \le \max V_y(T[i,j])$. Therefore, V_{i-j} and all $V_{k-(k+1)}$ for $k \in [i,j)$ fall inside the rectangle defined by vertices (in anticlockwise order) $(\min V_x(T[i,j]), \min V_y(T[i,j]))$, $(\max V_x(T[i,j]), \min V_y(T[i,j]))$, $(\max V_x(T[i,j]), \max V_y(T[i,j]))$, $(\min V_x(T[i,j]), \max V_y(T[i,j]))$. The greatest distance between any two points inside the rectangle is the length of the rectangle's diagonal d_{max}, where $d_{max}^2 = (\max V_x(T[i,j]) - \min V_x(T[i,j]))^2 + (\max V_y(T[i,j]) - \min V_y(T[i,j]))^2 \le \epsilon_t^2$. Thus $\epsilon_v(p_ip_j) = \max_{i\le k<j} dist(V_{i-j}, V_{k-(k+1)}) \le d_{max} \le \epsilon_t$. □

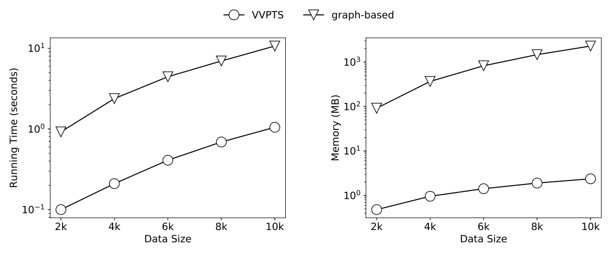
D. PROOF OF LEMMA 8

PROOF. Let $T_s = (p_{s_1}, p_{s_2}, \ldots, p_{s_m})$ and let $T_r = (p_{r_1}, p_{r_2}, \ldots, p_{r_l})$. Our aim is to prove that $m \le l$. We assume $m > l$. The proof has two major steps. First, we prove that for $k \in [1, l)$, $\Delta V_x^2(T[r_k, r_{k+1}]) + \Delta V_y^2(T[r_k, r_{k+1}]) \le \epsilon_t^2$. According to the definition, $\epsilon_v(p_{r_k} p_{r_{k+1}}) \le \frac{\sqrt{2}}{4}\epsilon_t$. $V_{r_k - r_{k+1}}$ falls inside $I_{r_k-(r_{k+1}-1)}$ when the error tolerance is $\frac{\sqrt{2}}{4}\epsilon_t$. $I_{r_k-(r_{k+1}-1)}$, which is the common intersection of D_i ($i \in [r_k, r_{k+1})$) whose center is $V_{i-(i+1)}$ and radius is $\frac{\sqrt{2}}{4}\epsilon_t$, should be non-empty. $\max V_x(T[r_k, r_{k+1}]) - \frac{\sqrt{2}}{4}\epsilon_t \le \min V_x(T[r_k, r_{k+1}]) + \frac{\sqrt{2}}{4}\epsilon_t$ and $\max V_y(T[r_k, r_{k+1}]) - \frac{\sqrt{2}}{4}\epsilon_t \le \min V_y(T[r_k, r_{k+1}]) + \frac{\sqrt{2}}{4}\epsilon_t$. It can be derived that $\Delta V_x^2(T[r_k, r_{k+1}]) + \Delta V_y^2(T[r_k, r_{k+1}]) \le \epsilon_t^2$. Second, we prove that $r_k \le s_k$ ($k \in [1, l)$) by induction. According to the definition, $r_1 = s_1 = 1$. According to Algorithm 2, for s_i ($1 \le i < m$), s_{i+1} is the greatest index j such that $\Delta V_x^2(T[s_i, j]) + \Delta V_y^2(T[s_i, j]) \le \epsilon_t^2$. As is proved in the first step, $\Delta V_x^2(T[r_1, r_2]) + \Delta V_y^2(T[r_1, r_2]) \le \epsilon_t^2$. Therefore, $r_2 \le s_2$. For $k = 3$, there are two possible cases: (1) If $r_3 \le s_2$, $r_3 < s_3$. (2) If $r_3 > s_2$, as is proved in the first step, $\Delta V_x^2(T[r_2, r_3]) + \Delta V_y^2(T[r_2, r_3]) \le \epsilon_t^2$. Since $s_2 \in [r_2, r_3)$, $\Delta V_x^2(T[s_2, r_3]) + \Delta V_y^2(T[s_2, r_3]) \le \epsilon_t^2$. Therefore $r_3 \le s_3$. By induction, for $1 \le k \le l$, $r_k \le s_k$. Since $n = r_l \le s_l$ and $m > l$ (assumption), $s_m > n$, which contradicts the definition that $s_m = n$. Thus, the assumption is wrong, indicating that $m \le l$. □

E. EXPERIMENTAL RESULTS ON THE T-DRIVE DATASETS

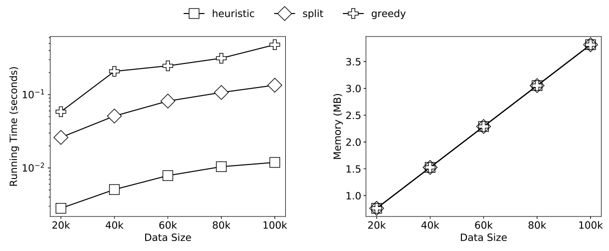
We provide the remaining experimental results of the performance study on the optimal algorithms and approximate algorithms on the T-Drive datasets in this part.

Effect of $|T|$. Sizes of input trajectories are set to 2,000, 4,000, 6,000, 8,000 and 10,000 for optimal algorithms and are set to 20,000, 40,000, 60,000, 80,000 and 100,000 for approximate algorithms. ϵ_t is fixed to 0.5. The results are shown in Figure 16 and Figure 17, which are similar to those on GeoLife.



(a) Running time (b) Memory

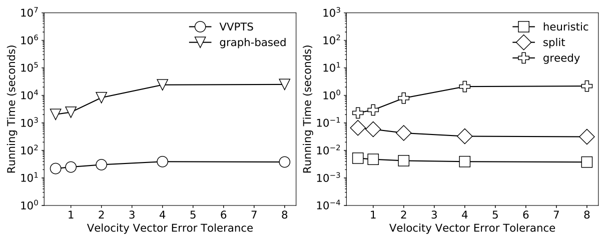
Figure 16: Effect of data size $|T|$ on optimal algs (T-Drive)



(a) Running time (b) Memory

Figure 17: Effect of data size $|T|$ on approx. algs (T-Drive)

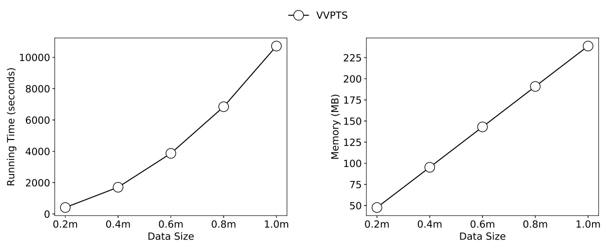
Effect of ϵ_t. We vary the tolerance ϵ_t on {0.5, 1, 2, 4, 8} and fix the size of input trajectory to be 50,000. The results are shown in Figure 18(a) and Figure 18(b), which are similar to those on GeoLife.



(a) Optimal algs (b) Approx. algs

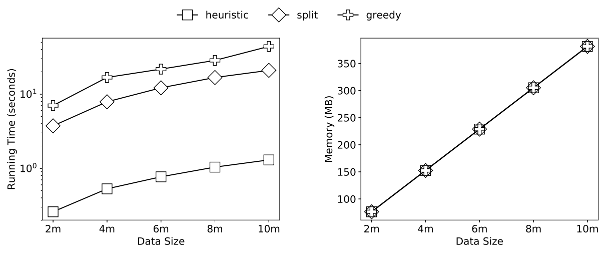
Figure 18: Effect of error tolerance ϵ_t on algs (T-Drive)

Scalability Test. The results are shown in Figure 20. Again, the results are similar to those on GeoLife.



(a) Running time (b) Memory

Figure 19: Scalability test on VVPTS (T-Drive)



(a) Running time (b) Memory

Figure 20: Scalability test on approx. algs (T-Drive)