An efficient online direction-preserving compression approach for trajectory streaming data

Ze Deng [a,b], Wei Han [a,b], Lizhe Wang [a,b,*], Rajiv Ranjan [c], Albert Y. Zomaya [d], Wei Jie [e]

[a] School of Computer Science, China University of Geosciences, Wuhan, 430074, China
[b] Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China
[c] Newcastle University, UK
[d] The Sydney University, Australia
[e] The West London University, UK

HIGHLIGHTS

- An online compression method for trajectory streaming data is developed to preserve error-bound direction information.
- An advanced online DPTS algorithm with a BQS structure is proposed which can significantly reduce the compression time.
- A parallel method of the online DPTS+ on a GPU platform is implemented, which further improved the time efficiency.

ABSTRACT

Online trajectory compression is an important method of efficiently managing massive volumes of trajec-tory streaming data. Current online trajectory methods generally do not preserve direction information and lack high computing performance for the fast compression. Aiming to solve these problems, this paper first proposed an online direction-preserving simplification method for trajectory streaming data, online DPTS by modifying an offline direction-preserving trajectory simplification (DPTS) method. We further proposed an optimized version of online DPTS called online DPTS$^+$ by employing a data structure called bound quadrant system (BQS) to reduce the compression time of online DPTS. To provide a more efficient solution to reduce compression time, this paper explored the feasibility of using contemporary general-purpose computing on a graphics processing unit (GPU). The GPU-aided approach paralleled the major computing part of online DPTS$^+$ that is the SP-theo algorithm. The results show that by maintaining a comparable compression error and compression rate, (1) the online DPTS outperform offline DPTS with up to 21% compression time, (2) the compression time of online DPTS$^+$ algorithm is 3.95 times faster than that of online DPTS, and (3) the GPU-aided method can significantly reduce the time for graph con-struction and for finding the shortest path with a speedup of 31.4 and 7.88 (on average), respectively. The current approach provides a new tool for fast online trajectory streaming data compression.

1. Introduction

Recent advances in sensing, networking, smart grid [1], smart home [2], and location acquisition technologies have led to a huge volume of trajectory streaming data (e.g., Global Positioning System (GPS) trajectories). There are three main challenges when there is such a huge volume of data: (1) storing the sheer volume of trajectory data may overwhelm available storage space, (2) the cost of transmitting a large amount of trajectory data over cellular or satellite networks can be expensive, and the large size of trajectory data makes it very difficult to discover useful patterns. Trajectory compression technologies can provide a solution for these challenges [3].

Trajectory data compression approaches are generally divided into two categories: offline or online compression [4]. The offline methods (e.g., Douglas–Peucker [5] and TD–TR [6]) discard some locations with negligible errors from an original trajectory,

* Corresponding author at: School of Computer Science, China University of Geosciences, Wuhan, 430074, China.
E-mail address: lizhe.wang@gmail.com (L. Wang).

which is already obtained before the compression process [7]. However, in many applications the trajectory data of the moving objects arrive in a stream. These applications include real-time trajectory tracking [8] and long-term location tracking [9]. Therefore, some online compression approaches have been proposed to deal with this case. The basic idea is to use segment heuristic for trajectory and remove some points of the recently received trajectories. Representative methods include the opening window algorithm [10], dead reckoning [11], and SQUISH-E(λ) [12]. However, existing online compression approaches have some drawbacks. First, according to [13], nearly all trajectory compression approaches are position-preserving trajectory simplification (PPTS) methods. These methods lose direction information so that many applications based on location-based service (LBS) cannot be broadly supported. Although two direction-preserving trajectory simplification (DPTS) methods [13,14] have been proposed to solve this issue, these two methods are designed for offline compression. To the best of our knowledge, no online DPTS method has been proposed to date. Second, current online approaches have individual drawbacks in terms of either time costs, compression ratio, and error boundaries. For example, the opening window algorithm suffers from $O(n^2)$ time complexity [12] and dead reckoning suffers from a low compression ratio. Meanwhile, the high efficiency of compression is a key requirement for current online trajectory compression methods because the volume and density of streaming data have been rapidly growing. Therefore, there is a need for an efficient direction-preserving compression approach for trajectory streaming data.

To address these research challenges, we propose an online direction-preserving trajectory compression method for trajectory streaming data that modifies an offline DPTS method [15], after which a data structure called a bounded quadrant system (BQS) [9,16] is used to optimize our compression method. Furthermore, a modern GPU platform is used to improve the performance of our trajectory compression method.

The main contributions of this study are as follows:

1. We have developed an online trajectory compression method for trajectory streaming data called online DPTS that preserves error-bound direction information and has a high compression ratio.
2. We have introduced an advanced online DPTS algorithm called online DPTS$^+$ with a BQS structure in [9,16] can significantly reduce the compression time of online DPTS.
3. We designed a parallel method of the online DPTS$^+$ on a GPU platform, which further improved the time efficiency of trajectory streaming data compression. The GPU-aided method accelerate the SP-theo algorithm in the online DPTS$^+$, with two well-designed GPU parallel schemes.
4. We performed extensive experiments to evaluate the proposed methods using real trajectory datasets.

To the best of our knowledge, the proposed compression method is the first online trajectory compression method that takes both direction preserving and parallel processing into consideration.

The remainder of this paper is organized as follows: Section 2 discusses work relating to online trajectory compression. Section 3 introduces our online trajectory compression approaches (i.e., online DPTS and online DPTS$^+$). Section 4 describes the GPU-aided compression approach on a modern GPU platform. Section 5 presents the experiments and performance evaluation results of the proposed approaches. Section 6 concludes with a summary and a plan for future work.

2. Related work

A number of successful attempts have been made regarding online trajectory compression. The most salient works are described.

Opening window (OPW) is a kind of traditional online trajectory compression algorithm. Such algorithms, including NOWA and BOPW [10], slide a window over the points on the original trajectory to approximate each trajectory using the number of points in the window so that the resulting spatial error is smaller than a bound. This process is repeated until the last point of the original trajectory is processed. The worst-case time complexity of OPW is $O(n^2)$. Opening window time ratio (OPW-TR) [10] extended OPW using a synchronized Euclidean distance (SED) error instead of spatial error.

Some fast online trajectory compression algorithms have been proposed to overcome the high time overheads of OPW and OPW-TR. These include uniform sampling [17] and dead reckoning [11]. The uniform sampling method carefully selects a few points to store and discards the remaining points at every given time interval or distance interval. Dead reckoning stores the location of the first point and the velocity at this point. It then skips every subsequent point whose location can be estimated from the information about the first point within the given SED value until it finds one point whose location cannot be estimated. The location of the point and the speed at the point are stored and used to estimate the locations of following points. This process is repeated until the input trajectory ends. The computational complexity of this kind of method is $O(n)$. However, the major drawback of this kind of method is the lower compression rates compared with OPW and OPW-TR. Therefore, a few online trajectory compression methods that can ensure both a high compression ratio and low computing overheads have been presented. For example, given parameters λ and μ, SQUISH-E can ensure a compression ratio of λ while preventing the SED errors that are not beyond μ. However, this algorithm does not preserve direction information.

Significantly different from the existing online trajectory compression methods, this paper focuses on the emerging challenges of (1) the direction-preserved online trajectory compression with error boundary and high compression rate and (2) enabling a high-performance solution to maintain the computational performance of the proposed method for trajectory streaming data. The proposed compression method is the first online trajectory compression method that takes direction preserving and parallel processing into consideration.

3. Online DPTS: online direction-preserved trajectory simplification

In this section, we formulate the problem, present the details of the proposed compression algorithm, and describe the algorithm optimization.

3.1. Problem formulation

In our setting, a central server continuously collects the location points of moving objects over time. Thus, such points relating to a moving object O form a trajectory stream. Noted that, the issue of streaming inconsistency, which means the order of location points in the original stream (in input) is different from the output, may happen because the server needs to receive location information of multiple moving objects concurrently. However, in this paper, we assume that the consistency of streaming has been achieved. In the future, we will consider the issue of streaming inconsistency and attempt to employ some methods such as in [18] or [19] to reorder the input streaming data before compressing trajectories.

Definition 1 (*Location Point*)**.** A location point, denoted as $p = (x, y, t)$, is a tuple that records the latitude, longitude, and timestamp of one location sample.

Definition 2 (*Trajectory Segment*)**.** A trajectory segment, denoted as $g = \{p_1, \ldots, p_n\}$, is a set of continuous location points.

Definition 3 (*Compressed Trajectory Segment*)**.** A compressed trajectory segment, denoted as $g' = \{ps_1, ps_2, \ldots, ps_m\}$, is the simplification of $g = \{p_1, p_2, \ldots, p_n\}$ where all points from ps_1 to ps_m are consecutive and contained in g.

Definition 4 (*Trajectory Stream*)**.** A trajectory stream, denoted as $S = \{g_1, g_2, \ldots\}$, consists of an unbounded set of trajectory segments.

Definition 5 (*Compressed Trajectory Stream*)**.** For a trajectory stream $S = \{g_1, g_2, \ldots, g_k, \ldots\}$, the compressed trajectory stream is defined as an unbounded set of compressed segments $S' = \{g'_1, g'_2, \ldots, g'_k, \ldots\}$, where g'_i is the simplification of $g_i \in S$.

Definition 6 (*The Direction of Trajectory Segment Line*)**.** Given a line in a trajectory segment $g = \{p_1, \ldots, p_n\}$ that is denoted as a vector $\overrightarrow{l} = \overrightarrow{p_i, p_j}$ where $1 \le i < j \le n-1$, the direction of $\overrightarrow{l}$, denoted as $\theta(\overrightarrow{l}) = \theta(\overrightarrow{p_i, p_j})$, is defined as the angle of an anticlockwise rotation from the position of the x-axis to the vector $\overrightarrow{p_i, p_j}$.

Definition 7 (*The Angular Difference Between Two Directions*)**.** For two directions $\theta 1$ and $\theta 2$, the angular difference between $\theta 1$ and $\theta 2$, denoted by $\Delta(\theta 1, \theta 2)$, is defined as the minimum of the angle of the anticlockwise rotation from $\theta 1$ to $\theta 2$ and that from $\theta 2$ to $\theta 1$, i.e., $\Delta(\theta 1, \theta 2) = \min\{|\theta 1 - \theta 2|, 2\pi - |\theta 1 - \theta 2|\}$.

Definition 8 (*The Compression Error of a Segment Line in g'*)**.** Let g' be a compression trajectory segment for a trajectory segment g. Given a segment line $\overrightarrow{l_j} = \overrightarrow{p_j, p_{j+1}}$ in g', the compression error of $\overrightarrow{l_j}$, denoted by $\epsilon(\overrightarrow{l_j})$, is defined as the greatest angular difference between $\theta(\overrightarrow{l_j})$ and $\theta(\overrightarrow{l_k})$, where $\overrightarrow{l_k} = \overrightarrow{p_k, p_{k+1}}$ is one segment line in g approximated by $\overrightarrow{l_j}$. That means:

$$\epsilon(\overrightarrow{l_j}) = Max(\Delta(\theta(\overrightarrow{l_j}), \theta(\overrightarrow{l_k}))),$$
$$where\ (l_j.p_j.t \le l_k.p_k.t) \wedge (l_k.p_{k+1}.t \le l_j.p_{j+1}.t). \quad (1)$$

Definition 9 (*The Compression Error of S'*)**.** The compression error of compressed trajectory stream S', denoted by $\varepsilon(S')$, is defined as the maximum error of a compressed segment in S', i.e.,

$$\varepsilon(S') = Max(\epsilon(l_j)),$$
$$where\ (l_j\ is\ one\ compressed\ segment\ line\ in\ S'). \quad (2)$$

Here, the objective is to quickly compress a trajectory stream S to form the corresponding compressed trajectory stream S' at one snapshot. As a result, $\varepsilon(S')$ is bounded with one direction threshold and S' has the smallest size.

3.2. Algorithm description

We proposed an online DPTS that combines an offline DPTS approach [15] and BQS data structure for online trajectory compression [12]. First, we introduce the offline DPTS approach, convert the offline-DPTS to its online version, and optimize the online DPTS using a BQS.

In [15], the authors proposed an implementation of direction-preserving simplification called a SP algorithm for the smallest size that is error-bound under a threshold ϵ_t. The SP algorithm consists of three steps:

Step 1: (Graph Construction): Constructing a graph with an error tolerance threshold, denoted as G_{ϵ_t}, based on an offline trajectory so that the error value of each edge $\overrightarrow{p_i, p_j}$ $(i < j)$ in G_{ϵ_t}, $\epsilon(\overrightarrow{p_i, p_j}) \le \epsilon_t$.

Step 2: (Shortest Path Finding): Computing the shortest path based on the graph G_{ϵ_t} from Step 1.

Step 3: (Solution Generation): Generating the solution for direction-preserving trajectory compression using the shortest path found in Step 2.

In Step 1, one straightforward solution for constructing G_{ϵ_t} is to try all possible pairs of $\overrightarrow{p_i, p_j}$ $(1 \le i < j \le n)$ to check whether $\epsilon(\overrightarrow{p_i, p_j}) \le \epsilon_t$. The time complexity of Step 1 is $O(n^3)$ because there exist $O(n^2)$ pairs of $(\overrightarrow{p_i, p_j})$ and the checking cost is $O(n)$, where n is the number of points in the trajectory. In Step 2, a breadth first search (BFS) procedure is employed to find the shortest path based on G_{ϵ_t}. The time complexity of Step 2 is $O(m^2)$ where m is the number of points in G_{ϵ_t}. For Step 3, it takes $O(m)$ time to find the solution. Therefore, the time complexity of the SP algorithm is $O(n^3)$.

According to the description of the SP algorithm, the dominant time-consumption part of SP focuses on Step 1. To improve the time efficiency of Step 1, a variant of the SP algorithm called SP-theo is proposed in [15]. SP-theo employs a concept called "feasible direction range" to reduce the time complexity of checking whether $\epsilon(\overrightarrow{p_i, p_j}) \le \epsilon_t$ from $O(n)$ to $O(c)$, where c is a small constant in cost cases. For a segment line $\overrightarrow{p_h, p_{h+1}}$ $(1 \le h < n)$ in a trajectory T, the feasible direction range of $\overrightarrow{p_h, p_{h+1}}$ with respect to one error tolerance ϵ_t, is denoted as $fdr(\overrightarrow{p_h, p_{h+1}}|\epsilon_t) = [\theta(\overrightarrow{p_h, p_{h+1}}) - \epsilon_t, \theta(\overrightarrow{p_h, p_{h+1}}) + \epsilon_t] \bmod 2\pi$.

Then, let $T[i,j] = \{p_i, p_{i+1}, \ldots, p_j\}$ be the sub-trajectory of T. The feasible direction range of $T[i,j]$ with respect to ϵ_t is denoted by $fdr(T[i,j]|\epsilon_t)$. Based on Lemma 4 in [15], if $\theta(\overrightarrow{p_i, p_j})$ is in $fdr(T[i,j]|\epsilon_t)$, $\epsilon_t(\theta(\overrightarrow{p_i, p_j})) \le \epsilon_t$. Therefore, checking whether $\epsilon(\theta(\overrightarrow{p_i, p_j})) < \epsilon_t$ in the SP-theo algorithm is equivalent to checking if $\epsilon(\theta(\overrightarrow{p_i, p_j}))$ is in $fdr(T[i,j]|\epsilon_t)$. Because the size of $fdr(T[i,j]|\epsilon_t)$ is bounded by $\min\{1 + \left\lfloor \frac{\epsilon_t}{\pi - \epsilon_t} \right\rfloor, j - i\}$, the computing cost can be bounded by a constant c. Therefore, the time complexity of constructing graph part of SP-theo is $O(c \cdot n^2)$, as there are $O(n^2)$ times of checking whether $\epsilon(\overrightarrow{p_i, p_j}) < \epsilon_t$ and each check can be done in $O(c)$ time with $fdr(T[i,j]|\epsilon_t)$. Meanwhile, the $fdr(T[i,j]|\epsilon_t)$ set can be incrementally computed using the following equation:

$$fdr(T[i,j]|\epsilon_t) = fdr(T[i,j-1]|\epsilon_t) \cap fdr(\overrightarrow{p_{j-1}, p_j}),$$
$$\text{where } 1 \le i < j \le n. \quad (3)$$

In this paper, we employ the SP-theo algorithm to compress an online trajectory stream S. The basic idea is that we incrementally compress S with the SP-theo algorithm and Eq. (3). The online compression algorithm is shown in Algorithm 1, which works as follows. Initially, we input all sample points in the first trajectory segment g_1 of S into Q (see line 6 in Algorithm 1). We then execute the SP-theo algorithm with Eq. (3) over these points to get a shortest path sp that is stored in Q (see line 8 and lines 14–21). We treat the sp as the compressed trajectory and append all points in sp to the tail of S' (see line 9). Next, we update Q to only keep the last two points (see line 10). The process is repeated until all trajectory segments in S are processed.

As an example, given a trajectory stream S consisting of two trajectory segments $\{g_1, g_2\}$, Fig. 1 illustrates the compression procedure. The trajectory segment $g_1 = \{p_1, p_2, p_3, p_4, p_5\}$ is compressed using SP-theo to get the shortest path $sp = \{p_1, p_2, p_3, p_5\}$ and store it in S'. We then merge the last two points $\{p_3, p_5\}$ and the next trajectory segment $g_2 = \{p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$

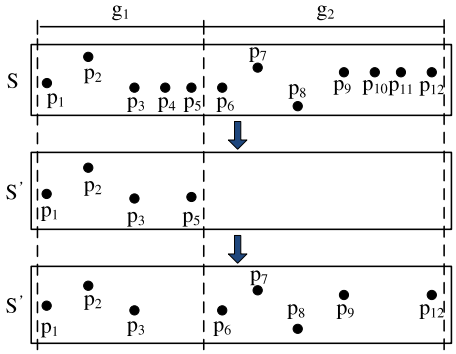


Fig. 1. Example of the online DPTS algorithm.

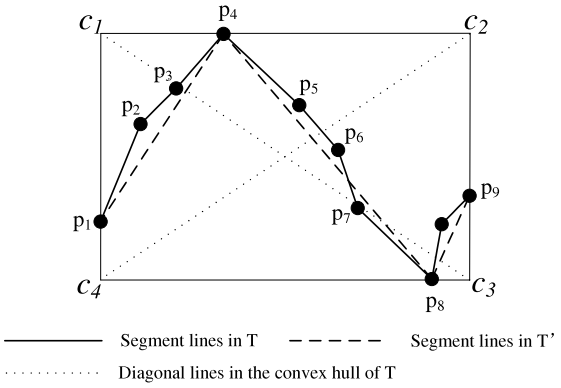


Fig. 2. Example of bounding $L_{\max}$.

into the new trajectory segment Q $= \{p_3, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$ and compress Q using SP-theo again. As a result, Q $= \{p_3, p_6, p_7, p_8, p_9, p_{12}\}$. Finally, we merge Q and S' in the previous round to get the current $S' = \{p_1, p_2, p_3, p_6, p_7, p_8, p_9, p_{12}\}$. Note that for each round we need to keep the last two points instead of just the last point to avoid keeping the points that should have been abandoned. For instance, in Fig. 1, the point p_5 can be removed through our scheme. As we can see, the proposed algorithm can process one trajectory stream by repeatedly calling the SP-theo algorithm in each trajectory segment. The time complexity of our algorithm is still $O(c \cdot n^2)$ for each trajectory segment, as we employed the SP-theo algorithm. Therefore, the algorithm processing efficiency still needs to be improved for processing the trajectory stream. In the following section, we propose how to improve the time efficiency of our algorithm.

Algorithm 1: The description of online DPTS

```
Online DPTS_Procedure(S, ϵ_t)/* Input: S is a
    trajectory streaming data on one time
    snapshot, ϵ_t is the upper bound on
    direction error tolerance. Output: S' is
    the compressed trajectory streaming data
    on the time snapshot.                  */
Initialize a queue Q ← {}
Initialize S' ← {}
Initialize a fdr set F ← {}
for each trajectory segment g_i ∈ S; i++ do
    Append all points in g_i into Q
    if Q.length > 2 then
        Q ← do_SP-theo(Q, F, ϵ_t)
        S' ← Q
        remove all points in Q but the last two points
    end
end
return S'
/* run the SP-theo algorithm on the current
    trajectory streaming data              */
do_SP-theo(Q, F, ϵ_t)
Compute incrementally all new fdr sets = {fdr set} from Q
  with F and equation (3)
F ← {fdr set} ∪ F
Construct a G_ϵt on Q with F
Set s_node as the first point in Q
Set e_node as the last point in Q
Get the shortest path sp from s_node to e_node with BFS
return sp
```

3.3. Algorithm optimization

As we can see, the time complexity of the proposed online DPTS for each trajectory segment is $O(c \cdot n^2)$ because there exist n^2 pairs of $(\overrightarrow{p_i, p_j})$ in each trajectory segment and the checking cost is O(c), where n is the number of points in one trajectory segment. Our heuristic optimization scheme is to reduce the number of pairs of $(\overrightarrow{p_i, p_j})$ before using the SP-theo algorithm to compress a trajectory segment g in S. Consequently, the total runtime of compressing S can be significantly reduced. To achieve this goal, we employed a BQS data structure applied in an online PPTS trajectory compression method [9,16] to filter trajectory segments. In [9,16], BQS is a convex hull that is formed by a bounding box and two angular bounds around all points to be compressed. Then, the PPTS compression method can be used to make fast compression decisions without calculating the maximum error in most cases. However, the BQS structure is based on the position error rather than the direction error. Therefore, we first propose one scheme to transform the direction error to the position error to employ the BQS structure, and then we introduce the optimized online DPTS algorithm using the BQS structure.

3.3.1. The transformation scheme

According to the descriptions of LEMMA 2 in [13], the DPTS method can give an error bound on the position ϵ_d when the direction error tolerance $\epsilon_t < \pi/2$. This means the following inequality holds:

$$\epsilon_d \leq 0.5 \cdot \tan(\epsilon_t) \cdot L_{\max}, \tag{4}$$

where $L_{\max} = Max(len(\overrightarrow{l_i}|T'))$ and $len(\overrightarrow{l_i}|T')$ is the distance length of a segment line $\overrightarrow{l_i}$ in the compressed trajectory T'. Therefore, by using the formula (4), it appears that we can transform a direction error ϵ_t into its corresponding position error ϵ_d and can apply the BQS to filter the trajectory stream S. However, the value of $L_{\max}$ in the formula (4) is generally unknown a priori unless we finish the whole compression procedure. To address this issue, we first present a theorem:

Theorem 1. *Given a compression trajectory T', $L_{\max}$ is bounded by the length of the diagonal line of the convex hull contain all points in T.*

Proof of Theorem 1. We use an example shown in Fig. 2 to prove Theorem 1. As we can see in Fig. 2, for a trajectory $T = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$, the corresponding convex hull is able to be represented by a box B $= \{C_1, C_2, C_3, C_4\}$. Given $T' = \{p_1, p_4, p_8, p_9\}$ is a compressed trajectory of T, it is easy to discover that the length of any diagonal line of B (i.e., C_1C_3 or C_2C_4) is greater than the length of any segment line in T'.

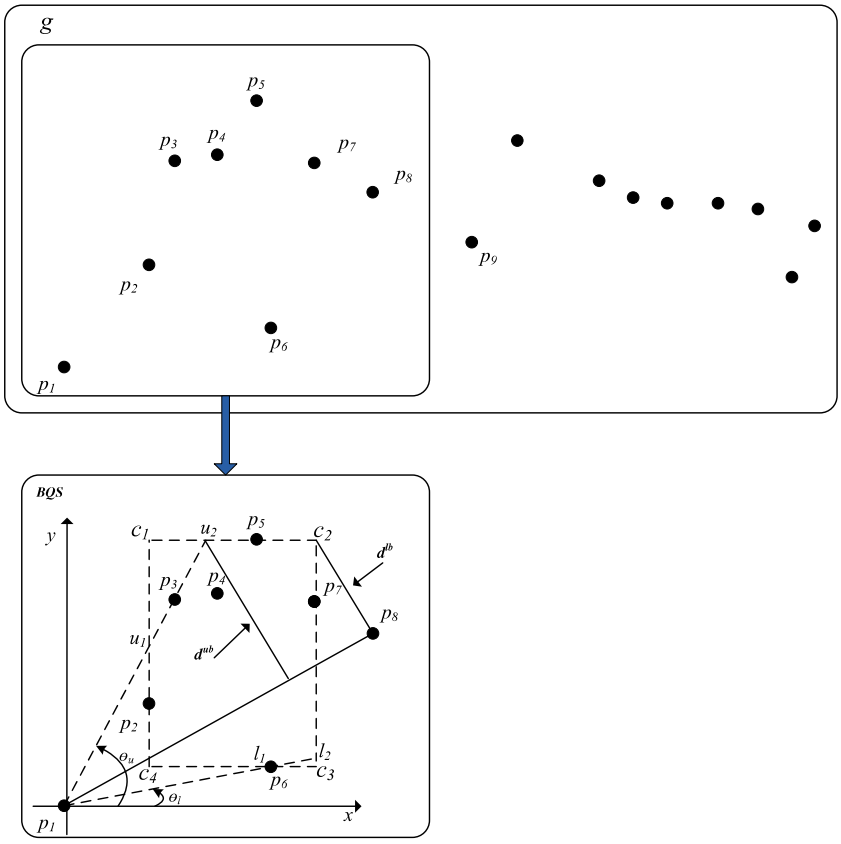


Fig. 3. Example of constructing BQS for a trajectory segment.

In our setting, a trajectory stream S consists of a set of trajectory segments $= \{g_1, g_2, \ldots, g_m\}$ so that the compression of S will sequentially compress these trajectory segments with the SP-theo algorithm (see algorithm 1). Therefore, for every compression procedure of one trajectory segment $g_i (1 \leq i \leq m)$, the value of $L_{\max}$, which is defined as $L_{\max}(g_i)$, can be bounded by the length of one diagonal line of a convex hull over all points in g_i that is defined to be $L_D(g_i)$. That means the position error of g_i, which is defined as $\epsilon_d(g_i)$, is bounded by the following inequality:

$$\epsilon_d(g_i) \leq 0.5 \cdot \tan(\epsilon_t) \cdot L_{\max}(g_i) \leq 0.5 \cdot \tan(\epsilon_t) \cdot L_D(g_i). \quad (5)$$

3.3.2. Optimizing the online DPTS algorithm using BQS

After the above-mentioned transformation from ϵ_t to ϵ_d, we can construct the corresponding BQS to filter each trajectory segment. The procedure is illustrated in Fig. 3.

In Fig. 3, for a trajectory segment g, we first buffer a few points (i.e., from p_1 to p_7). We then treat p_1 as the start point and p_8 as a new incoming point to be checked. For convenience, we assume all points (i.e., from p_2 to p_7) in the buffer are within $\epsilon_d(g)$ w.r.t. p_1. In fact, the assumption can be relaxed because we still can use the following SP-theo algorithm to process these points. Because the number of these points is very small, the compression performance is slightly influenced. Therefore, one BQS structure is constructed according to the following steps:

Step 1: Split the space into four quadrants from the start point p_1 of the current segment.
Step 2: For each quadrant where there exist points, a bounding box (i.e., $C_1C_2C_3C_4$) is set for points (from p_2 to p_7) in this quadrant.
Step 3: Two bounding lines record the smallest and greatest angles between the x axis and the line from the start point to any points for each quadrant (i.e., θ^l and θ^u).
Step 4: Get at most eight significant points, including four vertices on the bounding box (i.e., $C_1C_2C_3C_4$) and four intersection points from bounding lines intersecting with the bounding box (i.e., l_1, l_2, u_1, u_2).
Step 5: Based on the position deviations between lines from the start point to the significant points and the current path line(i.e., $\overrightarrow{p_1, p_8}$), we get a group of lower bound candidates and upper bound candidates for the maximum position deviation.
Step 6: From these candidates, a pair consisting of a lower bound and an upper bound $\langle d^{lb}, d^{ub} \rangle$ is derived to make compression decisions without the full computation of segment direction deviation in most of cases.

The pair of $\langle d^{lb}, d^{ub} \rangle$ can be computed using the formula (7), (8), (9), (10) in [9]. Based on the pair of bounds and the converted position error $\epsilon_d(g)$, the new incoming point p_8 can be determined using the following rules:

Rule 1: If the position distance between p_1 and $p_8 d(p_1, p_8) \leq \epsilon_d(g)$, p_8 belongs to the current segment and a new segment does not need to be started.
Rule 2: If $d^{ub} \leq \epsilon_d(g)$, p_8 belongs to the current segment and a new segment does not need to be started.
Rule 3: If $d^{lb} > \epsilon_d(g)$, p_8 breaks the tolerance and a new segment needs to be started.
Rule 4: If $d^{lb} \leq \epsilon_d(g) < d^{ub}$, p_8 cannot be determined using BQS.

To conveniently filter points, we set a state property f for each point to indicate whether they need to be filtered. If the value of f equals '1', it means it needs to be filtered and if the value is '0', it needs to remain in the trajectory segment. Therefore, when the incoming point p_8 satisfies rule 1 and rule 2, we set $p_8.f$ as '1' to filter this point; otherwise, we keep p_8 for further online DPTS compression. After determining the point p_8, we can continue

to process the rest of points. After processing all points in g, we compress all points with f = '0' using SP-theo. The optimized online DPTS compression algorithm (called online DPTS$^+$) is shown in Algorithm 2. Compared with the online DPTS algorithm, the new algorithm adds a filtering procedure before running compression (see line 9 in Algorithm 2 and Algorithm 3). Therefore, we can run the SP-theo algorithm with the time complexity of O(c × m^2), where $m \ll n$.

Algorithm 2: The description of online DPTS$^+$

```
1  Online DPTS+_Procedure(S, ε_t)/* Input: S is a
       trajectory streaming data on one time
       snapshot, and ε_t is the upper bound on
       direction error tolerance. Output: S' is
       the compressed trajectory streaming data
       on the time snapshot.                  */
2  Initialize a queue Q ← {}
3  Initialize S' ← {}
4  Initialize a fdr set F ← {}
5  for each trajectory segment g_i ∈ S; i++ do
6      Store all points in g_i in Q
7      if Q.length > 2 then
8          Q ← do_filterByBQS(Q, ε_t) //see algorithm 3
9          Q ← do_SP-theo(Q, F, ε_t) //see algorithm 1
10         S' ← Q
11         remove all points in Q but the last two points
12     end
13 end
14 return S'
```

Algorithm 3: Filtering using BQS

```
1  do_filterByBQS(Q, ε_t)
2  ε_d = 0.5 · tan(ε_t) · L_D(Q) /* transfer the direction
       error to the position error using formula
       (5)                                    */
3  set a tiny buffer B that contains the first λ points in Q, i.e.,
   Q[1:λ]
4  set the first λ - 1 points' filtering property f as '0'
5  set the first point in B as the start point s
6  set the last point in B as the new incoming point e
7  set i = λ and len = the length of Q
8  while i ≤ len do
9      if d(s,e) ≤ ε_d then // satisfy rule 1
10     Q[i].f = 1 and e → B
11     else
12         Construct or maintain a BQS structure over the buffer
           B
13         if d^ub ≤ ε_d(g) then // satisfy rule 2
14          Q[i].f = 1 and e → B
15         else
               // satisfy rule 3 or rule 4
16          Q[i-1].f = 0
17         s ← Q[i-1] // Current segment stops and
               new segment starts at the previous
               point before e
18
19         end
20     end
21 end
22 Update Q to only keep points whose f property equals 0
23 return updated Q
```

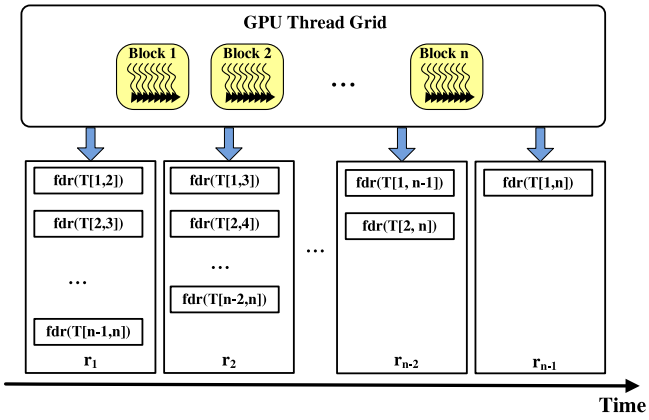


Fig. 4. The parallel scheme for constructing graph.

4. The GPU-aided online DPTS$^+$ method

To more efficiently compress trajectory streaming data using online DPTS$^+$, we focus on the solution for parallelizing our proposed online DPTS$^+$ using GPU. Because our proposed online DPTS$^+$ algorithm heavily depends on the SP-theo algorithm, we propose a way to parallelize the SP-theo algorithm.

According to the description of the SP-theo algorithm in Section 3.2, the graph construction (Step 1) and the shortest path finding (Step 2) are dominant in terms of time costs. Therefore, a parallel scheme is proposed to accelerate Step 1, and then we introduce how to employ a GPU-aided BFS to improve the computing performance of Step 2.

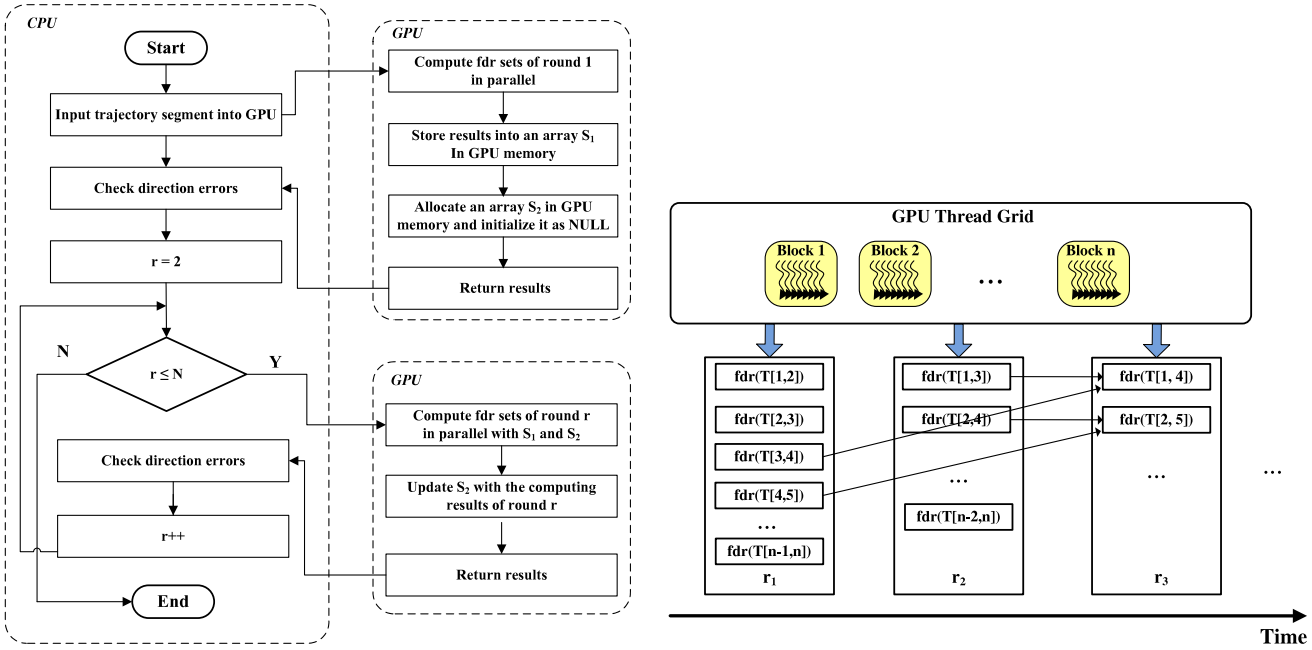
4.1. A parallel scheme on GPUs for graph construction

According to the above-mentioned descriptions about the graph construction of SP-theo in Section 3.2, we can see that the key point to parallelize graph construction is to parallelize the computational procedure of O(n^2) times of checking whether $\epsilon(\overrightarrow{P_i, P_j}) < \epsilon_t$, with each check taking O(c) time with $fdr(T[i,j]|\epsilon_t)$. Therefore, we propose a parallel scheme for Step 1 based on the method of computing $fdr(T[i,j]|\epsilon_t)$ with the incremental property in [13]. In [13], $fdr(T[i,j]|\epsilon_t)$ can be incrementally computed using Eq. (3): $fdr(T[i,j]|\epsilon_t) = fdr(T[i,j-1]|\epsilon_t) \cap fdr(\overrightarrow{P_{j-1}, P_j})$ where $1 \le i < j \le n$. So, after $j-i$ rounds, $fdr(T[i,j]|\epsilon_t)$ can be computing using the following equation:

$$fdr(T[i,j]|\epsilon_t) = \cap_{i \le h < j} fdr(\overrightarrow{P_h, P_{h+1}}|\epsilon_t). \quad (6)$$

Our parallel scheme is shown in Fig. 4. We compute all $fdr(T[i,j]|\epsilon_t)$ to check whether $\epsilon(\overrightarrow{P_i, P_j}) < \epsilon_t$ by executing the GPU kernels in $n-1$ rounds. In round 1, we compute all $fdr(T[h, h+1]|\epsilon_t)$ in parallel where $h \in [1, n-1]$ and store these results in GPU global memory for the following computing. In round r ($2 \le r < n$), we can parallelize the computational procedure for all $fdr(T[h, h+r]|\epsilon_t)$ where $h \in [1, n-r]$ based on Eq. (6) and all $fdr(T[h, h+1]|\epsilon_t)$ stored in global memory. For example, in round 2 we compute $fdr(T[1,3]|\epsilon_t), fdr(T[2,4]|\epsilon_t), \ldots, fdr(T[n-2,n]|\epsilon_t)$ in parallel. Of these, $fdr(T[1,3]|\epsilon_t)$ can be computed using Eq. (6) (i.e., $fdr(T[1,3]|\epsilon_t) = fdr(T[1,2]|\epsilon_t) \cap fdr(T[2,3]|\epsilon_t)$).

For our parallel scheme, two aspects can be improved. The first one is that as the value of r increases, the number of sets that intersected in Eq. (6) also increases. Thus, we can parallelize the procedure of Eq. (6) as well. For instance, for the n-1 round in Fig. 4, $fdr(T[1,n]|\epsilon_t) = fdr(T[1,2]|\epsilon_t) \cap fdr(T[2,3]|\epsilon_t) \cap fdr(T[3,4]|\epsilon_t) \ldots \cap fdr(T[n-1,n]|\epsilon_t)$.



(a) The flowchart for optimizing the computation of fdr sets.

(b) The effect of the optimization method.

Fig. 5. Illustrating the optimization of computing fdr sets.

We can optimize the computing procedure using well-known GPU parallel reduction methods in [20]. The second aspect is that we only store fdr sets computed in round 1 in the GPU memory, bearing in mind that the GPU memory space is very limited. Consequently, in our scheme, all fdr sets in other rounds are computed based on the round's t fdr sets. This scheme involves many repeated computations. For example, we compute $fdr(T[1,4]|\epsilon_t)$ and $fdr(T[2,5]|\epsilon_t)$ in parallel in round 3 as the following procedure:

$$fdr(T[1,4]|\epsilon_t) = fdr(T[1,2]|\epsilon_t) \cap fdr(T[2,3]|\epsilon_t) \cap fdr(T[3,4]|\epsilon_t).$$
$$fdr(T[2,5]|\epsilon_t) = fdr(T[2,3]|\epsilon_t) \cap fdr(T[3,4]|\epsilon_t) \cap fdr(T[4,5]|\epsilon_t).$$

In this procedure, the computation for $fdr(T[2,3]|\epsilon_t) \cap fdr(T[3,4]|\epsilon_t)$ is executed two times. To avoid this, we temporarily store all fdr sets of the last round in GPU memory. Thus, we can directly compute the fdr set of the current round based on the fdr sets of the last round and the partial fdr sets of first found. This method is illustrated in Fig. 5. Fig. 5(a) shows we attempt to store the computation results of fdr sets for the last round in GPU memory to accelerate the computing procedure of the current round. The additional space cost is that we need to keep an array S_2 with a maximum size of $|S_1| - 1$. Fig. 5(b) presents an example of the optimized effect of my method. In this example, we can employ the computing results stored in GPU memory that is $fdr(T[1,3]|\epsilon_t)$ and $fdr(T[2,4]|\epsilon_t)$ in round 2 to accelerate the computations of $fdr(T[1,4]|\epsilon_t)$ and $fdr(T[2,5]|\epsilon_t)$ in round 3.

4.2. The BFS implementation on GPUs

After constructing the graph with error tolerance threshold ϵ_t, G_{ϵ_t}, SP-theo runs a BFS algorithm on G_{ϵ_t} to find the shortest path from p_1 to p_n. Therefore, we employ a fast BFS implementation on GPUs (i.e., BFS-4K [21]) to accelerate this step.

The BFS-4K method uses the concept of frontier in [22] for parallel visits in one graph. Given one BFS tree generated by BFS has root s and contains all reachable vertices, the vertices in each level of the BFS tree make up a frontier. A procedure called *frontier propagation* is executed to form the BFS tree. The frontier propagation procedure checks every neighbor of a frontier vertex to see whether it has already been visited already. If not, the neighbor is added to a new frontier. BFS-4K implements the frontier propagation using two data structures, Fd and Fd_{new}. Fd represents the actual frontier, which is read by the parallel threads to start the propagation step. Fd_{new} is written by the threads to generate the frontier. Then, Fd_{new} is filtered to guarantee the correctness of the BFS visit and is swapped with Fd for the next iteration.

In our setting, we only need to find the shortest path from p_1 to p_n on G_{ϵ_t} using BFS-4K instead of computing all shortest paths between any two vertices. Therefore, we start the BFS-4K procedure from p_1 and terminate the frontier propagation once P_n is retrieved. We slightly modified the frontier propagation of BFS-4K to find the shortest path, which is shown in Algorithm 4. As we can see, the BFS-4K starts from p_{start} referring it as a root in the tree (see line 2) and Fd is set by p_{start} (line 4). Then, multiple iterations are run. Each iteration consists of two steps: propagation step (lines 6–9) and filtering step (lines 10–16). In the propagation step, the proposed techniques in [21] can be used to optimize this step, including exclusive prefix-Sum, dynamic virtual warps, dynamic parallelism, and edge-discover. In the filtering step, after filtering Fd_{new} with the hash table method in [21], we check whether the end point p_{start} is in the leaves of tree or not. If it is, we can terminate the finding procedure. Otherwise, the tree grows by one level and Fd is swapped by Fd_{new} for the next iteration.

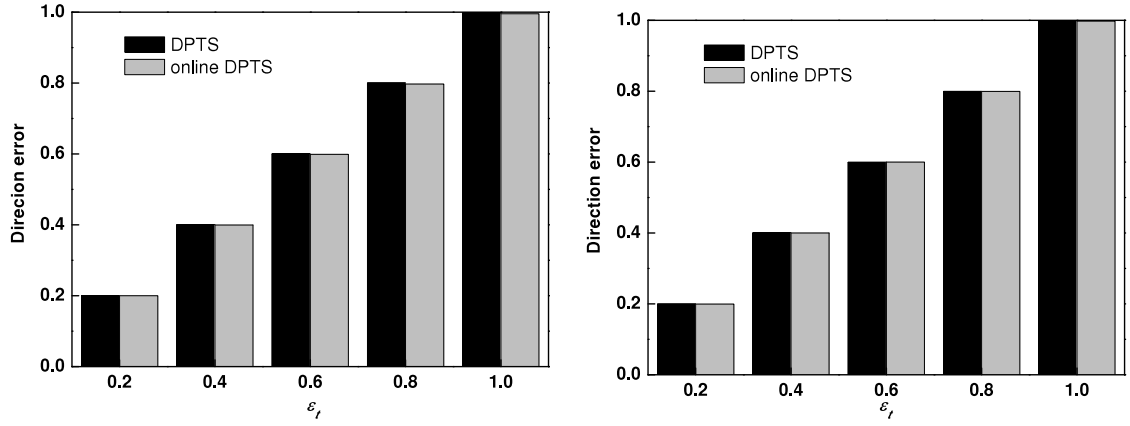
Note that for Step 3 in the SP-theo solution generation, the final compressed trajectory T' can be acquired from the tree using a parallel tree traversal.

5. Performance evaluation

We have evaluated the performances of the proposed online DPTS, online DPTS$^+$, and the GPU-aided online DPTS$^+$ against trajectory data streaming using a cutting-edge NVIDIA GPU. These experiments mainly concern direction error, compression rate, and compression time.

5.1. Experimental setup

The trajectory datasets used in this paper come from T-Drive trajectories [23] and GeoLife trajectories [22]. T-Drive recorded the



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 6. The comparison of direction error with DPTS.

Table 1
Datasets.

Set name	# of trajectories	Total # of positions	Average # of positions per trajectory	Directional difference between two adjacent segments
T-Driver	10,359	17,740,902	1713	(0.657, 0.803)
GeoLife	17,621	24,876,978	1412	(0.364, 0.615)

Algorithm 4: Finding the shortest path using BFS-4K

FindingSP_Procedure(G, p_{start}, p_{end}, **tree**) /* Input: G is one graph for BFS-4K, p_{start} is the start point, and p_{end} is the end point for BFS. Output: the tree is a BFS tree where p_{start} is its root node and p_{end} is located in its leaf nodes. */
tree.root ← p_{start}
tree.level++
Fd ← tree.root
while *true* **do**
  /* propagation step */
  foreach *each vector* $v \in Fd$ ***in parallel*** **do**
    ns ← finding Fd's neighbors
    Fd_{new} ← ns
  end
  /* filtering step */
  filtering Fd_{new} **in parallel**
  fill updated Fd_{new} into tree's leaves
  if $p_{end} \in$ *tree.leaves* **then**
    return tree
  end
  Fd ← Fd_{new}
  tree.level++
end

trajectories of 33,000 taxis over a period of three months in Beijing, and GeoLife contains 17,621 trajectories from different GPS loggers and GPS phones, with different sampling rates. The features of the two datasets are shown in Table 1.

All experiments were executed on one computer equipped with a Maxwell GPU (GTX TITAN X), and the configurations are presented in Table 2.

5.2. Evaluating the online DPTS algorithm

In this section, we evaluate the direction error, compression rate, and runtime of online DPTS. For comparison, we used the offline DPTS method, that is SP-theo. We randomly selected some trajectories from two datasets. For each trajectory, SP-theo processed the whole trajectory, while the online DPTS compressed the set of trajectory segments.

Table 2
Configurations of the computer.

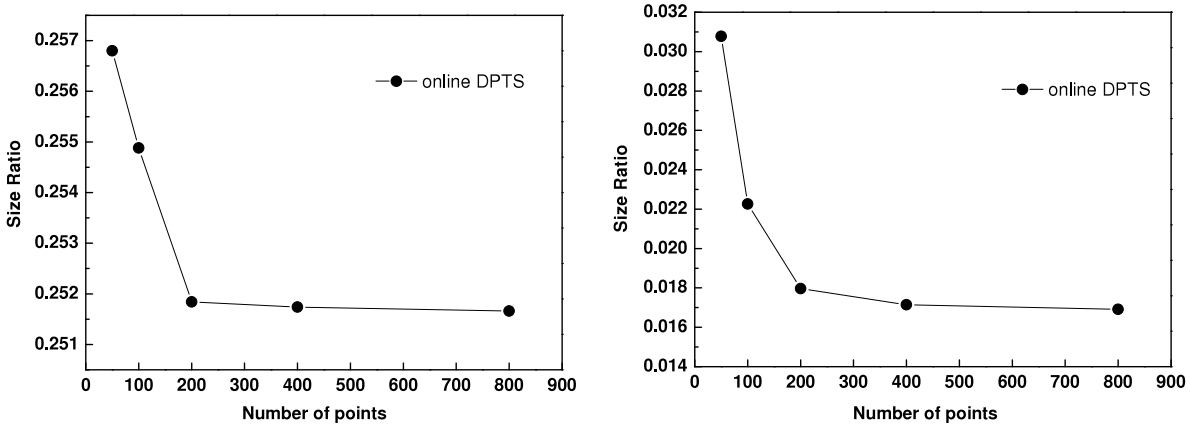
Specifications of CPU platforms	Computer
OS	Ubuntu14.04
CPU	i7-5820k (3.3 GHz, 6 cores)
Memory	32 GB DDR4
Specifications of GPU platforms	**GTX TITAN X**
Architecture	Maxwell
Memory	12 GB DDR5
Bandwidth	Bi-directional bandwidth of 16 GB/s
CUDA	SDK 7.0

5.2.1. Direction error

In this experiment, we randomly selected 10 trajectories from T-Driver and GeoLife and tuned the error tolerance ϵ_t ranging from 0.2 to 1 to compare the average direction error between DPTS and online DPTS. Each segment contains 200 points (the reason for selecting 200 is introduced in the following experiments). Fig. 6 shows that there is no difference in terms of direction error between the DPTS and online DPTS for two datasets. This indicates online DPTS can have the same direction tolerance as DPTS.

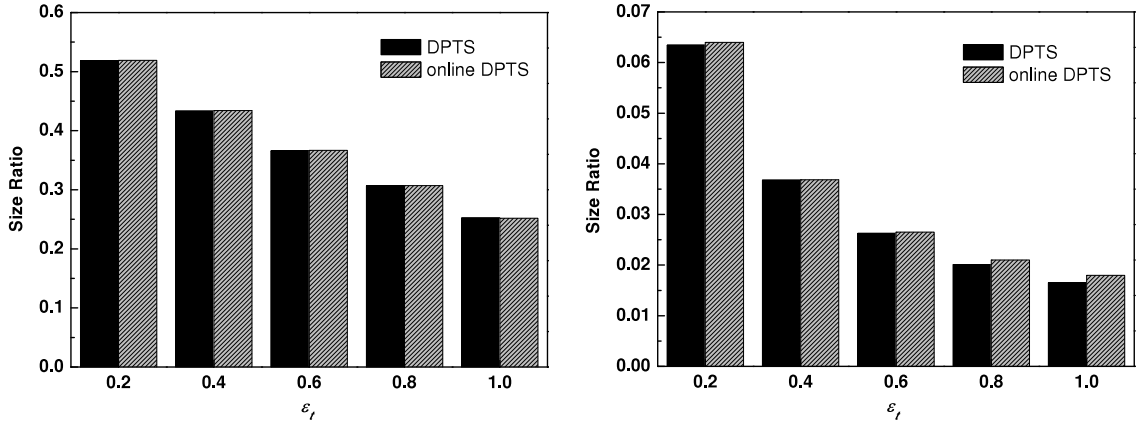
5.2.2. Compression rate

In this section, we investigate the compression rate of online DPTS. The compression rate is measured by *size ratio* that is defined in [13] and is equal to $\frac{\sum_{T' \in D'} |T'|}{\sum_{T \in D} |T|}$, where D is the set of raw trajectories and D' is the set of the corresponding compressed trajectories. We first observed the effect of segment size on the compression rate of our online DPTS, and then we compared the compression rate of online DPTS against the one of SP-theo. In the first experiment, we fixed the number of trajectory points at 5000 and the error tolerance ϵ_t at 1 to observe the size ratio of online DPTS with different segment sizes ranging from 50, 100, 200, 400 to 800. The experimental results in Fig. 7 show that the size ratio of online DPTS can keep a steady value when segment size is greater than 200 in both datasets.



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 7. The effect of segment size on online DPTS.



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 8. The compression rate of online DPTS.

In the second experiment, the segment size was fixed at 200 and the number of trajectory points was fixed at 5000. The segment size was set to 200 because we observed that the compression rate had no obvious change when the segment size ≥ 200 in the previous experiment. We then compare the size ratio of online DPTS against DPTS under different value of error tolerance from 0.2, 0.4, 0.6, 0.8 to 1. Fig. 8 shows that the size ratio of online DPTS is slightly higher than the one of DPTS for both datasets. The reason is that online DPTS compresses multiple trajectory segments while DPTS simplifies the whole trajectory, so online DPTS keeps a few trajectory points that can be removed if compressing the whole trajectory using DPTS. However, the difference between online DPTS and DPTS in terms of size ratio is very slight.

5.2.3. Runtime

In this section, we observe the compression time of online DPTS. We set the segment size at 200 (the reason is shown in the above experiments in terms of compression rate). We also set the $\epsilon_t = 1$ to observe the runtime of online DPTS and DPTS under different trajectory sizes (ranging from 2000 to 10,000).

The experiment results in Fig. 9 shows that online DPTS is faster than DPTS about 11% for the T-Driver dataset and 79% for the GeoLife dataset. The great performance gain of online DPTS compared to DPTS is because that the SP-theo algorithm in the online DPTS can run on some small-size trajectory segments. Meanwhile, the reason why the gain for the GeoLife dataset is much better than the T-Driver dataset is that the compression rate of the T-Driver is much lower than the one in the experimental results in Fig. 8. Therefore, both DPTS and online DPTS take much less time to compress T-Driver trajectories than GeoLife trajectories. Therefore, the superiority of online DPTS over DPTS is not obvious for the T-Driver dataset.

5.3. Evaluating the online DPTS$^+$ algorithm

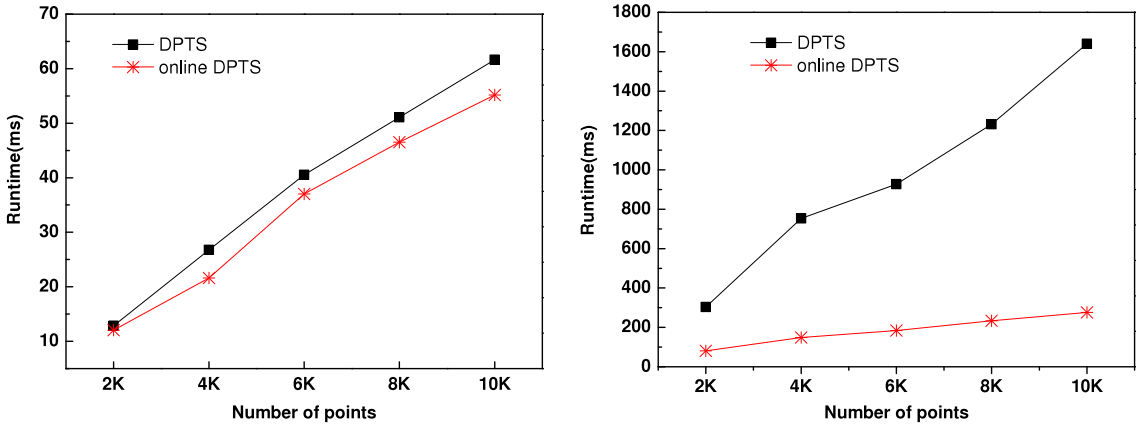
In this section, we evaluate the performance of proposed online DPTS$^+$ in terms of direction error, compression rate, and compression time against the online DPTS algorithm. For all experiments in this section, we set the segment size $= 200$.

5.3.1. Direction error

In this experiment, we also randomly selected 10 trajectories from T-Driver and GeoLife and tuned the error tolerance ϵ_t ranging from 0.2 to 1 to compare the average direction error between online DPTS and online DPTS$^+$. Fig. 10 shows there is no difference in terms of direction error between the online DPTS$^+$ and online DPTS for the two datasets. That indicates online DPTS$^+$ has the same direction tolerance as online DPTS.

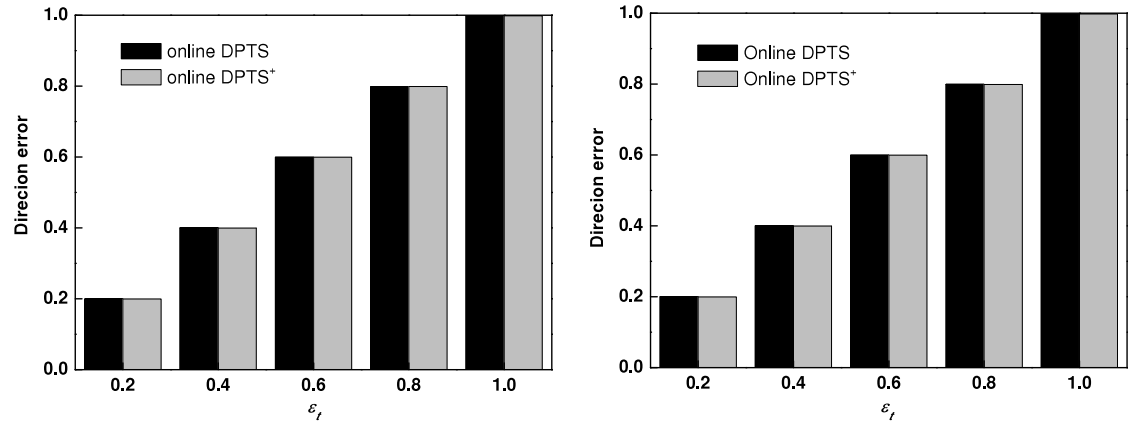
5.3.2. Compression rate

In this section, we investigate the compression rate of online DPTS$^+$. The compression rate is measured by *size ratio*. The number of trajectory points is fixed as 5000. We compare the size ratio of



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 9. The compression time of online DPTS.



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 10. The comparison of direction error with online DPTS.

online DPTS$^+$ against online DPTS under different values of error tolerance from 0.2, 0.4, 0.6, 0.8 to 1. According to the experimental results in Fig. 11, online DPTS$^+$ has the same compression rate as the online DPTS for the two datasets. This proves The filtering procedure in online DPTS$^+$ has no influence on the compression rate.

5.3.3. Runtime

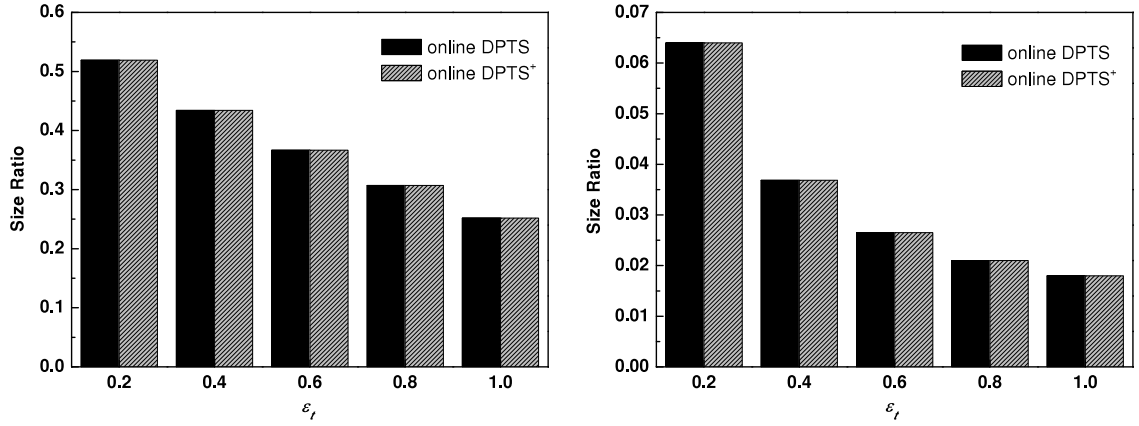
In this section, we investigate the time efficiency of online DPTS$^+$ processing trajectories. Because the core part of online DPTS$^+$ lies in pruning the trajectories with BQS, we first observed the pruning power of online DPTS$^+$ using the measuring method in [9]. The method uses a metric called pruning power, denoted as PP, which is defined as $1 - \frac{N^{computed}}{N^{total}}$, where $N^{computed}$ is the number of points needed to be computed with SP-theo and the number of total points. We randomly selected 10 trajectories with the fixed size $=$ 10,000 points and changed the direction error tolerance ϵ_t ranging from 0.2 to 1 to observe the average PP values. As we can see in Fig. 12, the average pruning power of online DPTS$^+$ is 70.5% for T-Driver and 85.1% for GeoLife. The reason the PP value for T-Driver is lower than that for GeoLife is because that GeoLife trajectories are easier to compress than T-Driver trajectories according to experiments about compressing rate (see Figs. 8 and 11), so that online DPTS$^+$ can prune more points from GeoLife than T-Driver.

Another observation is that our pruning power is lower than the one (90%) in [9]. The main reason for this is that the online DPTS$^+$ algorithm only prunes the points complying with Rule 1 and Rule 2 (see lines 23–35 in Algorithm 2) to accelerate the following SP-theo algorithm, while the method in [9] uses Rule 1, Rule 2, and Rule 3 to run its online trajectory compression algorithm.

In the following experiment, we evaluate the runtime of online DPTS$^+$. We set the $\epsilon_t = 1$ to observe the runtime of online DPTS$^+$ and online DPTS under different trajectory sizes (ranging from 2000 to 10,000). Because the online-DPTS$^+$ algorithm consists of two parts, filtering and compression, the time overheads of these two parts are measured individually. The experiment results in Fig. 13 show that online DPTS$^+$ outperforms online DPTS by an average of 2.23 times for the T-Driver dataset and 3.95 times for the GeoLife dataset. The results indicate that online DPTS$^+$ can significantly reduce time consumption by employing BQS structure to decrease the number of pairs of $\overrightarrow{p_i}, \overrightarrow{p_j}$ to check whether $\epsilon(\overrightarrow{p_i}, \overrightarrow{p_j}) \leq \epsilon_t$ during the compression. Meanwhile, the performance gain for GeoLife dataset is better than T-Driver dataset because online DPTS$^+$ can prune more points for GeoLife than T-Driver shown in Fig. 12. Additionally, the filtering time of online DPTS$^+$ takes about 30.1% of the whole time cost for T-Driver and 13.6% for GeoLife.

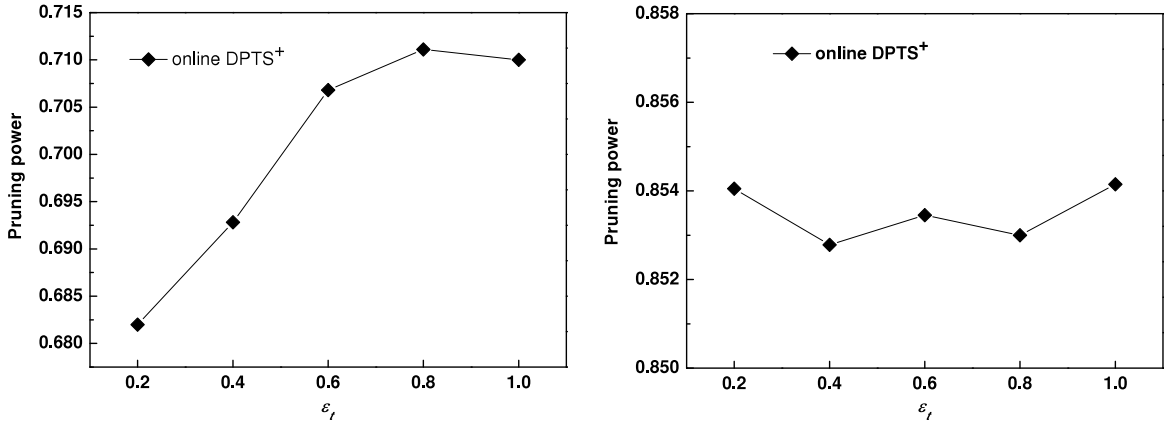
5.4. GPU-aided online DPTS$^+$ method evaluation

In this section, we observe the runtime of the GPU-based online DPTS$^+$ method when handing trajectory data. The SP-theo algorithm is the most time consuming part in online DPTS$^+$, and



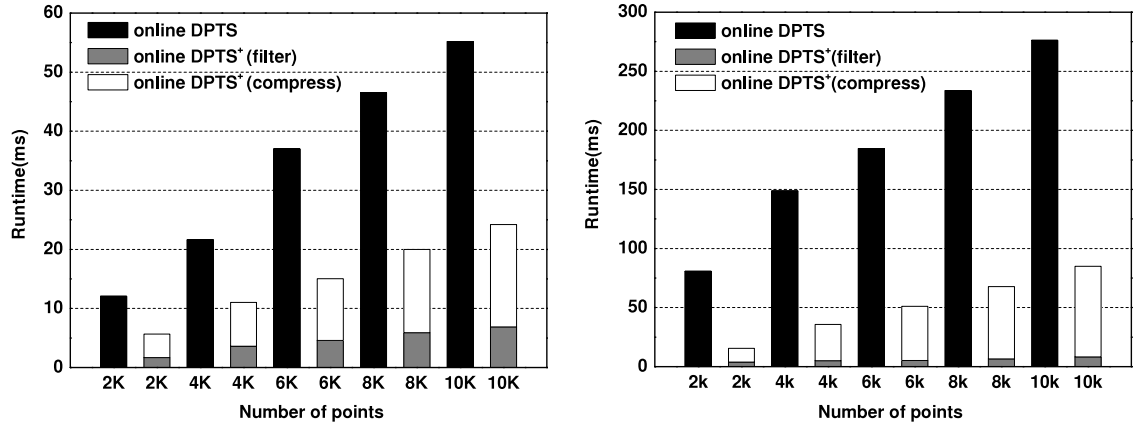
(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 11. The compression rate of online DPTS$^+$.



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 12. The pruning power of online DPTS$^+$.



(a) T-Driver dataset. (b) GeoLife dataset.

Fig. 13. The runtime of online DPTS$^+$.

the graph construction and shortest path finding dominate the time costs in the SP-theo algorithm. Therefore, we evaluate the time efficiency of the GPU-aided graph construction method and the method of shortest path finding in the following experiments. For convenience, we refer to graph construction and shortest path finding in the SP-theo algorithm as GC and SPF. Thus, the GPU-based graph construction and its advanced version are called G-GC and G-GC$^+$, respectively. The shortest path finding based on BFS-4K is called G-SPF.

5.4.1. Evaluating the GPU-aided graph construction

In this experiment, we randomly selected 10 trajectories whose lengths are more than 35K points from both the T-Drive and GeoLife datasets. We replaced the graph construction in SP-theo

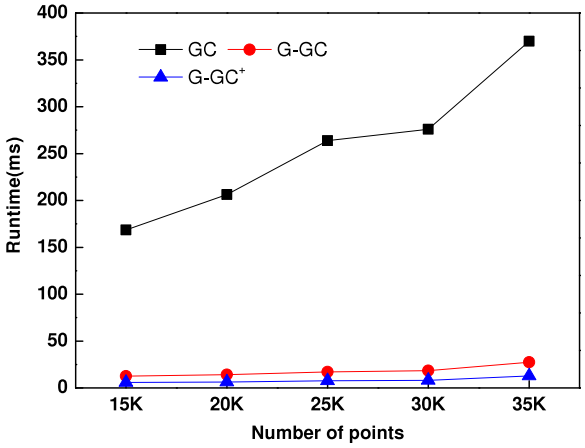


Fig. 14. The runtime of GPU-aided graph construction.

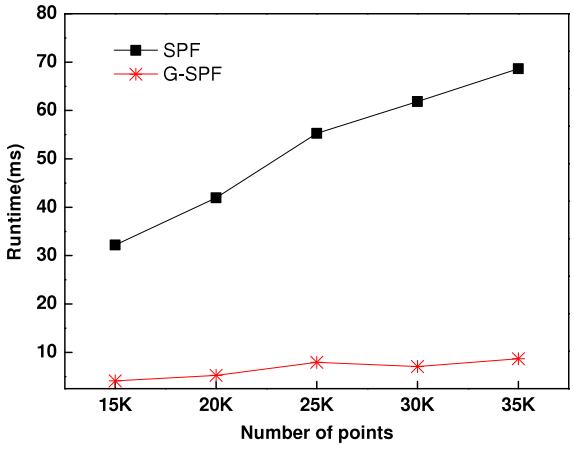


Fig. 15. The runtime of GPU-aided shortest path finding.

with the GPU-aided graph construction. We then observed the average time consumption of GC and G-GC for handling 10 sub-trajectories with various sizes (from 15K points to 35K points). The error tolerance ϵ_t was set to 1. The experimental results in Fig. 14 show that G-GC was 14.3 times faster than GC on average. Furthermore, G-GC$^+$ improved G-GC by about 2.2 times on average.

5.4.2. Evaluating the shortest path finding based on BFS-4K

In this experiment, based on the graphs from the experimental results in Fig. 14, we investigate the time consumption of G-SPF compared with SPF. Fig. 15 shows that G-SPF outperforms SPF about by 7.88 times on average. However, we also observed that the performance gain is less than the experimental results in [21]. The reason is that online DPTS$^+$ divided a trajectory into multiple segments and also used the BQS structure to filter trajectories so that the number of points in SPF stage is very small, which hinders the GPU's parallelism.

6. Conclusions and future work

This paper addresses the need to compress online trajectory streaming data by preserving direction information. We converted an offline DPTS algorithm into an online DPTS method and applied a BQS data structure in an online PPTS method to optimize our proposed online DPTS method, called online DPTS$^+$. The proposed online DPTS$^+$ meets the need to quickly compress trajectory streaming data. A parallel method for online DPTS$^+$ has been developed to ensure the performance of compressing trajectory streaming data with the support of a contemporary Maxwell GPU. The proposed approach provides a new tool for fast online trajectory stream compression.

The experimental results show that (1) the online DPTS outperforms offline DPTS with up to 79% less compression time while maintaining a comparable compression error and compression rate, (2) the compression time of online-DPTS$^+$ algorithm is 3.95 times faster than that of online DPTS, and (3) the GPU-aided methods can significantly reduce the time for the graph construction and for the shortest path finding with a speedup of 31.4 and 7.88 (on average), respectively. For future work, we will extend the compressed approach to process trajectories in the road network and consider the streaming inconsistency issue.

Acknowledgments

This work was supported in part by National Science and Technology Major Project of the Ministry of Science and Technology of China (2016ZX05014-003), the China Postdoctoral Science Foundation (2014M552112), China University of Geosciences (Wuhan) (No. 1610491B24).

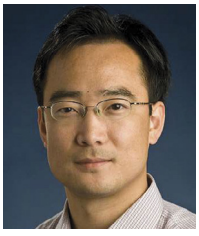
References

[1] X. Chen, S. Hu, Distributed generation placement for power distribution networks, J. Circuits Syst. Comput. 24 (2015) 1550009-1–1550009-23.
[2] X. Chen, T. Wei, S. Hu, Uncertainty-aware household appliance scheduling considering dynamic electricity pricing in smart home, IEEE Trans. Smart Grid 4 (2013) 932–941.
[3] I.S. Popa, K. Zeitouni, V. Oria, A. Kharrat, Spatio-temporal compression of trajectories in road networks, Geoinformatica preprint.
[4] J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, T. Wolle, Compressing spatio-temporal trajectories, LNCS 4835 (2007) 763–775.
[5] D.H. Douglas, T.K. Peucker, Algorithms for the reduction of the number of points required to represent a line or its caricature, Canad. Cartographer 10 (1973) 112–122.
[6] N. Meratnia, R.A. de By, Spatiotemporal compression techniques for moving point objects, in: International Conference on Extending Database Technology, EDBT, 2004, pp. 765–782.
[7] Y. Zheng, X. Zhou, Computing with Spatial Trajectories, Springer, 2011.
[8] R. Lange, F. Dürr, K. Rothermel, Efficient real-time trajectory tracking, VLDB J. 20 (2011) 671–694.
[9] J. Liu, K. Zhao, P. Sommer, S. Shang, B. Kusy, R. Jurdak, Bounded quadrant system: Error-bounded trajectory compression on the go, in: The IEEE International Conference on Data Engineering, ICDE, 2015, pp. 987–998.
[10] N. Meratnia, R.A. de By, Spatiotemporal compression techniques for moving point objects, LNCS 2992 (2004) 765–782.
[11] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, D. Vaccaro, On-line data reduction and the quality of history in moving objects databases, in: ACM International Workshop on Data Engineering for Wireless and Mobile Access, MobiDE, 2006, pp. 19–26.
[12] J. Muckell, P.W.O. Jr., J.-H. Hwang, C.T. Lawson, S.S. Ravi, Compression of trajectory data: a comprehensive evaluation and new approach, Geoinformatica 18 (2014) 435–460.
[13] C. Long, R.C. Wong, H.V. Jagadish, Direction-preserving trajectory simplification, Proc. VLDB Endow. 6 (2013) 949–960.
[14] C. Long, R.C. Wong, H.V. Jagadish, Trajectory simplification: On minimizing the direction-based error, Proc. VLDB Endow. 8 (2014) 49–60.
[15] C. Long, R.C.-W. Wong, H.V. Jagadish, Direction-preserving trajectory simplification, in: International Conference on Very Large Data Bases, 2013, pp. 949–960.
[16] J. Liu, K. Zhao, P. Sommer, S. Shang, B. Kusy, J.-G. Lee, R. Jurdak, A novel framework for online amnesic trajectory compression in resource-constrained environments, IEEE Trans. Knowl. Data Eng. PP (2016).
[17] J.S. Vitter, Random sampling with a reservoir, ACM TOMS 11 (1985) 37–57.
[18] F. Xhafa, V. Naranjo, L. Barolli, M. Takizawa, On streaming consistency of big data stream processing in heterogenous clutsers, in: 18th International Conference on Network-Based Information Systems, NBiS, 2015, pp. 476–482.
[19] L. Golab, T. Johnson, Consistency in a stream warehouse, in: Conference on Innovative Data Systems Research, 2011, pp. 114–122.
[20] NVIDIA CUDA C Programming Guide version 6.5, 2015.
[21] F. Busato, N. Bombieri, BFS-4K: An efficient implementation of BFS for kepler GPU architectures, IEEE Trans. Parallel Distrib. Syst. 26 (2015) 1826–1838.

[22] Y. Zheng, X. Xie, W.Y. Ma, Geolife: A collaborative social networking service among user, location and trajectory, IEEE Data Eng. Bull. 33 (2010) 32–40.
[23] J. Yuan, Y. Zheng, X. Xie, G. Sun, Driving with knowledge from the physical world, in: KDD, 2011, pp. 949–960.

Ze Deng received the B.Sc. degree from the China University of Geosciences, the M.Eng. degree from Yunnan University, and the Ph.D. degree from the Huazhong University of Science and Technology, China. He is currently an assistant professor with the School of Computer Science, China University of Geosciences, Wuhan, China. He is currently also a postdoctor with the Faculty of Resources, China University of Geosciences, Wuhan, China.

Wei Han received the B.Sc. degree from China University of Geosciences. He is currently working toward the master's degree with the School of Computer Science, China University of Geosciences, Wuhan, China. His research interests include data management, highperformance computing, and neuroinformatics.



Lizhe Wang (SM'09) received the B.Eng. degree (with honors) in electrical engineering with a minor in applied mathematics and the M.Eng. degree in electrical engineering, both from Tsinghua University, Beijing, China, and the doctor of engineering degree in applied computer science (magna cum laude) from the University Karlsruhe (now Karlsruhe Institute of Technology), Karlsruhe, Germany. He is a '100-Talent Program' professor at Institute of Remote Sensing & Digital Earth, Chinese Academy of Sciences (CAS), Beijing, China and a 'ChuTian' chair professor at School of Computer Science, China University of Geosciences, Wuhan, China. He is a senior member of the IEEE.

Rajiv Ranjan received the Ph.D. degree in 2009 in engineering from the University of Melbourne. He is a research scientist and a julius fellow in CSIRO Computational Informatics Division (formerly known as CSIRO ICT Centre). His expertise is in data center cloud computing, application provisioning, and performance optimization. He has published 62 scientific, peer-reviewed papers (seven books, 25 journals, 25 conferences, and five book chapters). His hindex is 20, with a lifetime citation count of 1660+ (Google Scholar). His papers have also received 140+ ISI citations. Seventy percent of his journal papers and 60% of conference papers have been A*/A ranked ERA publication. He has been invited to serve as the guest editor for leading distributed systems journals including IEEE Transactions on Cloud Computing, Future Generation Computing Systems, and Software Practice and Experience. One of his papers was in 2011's top computer science journal, IEEE Communication Surveys and Tutorials.

Albert Y. Zomaya (F'04) received the Ph.D. degree from the Department of Automatic Control and Systems Engineering, Sheffield University in the United Kingdom. He is currently the chair professor of High Performance Computing & Networking and Australian Research Council Professorial Fellow in the School of Information Technologies, The University of Sydney. He is also the director of the Centre for Distributed and High Performance Computing which was established in late 2009. He held the CISCO Systems chair professor of Internetworking during the period 2002–2007 and was also the head of school for 2006–2007 in the same school. Prior to his current appointment he was a full professor in the School of Electrical, Electronic and Computer Engineering at the University of Western Australia, where he also led the Parallel Computing Research Laboratory during the period 1990–2002. He served as an associate-, deputy-, and actinghead in the same department, and held numerous visiting positions and has extensive industry involvement. He is a fellow of the IEEE.

Wei Jie has been active in a broad spectrum of areas in parallel and distributed computing, in particular, grid and cloud computing, computing security technologies, e-science and e-research. Dr. Jie has been actively involved in professional services. He is the General Chair of the IEEE workshop on Security in e-Science and e-Research, and has served as Program Committee member for more than 40 international conferences and workshops. Dr. Wei Jie has published approximately 50 papers in international journal and conferences and has edited three books.