Effective Online Group Discovery in Trajectory Databases

Xiaohui Li, Vaida Čeikutė, Christian S. Jensen, *Fellow*, *IEEE*, and Kian-Lee Tan

Abstract—GPS-enabled devices are pervasive nowadays. Finding movement patterns in trajectory data stream is gaining in importance. We propose a group discovery framework that aims to efficiently support the online discovery of moving objects that travel together. The framework adopts a sampling-independent approach that makes no assumptions about when positions are sampled, gives no special importance to sampling points, and naturally supports the use of approximate trajectories. The framework's algorithms exploit state-of-the-art, density-based clustering (DBScan) to identify groups. The groups are scored based on their cardinality and duration, and the top-k groups are returned. To avoid returning similar subgroups in a result, notions of domination and similarity are introduced that enable the pruning of low-interest groups. Empirical studies on real and synthetic data sets offer insight into the effectiveness and efficiency of the proposed framework.

Index Terms—Moving objects, trajectory, travel patterns

1 Introduction

Today's Internet-enabled mobile devices are equipped with geopositioning sensors that can readily identify location information, notably GPS data. This has resulted in the availability of rapidly increasing volumes of trajectory data that capture the movements of a variety of objects, including smartphone users, animals, vehicles, and vessels.

This development renders it increasingly important to be able to extract movement patterns from trajectory data. A central pattern is that of objects that have traveled together. Existing definitions, notably *flock* [12], [13], *convoy* [17], and *swarm* [21], capture different notions of objects traveling together. For example, a flock consists of objects that travel together within a disk of a user-specified radius. However, the disk radius influences the result greatly and is a difficult-to-set parameter. Convoys, on the other hand, do away with this parameter by employing the notion of density-connectedness. Swarms also relax the requirement that objects must form groups for consecutive time points.

We may think of the true trajectory tr_{T} of a moving object as a continuous function from the time domain $\mathbb{T}$ to points in the n-dimensional euclidean space $\mathbb{R}^n$, where the movement occurs. For most typical applications, $n = 2$. In practice, the trajectory of an object is collected by sampling the object's positions according to some policy, resulting in a set of sampling points $(t, \vec{r}) \in \mathbb{T} \times \mathbb{R}^n$. A trajectory tr is then given by the stepwise linear function obtained by connecting temporally consecutive sampling points with line segments, as shown in Fig. 1a. Different sets of sampling points may represent the same such function. This holds for $\{(3,3),(7,7)\}$ and $\{(3,3),(5,5),(7,7)\}$, shown in Fig. 1b.

The use of different representations (sampling points) of the same trajectory in an algorithm should not affect the outcome of the algorithm. We call this property *sampling independence*.

Definition 1 (Sampling Independence[1]). *Let two trajectory sampling point sets tr_a^{rep} and tr_b^{rep} that represent the same trajectory be denoted as $tr_a^{rep} \equiv tr_b^{rep}$. Next, two collections of trajectory sampling point sets, TR_0 and TR_1, represent the same trajectories, denoted as $TR_0 \equiv TR_1$, if and only if $\forall i \in \{0,1\}$ $(\forall tr_a^{rep} \in TR_i$ $(\exists tr_b^{rep} \in TR_{1-i}(tr_a^{rep} \equiv tr_b^{rep})))$.*

With these definitions in place, we say that an algorithm A operating on collections of trajectory point sets satisfies *sampling independence* if and only if $\forall \boldsymbol{TR}_a, TR_b(TR_a \equiv TR_b \Rightarrow A(TR_a) = A(TR_b))$.

Sampling independence has several benefits. First, we observe that sampling-dependent approaches that simply compare trajectories as of given sampling times suffer from the lossy-pattern problem. Consider the trajectories of two objects o_1 and o_2 in Fig. 1b and assume that for a set of objects to form a group, their distance e must not exceed 2 for at least 2 time units. The two trajectories satisfy this requirement because they are within distance 2 from time 3 to time 5. However, the two trajectories are sampled at different times. If we introduce an artificial sampling point for o_2 at time 3, which existing techniques (e.g., convoys and swarms) would do to enable comparison, no group is found because we are missing a sampling point at time 5. Sampling independence rules out such approaches.

Second, the approach of adding sampling points so that all points are sampled at the same times is not scalable. To

- *X. Li and K.-L. Tan are with the School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417. E-mail: {lixiaohui, tankl}@comp.nus.edu.sg.*
- *V. Čeikutė and C.S. Jensen are with the Department of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark. E-mail: {ceikute, csj}@cs.au.dk.*

Manuscript received 31 Oct. 2011; revised 20 Mar. 2012; accepted 22 Sept. 2012; published online 1 Oct. 2012.
Recommended for acceptance by T. Sellis.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-10-0663.
Digital Object Identifier no. 10.1109/TKDE.2012.193.

1. While the term "sampling point independence" may be more precise, we use the shorter term "sampling independence" throughout this paper.

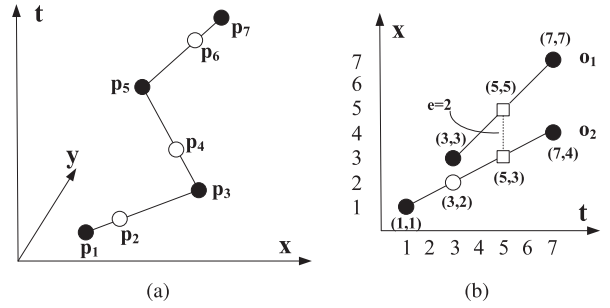


Fig. 1. Trajectory semantics and pattern loss.

see why, assume a trajectory collection TR in which each trajectory is sampled S times, and let the sampling times be chosen at random from a continuous time domain. Each time a new trajectory is added to the collection, a total of $2|TR|S$ artificial sampling points must be added (S to each existing trajectory, and $|TR|S$ to the new trajectory). While these assumptions are extreme, the argument illustrates well that sampling dependence does not scale with $|TR|$ and S.

Third, sampling independence offers a foundation for seamless integration of approximate trajectories. With sampling independence, a computation does not have to occur at each sampling time, and approximate trajectories can be used seamlessly in place of more accurate trajectories.

Sampling independence requires decoupling defining and finding travel-together patterns from sampling points. Among the existing definitions, only flocks satisfy this property, while the other definitions, for example, of convoy and swarm, are based on sampling points and so cannot be used in a sampling-independent framework.

Our framework achieves sampling independence by first defining a travel-together pattern that is independent of sampling points, and second, by adopting an event-based approach that is also sampling independent.

The second property that the framework considers is density-connectedness that aims to avoid the lossy-flock problem identified by Jeung et al. [17]. The problem occurs because 1) a flock is very sensitive to a user-specified disc size that is independent of the data distribution and 2) the use of circular shape may not always be appropriate. To achieve this property, our framework employs DBScan [10] for continuous clustering.

The third property of the framework is to support online processing, where sampling data arrive continuously. Efficiency is a key factor in online processing. Processing the raw trajectories may be too time consuming, so the framework needs to support online trajectory simplification that trades result accuracy with efficiency. This is the fourth property of the framework.

Table 1 categorizes previous work and our proposed solution (denoted Group) with respect to the four requirements as discussed above. Previous work is discussed in detail in Section 2.2.

In our framework, a *group* is a cluster that has at least m moving objects being density-connected for at least a specified *duration* of time. This definition fundamentally differs from sampling dependent definitions (e.g., convoy and swarm) by requiring moving objects to be density-connected for a time duration instead of at consecutive or nonconsecutive sampling times (continuous versus discrete).

TABLE 1
Algorithm Comparison

Property	Flock	Convoy	Swarm	Group
Sampling independence	✓	✗	✗	✓
Density connectedness	✗	✓	✓	✓
Traj. approximation	✓	✓	✓	✓
Online processing	✓	✗	✗	✓

To find groups from a collection of trajectories without relying on sampling points, our main idea is to predict events that can happen in the future according to the motions of the moving objects. This is carried out by the *continuous clustering* function module in the framework (Fig. 2). This module first uses DBScan to find clusters with an arbitrary spatial extent and to avoid a disk radius parameter. As new data arrive, these clusters are continuously monitored. Objects exiting or joining a cluster and cluster expiry or merging are all modeled as future events, and then processed when they actually occur. The event-based approach enables sampling independence—computation occurs when events occur, not at sampling times. This design also enables online processing: as new trajectory samples arrive, events can be derived and processed in due course. There is no need to recompute the entire result. Online trajectory simplification is applied when new data arrive to smoothen the trajectories and, thus, reduce the total number of events.

The second module serves to handle efficiently the history of each existing group to find groups to return to the user. This module inherently has high time complexity because it potentially has to consider exponential numbers of subgroups. To improve its efficiency, we avoid materializing every subgroup by exploiting the relations between subgroups. The main idea is that a group may have many subgroups that may contain a subset of its objects or may exist for a subinterval of its time interval. Such dominated subgroups do not offer new information and are pruned early.

Next, two groups may share too many common members. We propose a similarity measure for groups and return only a set of groups such that no two groups are more similar than a given similarity threshold.

Further, we score groups according to their cardinality and duration and then return the top-k groups. To enable this functionality, the framework maintains a candidate result list and a variable *minScore* that is the lowest score among all the candidates found so far. For each group to be scored, the framework first computes an upper bound on the score of the group for its entire lifespan. The group is pruned if the upper bound is smaller than *minScore*.

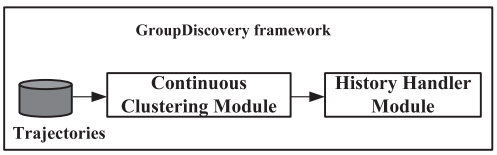


Fig. 2. The sampling-independent framework.

TABLE 2
Symbols Summary

N	Total Number of trajectories
N_c	Cardinality of a group
δ	Tolerance for line simplification
τ	Duration threshold
e	Distance threshold in density-based clustering
m	Cardinality threshold in density-based clustering
ρ	Duration of a cluster
θ	Similarity threshold

Otherwise, possible groups with higher scores in the cluster are inserted into the candidate list, and *minScore* is updated.

In Section 2, we formalize the notions covered above.

The main contributions of the paper are as follows:

- We propose a novel definition of *group* that we believe enables the computation of interesting patterns.
- We propose a corresponding, efficient group discovery framework that is the first to satisfy the four requirements listed in Table 1.
- We conducted an extensive empirical study on both real and synthetic data sets to evaluate the proposed framework. Our results offer insight into the effectiveness and efficiency of the proposed framework.

The remainder of the paper is organized as follows: Definitions and related work are covered in Section 2. We describe the GroupDiscovery framework in Section 3. Section 4 reports on the empirical evaluation of the GroupDiscovery framework. Finally, Section 5 concludes this paper with directions for future work.

2 Preliminaries and Related Work

2.1 Definitions

We proceed to define the group query along with supporting definitions. The notations used in this section and throughout the paper are summarized in Table 2.

As we aim to return interesting groups to the user, we introduce a scoring function for groups.

Definition 2 (Scoring Function). *Let C be a set of N_c objects that travel together during a time interval $[t_s, t_e]$. The score of C is*

$$\text{Score}(C) = \alpha N_c + (1 - \alpha)\rho,$$

where α $(0 \leq \alpha \leq 1)$ is a user-provided parameter and $\rho = (t_e - t_s)$.

The idea underlying the definition is that larger and longer groups are more interesting than smaller and shorter groups. Parameter α allows us to balance cardinality and duration.

Next, we define a domination relation that allows us to prune groups that do not contribute any new information. Some other related work, for example, [12] and [21], has similar notions, either explicitly or implicitly.

Definition 3 (Domination). *Given two groups C_1 and C_2, we say C_1* dominates *C_2, denoted by $C_1 \prec C_2$, if:*

1. *C_1 is a superset of C_2, and*
2. *the time interval of C_1 contains that of C_2.*

Now, we are ready to define the group query to be supported.

Definition 4 (Group Query). *Given a trajectory collection TR, a distance threshold e, a cardinality threshold m, a duration threshold τ, an integer k, and the scoring function given above, the group query returns k groups of objects, so that each group*

1. *is a density-connected cluster w.r.t. e and m,*
2. *has a duration no shorter than τ,*
3. *is not dominated by any other group, and*
4. *has a top-k score.*

2.2 Related Work

2.2.1 Co-Movement Discovery

Several recent proposals aim to identify objects that travel together based on trajectory data.

The notion of flock was introduced by Laube and Imfeld [19] and further studied by others [1], [3], [5], [12], [13], [25]. A flock is a set of moving objects that travel together in a disk of radius r for a time interval whose duration is at least k consecutive time points. One similar notion identifies a set of objects as a moving cluster [18] if the objects when clustered at consecutive sampling times exhibit overlaps above a given threshold. Another similar notion, Herd [14], relies on the F-score to identify cluster overlaps at consecutive sampling times. A recent study by Jeung et al. [17] proposes the notion of convoy that uses density connectedness [10] for spatial clustering. Aung and Tan [2] proposed the notion of evolving convoys to better understand the states of convoys. Specifically, an evolving convoy contains both dynamic members and persisted members. As time passes, the dynamic members are allowed to move into or out of the evolving convoy, creating many stages of the same convoy. At the end, the evolving convoys with their stages are returned. The differences between evolving convoys and our work are: 1) evolving convoys are sampling dependent, and 2) instead of collapsing convoys into stages, our work relies on a scoring function to return meaningful results. The advantages of the scoring approach are its simplicity and the control it offers.

In contrast to the above, the notions of group pattern [28] and swarm [21] permit patterns where moving objects travel together for a number of nonconsecutive sampling times. Group patterns rely on disk-based clustering, while swarms use density connectedness for the spatial clustering.

None of the above proposals satisfies the four requirements that motivate our work.

To avoid running expensive incremental clustering (e.g., DBscan) at each time point, our proposal predicts the time when an event may occur. The motivation is that the number of events can be much fewer than the total number of time points, which can be very large, depending on the sampling technique.

The techniques used for supporting convoy and swarm discovery are capable of exploiting trajectory simplification. In convoy discovery, line segments are first clustered to identify candidate clusters. Then, a refinement step considers the sampling points for these candidates. Thus, the accurate trajectories are only needed in the refinement step.

The techniques used for swarm discovery apply sampling in a preprocessing step.

The techniques for computing convoys and swarms cannot be adapted easily to online settings. Convoys are computed by partitioning the time domain into intervals, upon which line segment clustering is applied to each interval. When new data stream in, the time domain needs to be redivided, and the computation needs to start from scratch. The techniques used for swarm discovery assume a static trajectory collection; without this assumption, swarms may not be maximal w.r.t. time. The pruning rules used in swarm discovery also require the time domain to be known beforehand. In contrast, our proposed solution employs an existing online simplification technique in the preprocessing step, monitors clusters *continuously* and records the histories of clusters. The evaluation of groups is only dependent on the most recent history, so the GroupDiscovery framework is amenable to online processing.

2.2.2 Moving Objects and Trajectory Clustering

The continuous clustering of the current positions of moving objects is related to the problem studied here. Jensen et al. [16] proposed a disk-based, incremental approach to continuously cluster moving objects. The scheme incrementally maintains and exploits a summary structure, called a *clustering feature*, for each cluster. During a time period, a moving object may be inserted into, or deleted from, a moving cluster. Next, the techniques monitor the average radiuses of moving clusters, which change over time as the member objects move. When the average radius of a cluster c exceeds a threshold, c is split. In addition, a cluster is split if its cardinality exceeds a given threshold. Clusters may be merged if their cardinalities fall below a threshold.

Since this approach is disk based, it is lossy as pointed out by Jeung et al. [17]. As such, it cannot be directly applied in our scheme, which uses density-connectedness to avoid the lossy problem.

Another line of research is trajectory clustering, in which the goal is to find the common paths of a group of moving objects. Trajectory clustering builds on advances in time-series data analysis. Thus, many clustering techniques that resemble their counterparts in time-series data analysis have been proposed, such as Dynamic Time Warping (DTW) [29], Longest Common Subsequences (LCSS) [27], Edit Distance on Real Sequence (EDR) [7], Edit distance with Real Penalty (ERP) [6], and a partition-and-group framework [20]. Li et al. [21] point out that these approaches are ill suited to find groups because the trajectories in a group may be quite different although they are close to each other (e.g., straight line trajectories versus wave-like trajectories).

2.2.3 Trajectory Simplification

Trajectory simplification can improve the efficiency of many algorithms that operate on the trajectories by removing relatively unimportant data points. Trajectory simplification algorithms can be classified into batch or online algorithms. Batch algorithms, such as the Douglas-Peucker (DP) algorithm [9] and its variants, require the entire trajectory to be available, and thus are expected to produce relatively high-quality approximation. In contrast, online algorithms, such as reservoir sampling algorithms [26] and sliding window algorithms [22], can work with partial trajectories and can be used for compressing data streams.

The GroupDiscovery framework employs the Normal Opening Window (NOPW) algorithm [22]. This algorithm starts by initializing an empty sliding window and setting the first point as an *anchor point* p_a. When a new location point p_i is added into the sliding window, a line segment $\overline{p_a p_i}$ is used to fit every location point in the sliding window. If no location point deviates from $\overline{p_a p_i}$ by more than a user-specified error bound, the sliding window grows by including the next new point p_{i+1}. Otherwise, the point with the highest error p_e is selected. The line segment $\overline{p_a p_e}$ is included as part of the approximate trajectory, and p_e is set as the new anchor point. The time complexity of NOPW is $O(n^2)$, where n is the number of data points in a trajectory.

In another line of research, Fagin et al. [11] propose the Threshold Algorithm (TA) that has some similarity with the history handler module. TA operates on a database where each object has m grades, one for each of m attributes, for example, a multimedia database. Given a monotone aggregation function, for example, min or average, that combines the individual grades to obtain an overall grade, TA finds the objects with top-k overall grades by concurrently accessing the sorted list of the attributes. It is shown that TA is instance optimal.

3 Group Discovery Framework

In this section, we describe the GroupDiscovery framework and its two functional modules in detail. This online algorithm (Algorithm 1) first initializes the variable U to store the unclassified objects, H to store the history of the clusters, and R to store the result. It then receives new trajectory data at each iteration, and calls NOPW to simplify the trajectory that are then processed by the Continuous Cluster and History Handling modules to find groups.

Algorithm 1: DiscoverGroups$(e, m, \tau, k, \delta, \theta)$

```
1 U ← ∅; H ← ∅; R ← ∅;
2 while true do
3 |   TR ← receiveNewTR();
4 |   NOPW(TR, δ);
5 |   FindContinuousCluster(TR, e, m, τ, k, δ, U, H) ;
6 |   S ← RevHist(H);
7 |   CheckCandidates(S, R, θ, k);
```

3.1 Continuous Clustering Module

3.1.1 Overview

Given a trajectory collection TR, we aim to discover groups of objects that travel together. A trajectory represents the movement history of an object for some time interval. As all trajectories are not defined for the same time interval, TR represents the movement of a dynamic set O of objects. The objects in O at time t can be retrieved by the function $getObjects(TR, t)$.

The overall approach is to maintain a heap *eventQ* that contains events that affect the clustering of the moving objects. Each event is associated with the time when it occurs,

and the heap is ordered on these times. Events are inserted into *eventQ* that correspond to the object movements given by the trajectories in TR. The module then processes events in *eventQ* in time order, causing additional events to be entered into *eventQ* and eventually processed.

An event has the format $(t, obj, cid_1, cid_2, type)$, where t is the time when the event occurs, obj is an object, cid_1 and cid_2 are cluster identifiers, and $type$ is the event type, which is one of the following:

- *APPEAR*. An object appears in the system. Four mutually exclusive cases can happen: The object may stay by itself being unclassified; it may join an existing cluster, thus causing a JOIN event to be created; it may cause multiple clusters to merge, thus causing one or more MERGE events to be created; or it may form a new cluster with other objects.
- *DISAPPEAR*. An object disappears from the system. As for object appearance, there are four cases: No clusters are affected; a cluster loses a member, causing an EXIT event to be created; a cluster splits into multiple smaller clusters; or a cluster expires, which results in the creation of an EXPIRE event.
- *UPDATE*. An object updates its velocity, which affects the predicted exit time of object(s), which in turn possibly affects the merging or expiry of clusters.
- *EXIT*. An object exits from a cluster. The exit event can cause a cluster to expire, for example, because it has fewer than m members, in which case an EXPIRE event is inserted into *eventQ*.
- *JOIN*. An object joins a cluster.
- *EXPIRE*. A cluster expires. When this occurs, the former cluster members may be in no new cluster, rendering them unclassified.
- *MERGE*. Multiple (at least two) clusters merge.
- *SPLIT*. One cluster splits into multiple clusters.

Among these events, the first three events, APPEAR, DISAPPEAR, and UPDATE, can be read directly from the data stream, and thus are called *primary events*. In contrast, EXIT, JOIN, EXPIRE, MERGE, and SPLIT are *secondary events*.

Each object o in the system has a data structure $(oid, \vec{p}, \vec{v}, cid, label)$, where oid is its identifier, $\vec{p}$ and $\vec{v}$ are its current location and velocity, and cid is the identifier of the cluster o currently belongs to, if any. An object o initially has $o.cid = 0$, indicating that it is unclassified. The field $label$ represents the type of the object in the cluster. In DBScan [10], clusters contain two types of objects. A *core object* has at least $MinPts$ objects in its e-neighborhood, where $MinPts$ is a user-specified parameter that we set to m. Any other object is a *border object*. A core object has ($o.label = \text{CORE}$); a border object has ($o.label = \text{BORDER}$).

The top-level algorithm keeps the currently unclassified objects in a set U, and attempts to cluster them each time an event occurs. In addition to *eventQ*, we employ a mapping getMemByCid from cluster identifiers to the sets of objects in a cluster.

Upon termination of the algorithm, the histories of clusters are recorded in a data structure that is passed on to the history handler module for computation of domination relations and thus the true groups.

3.1.2 Event Processing

Algorithm 2 contains the pseudocode of FindContinuousCluster, the top-level continuous clustering algorithm. It takes the following arguments: a trajectory collection TR, the four parameters e, m, τ, and k from the group query definition, δ for RangeQuery, and U and H for storing the unclassified objects and the cluster history, respectively.

Algorithm 2: FindContinuousCluster($TR, e, m, \tau, k, \delta, U, H$)

```
1  insert new primary events into eventQ;
2  while eventQ is not empty do
3      evt ← eventQ.pop;
4      O ← getObjects(TR, evt.t);
5      C ← getMemByCid(evt.cid);
6      if evt.type = APPEAR then U.add(evt.obj);
7      else if evt.type = DISAPPEAR then
8          HandleObjDisapp(evt, O, H);
9      else if evt.type = UPDATE then
10         HandleObjUpdate(evt, O, H);
11     else if evt.type = EXIT then
12         o ← evt.obj; C.delete(o); o.cid ← 0;
13         U.add(o); update H;
14         if o.label = BORDER then CheckCore(o, C);
15         if o.label = CORE then
16             L ← o.getNeighbors();
17             foreach o' ∈ L do
18                 if o'.label = BORDER then
19                     L' ← o'.getNeighbors();
20                     if L' has no core objects then
21                         C.delete(o'); o'.cid ← 0;
                           U.add(o');
22         if (|C| < m) ∨ (C has no core objects) then
23             insert (evt.t, o, C.cid, ⊥, EXPIRE) into
               eventQ;
24     else if evt.type = JOIN then
25         o ← evt.obj; o.cid ← C.cid;
26         C.add(o); U.delete(o);
27         update H;
28         if o.label = BORDER then
29             t ← getExitTime(o, C);
30             insert (t, o, C, EXIT) into eventQ;
31     else if evt.type = EXPIRE then
32         mark members in C unclassified;
33         update H;
34     else if evt.type = MERGE then
35         HandleMerge(evt, O, H); update H;
36     else if evt.type = SPLIT then
37         HandleSplit(evt, O, H); update H;
38     Insert(U, O, e, m, τ, δ, H);
```

The algorithm initially scans TR and inserts APPEAR, DISAPPEAR, and UPDATE events into *eventQ*. The algorithm then repeatedly pops and processes events. Given an event $(t, o, \perp, \perp, \text{APPEAR})$, o is inserted into U. Given an event $(t, o, \perp, \perp, \text{DISAPPEAR})$, o is removed from the system. If o is a member of the cluster C, o is also removed from C which then expires if it has less than m members.

Given an event $(t, o, \perp, \perp, \text{UPDATE})$, if o is a border object of the cluster C identified by cid, o's exit time is recomputed and updated in *eventQ*. If o is a core object, the algorithm recomputes the exit times of the border objects in o's e-neighborhood and the expected split time, and

updates the *eventQ*. Lastly, if o is a unclassified object, the procedure Insert is eventually called to cluster o.

Given an event $(t, o, cid, \perp, \text{EXIT})$, the object o is removed from the cluster identified by cid and marked as unclassified. If o is a border object, CheckCore procedure is called to check whether o's core object still has enough neighbors. If o is a core object, o's border neighbors that are not density-connected by any other core objects are also removed. The cluster identified by cid expires if it has fewer than m members after the event. Given an event $(t, o, cid, \perp, \text{EXPIRE})$, all objects in the identified cluster C are marked as unclassified and inserted into U.

Given an event $(t, o, cid_1, cid_2, \text{MERGE})$, the two clusters identified by cid_1 and cid_2 are merged. The cid field of the members of the new cluster is updated.

Given an event $(t, o, cid, \perp, \text{SPLIT})$, the cluster identified by cid splits into two clusters. Depending on their specific connectedness to core members, the members in C have their cid field updated. In Algorithm 2, we provide pseudocode for APPEAR, EXIT, and JOIN events.

The Insert procedure used at the end of Algorithm 2 inserts a unclassified object into an existing cluster if the object moves into the cluster, and detects any new clusters in U that contains the set of unclassified objects. The detailed pseudocode of the Insert procedure is shown in Algorithm 3. It first initializes a variable G that records already classified objects that are in the e-neighborhood of any object in U. Variable G is populated by calling a RangeQuery procedure for each object o in U that returns all objects in O that are in the e-neighborhood of o (lines 2-6).

Algorithm 3: Insert$(U, O, e, m, \tau, \delta, H)$

```
1  G ← ∅;
2  foreach o ∈ U do
3      L ← RangeQuery(o, e + 2δ, O);
4      foreach o′ ∈ L do
5          if o′.cid > 0 then
6              G.add(o′);
7  foreach o ∈ G do
8      C ← getMemByCid(o.cid);
9      L ← RangeQuery(o, e + 2δ, O);
10     if |L| ≥ m then
11         L.delete(o); o.label ← CORE;
12         ExpandCluster (O, o, C, L, e + 2δ, m, U, H);
13 if |U| ≥ m then
14     CS ← DBScan(U, e + 2δ, m);
15     foreach C ∈ CS do
16         foreach o ∈ C do
17             if o.label = BORDER then
18                 t ← getExitTime(o, C);
19                 insert (t, o.oid, C.cid, EXIT) into
                   eventQ;
20 update H;
```

In lines 7-11, if the RangeQuery result for an object $o \in G$ contains at least m objects, o's label is updated to CORE, and every unclassified object in o's neighborhood is removed from U and inserted into the cluster that o belongs to. The procedure ExpandCluster (line 12) recursively expands a cluster by exploring the new core object. After every object in G is considered, if U still has at least m objects, the procedure clusters the remaining objects in U (lines 13-19). It uses a variable CS that contains a set of clusters. First, it clusters all the moving objects as of the current time. Second, it calculates the exit time for each border object. These objects and their time to exit are inserted into *eventQ* as events.

The ExpandCluster procedure expands a cluster by recursively exploring the new core object of the cluster. The detailed pseudocode is shown in Algorithm 4. In the parameters, $crObj$ is a core object of cluster C and L is $crObj$'s current neighbors. The global variable $crTime$ records the current time. For each $o' \in L$, if o' is unclassified, a JOIN event is detected (line 3). The global variable $crTime$ records current time. Then, the procedure retrieves o' neighbors in line 4. If o' has at least m neighbors, o' is also a core object, and thus ExpandCluster is called recursively (lines 5-7). If o' has fewer than m neighbors (it is a border object), the procedure calculates the exit time of o'. An EXIT event is created for o' and inserted into *eventQ* (lines 8-11).

Algorithm 4: ExpandCluster$(O, crObj, C, L, e, m, U, H)$

```
1  foreach o′ ∈ L do
2      if o′.cid = 0 then
3          insert (crTime, o′, C.cid, JOIN) into eventQ;
4          L′ ← RangeQuery(o′, e, O);
5          if |L′| ≥ m then
6              o′.label ← CORE;
7              ExpandCluster(O, crObj, C, L, e, m, U, H);
8          else
9              o′.label ← BORDER;
10             t ← getExitTime(o′, C);
11             insert (t, o′, C.cid, EXIT) into eventQ;
12     else if o′.cid ≠ crObj.cid ∧ o′.label = CORE
       then
13         insert (crTime, o′, C.cid, MERGE) into
           eventQ;
14 update H;
```

When o' is a core object from another cluster, cluster C is merged with the cluster identified by $o'.cid$. A MERGE event is created and inserted into *eventQ* (line 13).

3.1.3 Detecting Cluster Expiry and Split Events

The exit of a single object can cause a cluster to *expire*. We show how to detect when cluster expiry occurs, so that repeated checking is avoided.

In DBScan, an object o is *directly density-reachable* from an object o' if o is within o''s e-neighborhood, and o' is surrounded by sufficiently many objects. Next, o is called *density-reachable* from o' if there is a sequence of objects $o_1, o_2, \ldots, o_n$ with $o_1 = o'$ and $o_n = o$ such that o_{i+1} is directly density-reachable from o_i. Then, an object o is *density-connected* to an object o' w.r.t. e and m if there is an object o'' such that both o and o' are density-reachable from o'' w.r.t. e and m.

Since a group is required by definition to have at least m objects, and at least one object in the group is a core object by the definition of density-connectedness, core objects should have at least $(m-1)$ neighbors within their

e-neighborhood, whereas border objects have fewer neighbor objects.

The removal of a border object o from a cluster C causes C to expire if only $(m-1)$ members remain. The removal of a core object o from C is more complicated. The neighbors of o may no longer be connected after the removal of o. Thus, we have to check 1) the possibility to remove o's neighbors and 2) the cardinality of C to determine whether or not C expires.

Two core objects may travel apart from each other, causing the cluster to split. The expected time of such each can also be calculated in a similar way.

3.1.4 Object Exit Time and Join

Once a cluster has formed, the getExitTime procedure computes the exit time for each border object in the cluster. It starts by identifying the boarder objects and the core objects. Next, it models the movement of an object o in $\mathbb{R}^2$ as a linear function of time: $o = (x^{t_{ref}}, y^{t_{ref}}, v_x, v_y)$, where $(x^{t_{ref}}, y^{t_{ref}})$ is the position of o at reference time t_{ref} and (v_x, v_y) is its latest reported velocity. Then, it computes the exit time for a boarder object by calculating the time when the distance between boarder object and its core object is e [4].

After a boarder object exits its cluster, its core object has one less neighbor. When the core object has less than $m-1$ neighbors, this core object becomes a boarder object.

We follow a passive strategy for detecting join events, which is carried out by the Insert procedure at the end of each iteration.

3.1.5 Distance Bounds

With line simplification, in order not to miss clusters, a relationship has to be established between the simplified and original trajectories. In the Insert procedures using RangeQuery, the following lemma is applied.

Lemma 3.1. *Let tr_i, tr_j be the trajectories of o_i and let o_j, and let tr'_i, tr'_j be their trajectories after trajectory simplification with the sliding window algorithm. Let $tr_i(t)$ be the position of o_i at time t. Then, $D(tr'_i(t), tr'_j(t)) > 2\delta + e \Rightarrow D(tr_i(t), tr_j(t)) > e$.*

Proof. To prove the lemma by contradiction, assume the inequality $D(tr_i(t), tr_j(t)) \leq e$ holds. After simplification, $tr'_i(t)$ is either the position of a sampling point or a position on a line segment. By definition of the sliding window algorithm, in the first case, $tr'_i(t) = tr_i(t)$, and in the second case, $D(tr'_i(t), tr_i(t)) \leq \delta$. In both cases, $D(tr'_i(t), tr_i(t)) \leq \delta$. Similarly, we have $D(tr'_j(t), tr_j(t)) \leq \delta$. According to the triangle inequality, we have $D(tr'_i(t), tr'_j(t)) \leq D(tr'_i(t), tr_i(t)) + D(tr'_j(t), tr_j(t)) + D(tr_i(t), tr_j(t)) \leq 2\delta + e$. This contradicts the assumption that $D(tr'_i(t), tr'_j(t)) > 2\delta + e$. □

According to Lemma 3.1, the range search distance $2\delta + e$ used in the GroupDiscovery framework is safe.

3.2 A Running Example

Fig. 3 shows the trajectories of six moving objects, $o_1, \ldots, o_6$ being sampled at every time point from time t_0 to time t_9. It is assumed that the objects move beyond the 10 time units

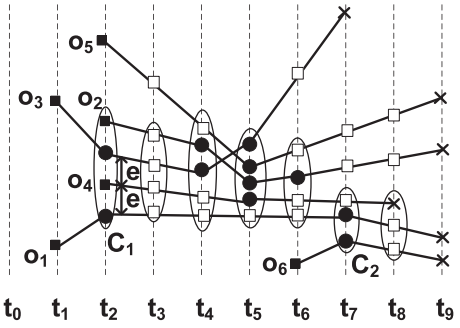


Fig. 3. Trajectories of six moving objects.

shown. We use $o_1, \ldots, o_6$ as the identifiers of the objects. APPEAR events are solid squares, UPDATE events are solid dots, and DISAPPEAR events are crosses. In contrast, hollow and square points mean that there are no primary events. For instance, object o_4 has an APPEAR event, an UPDATE event, and a DISAPPEAR event.

Let the group query parameters m (cardinality) and τ (duration) each be 3. Parameter e (clustering distance threshold) is as shown in the figure, and parameter k is not needed. FindContinuousCluster runs as follows:

As the trajectory data stream in, at t_1, events $(t_1, o_1, \perp, \perp, \text{APPEAR})$ and $(t_1, o_3, \perp, \perp, \text{APPEAR})$ are handled, resulting in $U = \{o_1, o_3\}$ (unclassified objects).

At time t_2, five events are handled. Objects o_2, o_4, and o_5 appear and are added to set U, and objects o_1 and o_3 update their velocities. Then, Insert finds a cluster $C_1 = \{o_1, o_2, o_3, o_4\}$, where o_3 and o_4 are core objects, and o_1 and o_2 are border objects. Suppose both o_1 and o_2 are calculated to exit at t_{10}. Then, exit events for border objects are inserted into *eventQ*: $(t_{10}, o_1, C_1, \perp, \text{EXIT})$ and $(t_{10}, o_2, C_1, \perp, \text{EXIT})$. After this iteration, $U = \{o_5\}$.

No events occur at time t_3. At t_4, o_2 and o_3 update their velocities. A join event for o_5 is detected and handled. The result is that $C_1 = \{o_1, o_2, o_3, o_4, o_5\}$, where o_2 is promoted to a core object. Thus, the exit event for o_2 in $eventQ$ is deleted. Finally, an exit event is created for o_5, $(t_7, o_5, C_1, \perp, \text{EXIT})$, as o_5 is a border object.

At t_5, objects o_2, o_3, o_4, and o_5 update their velocities. It is detected that o_3 becomes a border object and that o_5 becomes a core object. The exit event of o_5 in *eventQ* is deleted. A new exit event $(t_6, o_3, C_1, \perp, \text{EXIT})$ is created and inserted into *eventQ*.

At t_6, o_6 appears and is added to U. Core object o_2 updates its velocity, causing border objects to update their expected exit time and to update the $eventQ$. Object o_3 exits C_1, and o_5 becomes a border object. An exit event is created for o_5: $(t_7, o_5, C_1, \perp, \text{EXIT})$. Now, $U = \{o_3, o_6\}$ and $C_1 = \{o_1, o_2, o_4, o_5\}$.

At time t_7, o_3 disappears and o_1 updates its velocity. After the exit of o_5, o_2 downgrades to a border object, triggering the expiry of C_1 because no core objects exist. An expire event $(t_7, o_5, C_1, \perp, \text{EXPIRE})$ is created and inserted into $eventQ$. It is then immediately handled yielding $U = \{o_1, o_2, o_4, o_5, o_6\}$. Insert then finds a new cluster $C_2 = \{o_1, o_4, o_6\}$. New exit events are created for border objects o_4 and o_6. Now, $U = \{o_2, o_5\}$

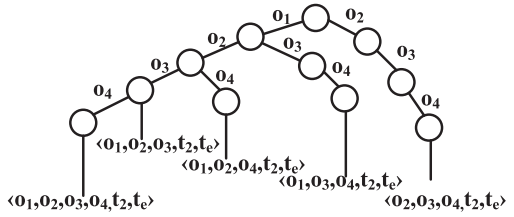
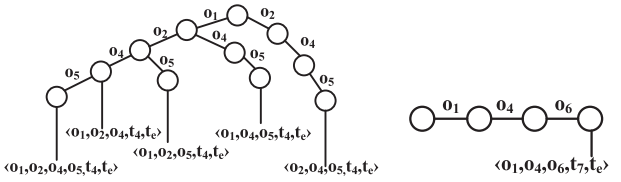


Fig. 4. Trie, for example, cluster C_1 at time t_2.



(a) After o_3 Exits (b) After o_2 and o_5 Exit

Fig. 6. Tries after removals.

At time t_8, o_4 disappears, resulting in the expiry of C_2 because $|C_2| < m$. Now $U = \{o_1, o_2, o_5, o_6\}$.

3.3 History Handler Module

The GroupDiscovery framework needs to deal with potentially large numbers of groups. In our first approach, the history handler module manages the groups using a trie. However, we observe that not all subgroups need to be materialized. Based on this, the history handler module can employ a hash structure that maps clusterIDs to lists of cluster statuses (called *cluster histories*). We call the first method *GroupDiscovery (GD)* and the improved method *GroupDiscoveryPlus(GD+)*.

3.3.1 Group Discovery

In this approach, the history handler module employs a modified trie to represent subgroups during group discovery. A trie, or prefix tree, is an ordered tree data structure, where each vertex represents a string (a word or a prefix). The descendants of a vertex have the string associated with that node as a common prefix; the root is associated with the empty string.

We represent a subgroup as the string that contains the identifiers of its member objects in sorted order. Each path in the trie structure represents a subgroup, and each leaf contains the subgroup members, and the subgroup's start and end time for computing the subgroup's score.

In our running example in Section 3.2, cluster $C_1 = \{o_1, o_2, o_3, o_4\}$ is formed at time t_2. Its subgroups are shown in Fig. 4 with $m = 3$. The starting time of all subgroups is t_2. The ending time (t_e) is undefined initially, but is set to the time when a cluster expires.

When an object joins a cluster, new subgroups containing the object are created, and new paths are inserted into the trie. In our running example, object o_5 joins C_1 at t_4. The trie of the resulting cluster is presented in Fig. 5. The starting time of each new path with object o_5 is set to t_4.

When a border object exits a cluster, every path in the trie representing a subgroup that contains this object should be removed. Before the removal, the algorithm follows these paths, retrieve the leaf nodes, and sets the end times (t_e) in these leaf nodes to be current time. These leaf nodes represent subgroups that are then scored and returned. In our running example, when o_3 (a border object) exits C_1 at time t_6, the resulting trie structure is presented in Fig. 6a.

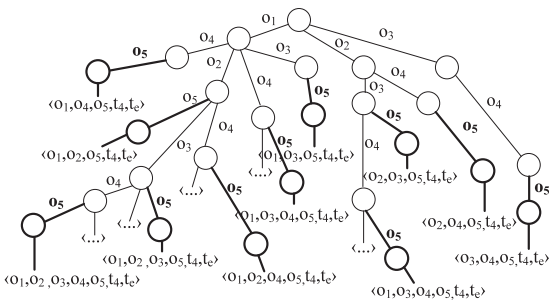


Fig. 5. Trie after insertion of o_5.

If an exiting object is a core object, its exit can cause other objects to also exit the cluster. Then, every path containing any exit object is removed from the trie. Suppose object o_2 is a core object of a cluster $C_1 = \{o_1, o_2, o_4, o_5, o_6\}$, which starts from time t_7. Its exiting causes the exit of o_5. The resulting cluster is then $\{o_1, o_4, o_6\}$. The corresponding trie is shown in Fig. 6b.

When two clusters are merged, it is straightforward to update the trie with new paths. No paths are removed because objects in existing subgroups still travel together. However, updating the trie when a cluster expires is complicated because some objects from the old cluster may form a new cluster after reclustering.

Suppose $C_1 = \{o_1, o_2, o_3, o_4\}$ expires and that $C_2 = \{o_1, o_3, o_4, o_7\}$ and $C_3 = \{o_2, o_5, o_6\}$ result from reclustering. Although o_2 is in a new cluster, o_1, o_3, and o_4 still travel together. This fact can be found by intersecting C_1 and C_2. In general, when a cluster expires, all its subgroups are enumerated. It is then checked for each subgroup whether its members belong to the same cluster after reclustering. If not, the path representing the subgroup is removed. After removing a path, its leaf node is then checked by the procedure CheckCandidate (Algorithm 6).

3.3.2 Group Discovery Plus

In the naive approach, many subgroups are materialized and inserted into the trie. Although search for subgroups is efficient, there are still an exponential number of subgroups. Thus, the trie becomes a bottleneck for large clusters.

The following simple, yet important, observation contributes to solving the exponential subgroup problem.

Lemma 3.2. *A subgroup that forms no earlier and expires no later than does a superset group is dominated by that superset group.*

Proof. Let C_2 be a subgroup of C_1 with a lifetime contained in that of C_1. Then, $|C_1| \geq |C_2|$, $C_1.t_s \leq C_2.t_s$, and $C_2.t_e \leq C_1.t_e$. Thus, C_1 dominates C_2. □

Thus, only subgroups that are formed earlier or expire later than their superset clusters need to be considered. The GroupDiscoveryPlus algorithm only maintains such subgroups.

GD+ uses a hash table H that maps cluster identifiers to the history statuses of the clusters. A history status is similar to a leaf in the trie and contains the object identifiers of the members in the cluster together with a time interval.

Update occurs only to the most recent information, called the *active status*. Once an event happens the active status is updated, and the score is computed. Then, a new active status is appended to the list. In case a cluster expires, GD+ goes through the history of the cluster and generates each possible candidate.

In our running example, we have a cluster $C_1 = \{o_1, o_2, o_3, o_4\}$ at t_2, and $m = 3$. GD+ inserts $\langle C_1, \langle\{o_1, o_2, o_3, o_4\}, t_2, nil\rangle\rangle$ into H. At t_4, o_5 joins C_1, which becomes $C_1 = \{o_1, o_2, o_3, o_4, o_5\}$. GD+ then sets the end time in the active entry to t_3 and appends a new entry $\langle\{o_1, o_2, o_3, o_4, o_5\}, t_4, nil\rangle$ to the history for C_1 in H.

At t_6, o_3 leaves C_1. At t_7, the exits of o_2 and o_5 cause C_1 to expire. Then, $C_2 = \{o_1, o_4, o_6\}$ forms. At t_9, cluster C_2 also expires. At this time, the entries in H are $\{\langle C_1, \langle\{o_1, o_2, o_3, o_4\}, t_2, t_3\rangle, \langle\{o_1, o_2, o_3, o_4, o_5\}, t_4, t_5\rangle, \langle\{o_1, o_2, o_4, o_5\}, t_6, t_6\rangle\rangle, \langle C_2, \langle\{o_1, o_4, o_6\}, t_7, t_8\rangle\rangle\}$

GD+ accesses the entry for each *cid* in reverse order and finds

1. that o_1, o_2, and o_4 travel together from t_2 to t_6,
2. that o_1, o_2, o_3, and o_4 travel together from t_2 to t_5,
3. that o_1, o_2, o_3, o_4, and o_5 travel together from t_4 to t_5,
4. that o_1, o_2, o_4, and o_5 travel together from t_4 to t_6, and
5. that o_1, o_4, and o_6 travel together from t_7 to t_8.

So the final entries in H are $\{\langle C_1, \langle\{o_1, o_2, o_3, o_4\}, t_2, t_5\rangle, \langle\{o_1, o_2, o_3, o_4, o_5\}, t_4, t_5\rangle, \langle\{o_1, o_2, o_4, o_5\}, t_4, t_6\rangle, \langle\{o_1, o_2, o_4\}, t_2, t_6\rangle\rangle \langle C_2, \langle\{o_1, o_4, o_6\}, t_7, t_8\rangle\rangle\}$.

The RevHist algorithm that manages the history statuses of clusters is shown in Algorithm 5. Recall that the history statuses of a cluster is a list of items with members and their starting and ending times.

Algorithm 5: RevHist(H)

```
1  for i ← H.size() − 1; i ≥ 0; i − − do
2      curr ← H.get(i);
3      for j ← i − 1; j ≥ 0; j − − do
4          prev ← H.get(j);
5          if curr.t_s = prev.t_e + 1 then
6              mem ← curr ∩ prev;
7              if mem = curr then curr.t_s ← prev.t_s;
8              else if mem = prev then
                   prev.t_e ← curr.t_e;
9              else if mem.size() ≥ m then
10                 create a new entry e with members
                   mem;
11                 e.t_s ← prev.t_s;
12                 e.t_s ← curr.t_e;
13                 curr ← e;
14                 H.add(e);
```

The procedure starts from the last item (*curr*) of the list, and compares the previous items (*prev*) in reverse order. If *curr* is a subset of *prev*, the current subgroup actually travels together from the previous item's start time. If *curr* is a superset of *prev*, the previous subgroup travels together until the current item's end time. Lastly, if the intersection between *curr* and *prev* has at least m objects, a new entry is created with the previous item's start time and the current item's end time. This new entry is then inserted into the list.

3.4 Returning Meaningful Results

The GD and GD+ algorithms may return groups that are similar to each other. To improve the quality of results, we propose a similarity measure for groups.

Definition 5. *We define the similarity of two groups C_1 and C_2 as*

$$\mathrm{Sim}(C_1, C_2) = \frac{\|C_1 \cap C_2\|}{\|C_1 \cup C_2\|}.$$

We then require that result groups are more diverse than a given threshold θ. The following statement should then be true. $\forall C_1, C_2 \in R(C_1 \neq C_2 \Rightarrow (\mathrm{Sim}(C_1, C_2) \leq \theta))$. The similarity checking is carried out by the Algorithm 6.

Algorithm 6: CheckCandidate(S, R, θ, k)

```
1  foreach C ∈ S do
2      if C.t_e − C.t_s ≥ τ then
3          dominated ← false;
4          updated ← false; C_sim ← null;
5          foreach C′ ∈ R do
6              if C′ ≺ C then dominated ← true; break;
7              else if C ≺ C′ then R.delete(C′);
8              if Sim(C′, C) > θ then C_sim ← C′ ;
9          if ¬ dominated then
10             if C_sim = null then
11                 R.add(C);
12                 updated ← true;
13             else if Score(C) > Score(C_sim) then
14                 R.delete(C_sim);
15                 R.add(C);
16                 updated ← true;
17         if updated then
18             if R.size() > k then
19                 R ← R.topK();
20             update R.minScore;
```

The CheckCandidate procedure takes a candidate group collection S, the top-k set R, a threshold θ, and k as parameters. The groups with top-k scores are to be inserted into R, and $R.minScore = \min_{C \in R} C.score$, i.e., the smallest top-k score. The procedure iterates over every candidate group C, and validates that the duration of C exceeds τ. The variable *dominated* captures whether C is dominated by some entry in R. The variable *updated* captures whether R has been updated. The variable C_{sim} refers to a similar entry in R, if any, and is set to null initially. For each entry C' in the candidate list R, if C' dominates C, then dominated is set to true. C cannot be in the result and the program terminates. Otherwise, if C dominates C', C' is removed from R, as C' cannot be in the result. If C' and C are similar, then C_{sim} refers to C'. If C is neither dominated by nor similar to any entry, C is inserted into R, and updated is set to true. Otherwise, the algorithm includes into R the group with the higher score between C and C_{sim}, and it excludes the other group. Updated is set to true if R is updated. In the end, if R is updated and its size exceeds k, the algorithm picks the top-k groups in R. $R.minScore$ is updated accordingly.

3.5 Avoiding RevHist Calls

Recall that the group query returns k results that score the highest and that a cluster generates many groups or

subgroups during its entire life. We can save actual-score computations (invocations of the RevHist procedure) by computing instead upper bounds on the scores of groups. If the upper bound score for a cluster is smaller than the lowest actual score of the kth candidate group, the groups generated by the cluster cannot be in the result.

Here, we develop an upper bound score of a cluster. Suppose that the candidate list initially maintains k candidates. When a cluster C expires, we compare the upper bound of C with $R.minScore$. The RevHist procedure is invoked only if the upper bound of C exceeds $R.minScore$.

The cardinality of a cluster C can change over time. We calculate the upper bound score of C by using C's maximal cardinality.

Definition 6. *Let N_{max} denote the maximal cardinality of C during its lifetime, the duration of which is ρ_C. The upper bound score of C, $\mathrm{Score}_{UB}(C)$, is defined as*

$$\mathrm{Score}_{UB}(C) = \alpha N_{max} + (1-\alpha)\rho_C.$$

The upper bound score of C is never smaller than the actual score of any group of C.

Lemma 3.3. *Let $\mathrm{Score}_{UB}(C)$ be as defined as above and let C^i be a group generated by C. We have*

$$\mathrm{Score}(C^i) \leq \mathrm{Score}_{UB}(C).$$

Proof. Because C^i is a group of C, we have $|C^i| \leq N_{max}$. The lifetime of C^i also cannot exceed the lifetime of C, so $\rho_{C^i} \leq \rho_C$. Thus, $\mathrm{Score}(C^i) = \alpha N_{C^i} + (1-\alpha)\rho_{C^i} \leq \alpha N_{max} + (1-\alpha)\rho_C = \mathrm{Score}_{UB}(C)$. □

3.6 Complexity Analysis

In this section, we characterize the running time of the GD and GD+ algorithms.

Lemma 3.4. *Let TR be a trajectory collection of N trajectories, let Q be the total number of primary events in TR, and let M be the total number of events that change the status of any cluster, i.e., exit, join, expire, and merge events. The running time of FindContinuousCluster is $O(Q + MNlgN)$.*

Proof. We use a Fibonacci-heap to implement *eventQ*, where an insertion takes $O(1)$ time. We also assume that RangeQuery has time complexity $O(lgN)$. FindContinuousCluster first inserts primary events into *eventQ*, taking $O(Q)$ time. For each event that changes the status of clusters, FindContinuousCluster calls the Insert procedure. In Insert, U and G are mutually exclusive sets that sum up to N. Each object in either U or G calls RangeQuery at most once, so the two loops in Insert take $O(NlgN)$ time. Lastly, the Insert procedure calls DBScan, which has complexity $O(NlgN)$. So the Insert procedure has time complexity $O(NlgN)$. The total complexity of calling Insert for M events is $O(MNlgN)$. The time complexity of FindContinuousCluster is then $O(Q + MNlgN)$. □

Next, we consider the time complexity of handling subgroups.

Lemma 3.5. *Given P clusters, the time complexity for the History Handler module in GD is $O(M\sum_{i=1}^{P} 2^{N_i})$, where $\sum_{i=1}^{P} N_i = N$ and $0 \leq P \leq \frac{N}{m}$.*

Proof. In the History Handler module, every operation needed when the statuses of clusters are updated is performed on the trie structure. If the objects do not form clusters, there is no need to update the trie, so the worst case is that every moving object is a member of a cluster. Let the N moving objects form P clusters ($0 \leq P \leq \frac{N}{m}$) at each event time. The worst case is that these clusters have to be completely rebuilt at each event time. Let the cardinalities of these clusters be $N_1, N_2, \ldots, N_P$ ($\sum_{i=1}^{P} N_i = N$). For a cluster C_i, the number of combinations is

$$\binom{N_i}{m} + \binom{N_i}{m+1} + \cdots + \binom{N_i}{N_i} \leq 2^{N_i},$$

where m is the cardinality threshold. Then, the total number of combinations for all clusters is $\sum_{i=1}^{P} 2^{N_i}$. Again, let M be the number of events that change the status of clusters, the complexity is $O(M\sum_{i=1}^{P} 2^{N_i})$. □

Lemma 3.6. *Given P clusters, the time complexity for the History Handler module in GD+ is $O(PM^2)$.*

Proof. We consider again the worst case where every object is in a cluster and let the N moving objects form P clusters at each event time. The worst case is that for each event time, every cluster has to create a new snapshot at the end of its history list. In total, each cluster has to create M snapshots. Maintaining the data structures of these snapshots takes linear time. The input size for RevHist is P clusters, each with a list of snapshots of size M. For each cluster, every snapshot may have to intersect with the snapshots in front of it, which takes time $O(M^2)$ in the worst case. Therefore, the complexity of going through every cluster is $O(PM^2)$. So, the complexity of the History Handler module in GD+ is $O(PM + PM^2) = O(PM^2)$. □

Theorem 3.7. *The time complexity for GD is $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + M\sum_{i=1}^{P} 2^{N_i})$, whereas the time complexity for GD+ is $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + PM^2)$, where n_i is the number of data points in the ith trajectory.*

Proof. The time complexity of the framework is the total of NOPW and its two functional modules. The time complexity of NOPW is $O(\sum_{i=1}^{N} n_i^2)$. From the lemmas above, Continuous Clustering in either GD or GD+ has complexity $O(Q + MNlgN)$. In GD and GD+, the History Handler Module has complexity $O(M\sum_{i=1}^{P} 2^{N_i})$ and $O(PM^2)$, respectively. Therefore, GD has complexity $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + M\sum_{i=1}^{P} 2^{N_i})$. GD+ has complexity $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + PM^2)$. □

4 Experiments

The experiments were performed on a PC with Intel Xeon (2.66 Ghz) quad-core and 8 GB of main memory running Linux (kernel version 2.6.32). Every instantiation of JVM allocates 2 GB of virtual memory. All the algorithms were implemented in Java, including the Convoy algorithm [17].

(a) Car (b) Truck (c) Starkey

Fig. 7. Visualization of data sets.

TABLE 3
Settings for Experiments

Parameter	Car	Truck	Starkey	GSTD
N	2,305	267	252	5k – – 25k
m	6	10	5	5
τ	3min	3min	5days	10min
e	400m	10m	1,000m	30m
δ	25m	5m	100m	10m
θ	0.5	0.5	0.5	0.5
α	0.5	0.5	0.5	0.5

4.1 Data Sets and Parameter Settings

To evaluate the performance of the proposed framework, we use three real trajectory data sets obtained from vehicles and animals, and one synthetic data set generated by a free space trajectory data generator, the GSTD generator [23].

Fig. 7 shows a subset of trajectories from these data sets.

Car. This data set stems from the INFATI project [15], and has around 1.8 million data points obtained from 20 different vehicles collected during a 4-month period. A new trajectory is created each time the duration between two consecutive data points from the same vehicle exceeds 10 minutes. Also, to increase the probability of finding more objects that travel together, we omit the dates from time stamps and consider only the time of day. After preprocessing, 2,305 separate trajectories were obtained.

Truck. This data set contains 111,930 data points, representing 267 trajectories from 50 trucks moving in the Athens, Greece region.[2] As in Car, the dates are removed. Each trip of a truck is represented as one trajectory.

Starkey. This data set contains 252 trajectories obtained from the Starkey Project,[3] a radio-telemetry study of elk habitats (elk, mule deer, and cattle) in Northeastern Oregon from 1993 to 1996. The collection has 268,216 data points. Each trajectory represents the movement of a particular animal. In this data set, dates were taken into account because each animal was tracked for a long period of time. Also, because the sampling varies substantially across different animals, it was difficult to consistently partition the trajectory of an animal into shorter trajectories.

GSTD. We use the GSTD generator to generate data sets with varying numbers of trajectories. The data points are assumed to be taken from the same day.

We partition Car, Truck, and GSTD into 144 parts, each containing data for 10 min, and we partition Starkey into 146 parts, each containing data for 10 days. In the following experiments, after GD and GD+ process one part, the next part is streamed in as new data, resulting in new events being inserted into *eventQ*. Statistics on the data sets and default settings for key parameters are shown in Table 3.

2. http://www.rtreeportal.org.
3. http://www.fs.fed.us/pnw/starkey/data/tables/.

TABLE 4
Simplification

Tolerance	Car	Starkey
0	1,747,757	268,216
25	109,907	214,277
50	98,460	179,059
100	95,476	135,211
150	95,239	107,885
200	95,202	90,523

Tolerance	Truck
0	111,930
5	76,557
10	64,677
15	57,157
20	51,881
25	47,766

TABLE 5
Synthetic Data Set

Num Traj.	Num Data Points	After Simpl.
5k	210,130	186,265
10k	420,296	382,187
15k	630,268	697,271
20k	840,813	812,276
25k	1,050,876	972,138

We report on both efficiency and effectiveness studies of our proposed framework, and we compare with the Convoy algorithm [17]. In the studies of the effects of polyline simplification, we use the existing online simplification technique, Normal Opening Window [22]. For Car and Truck, the reduction rate is high even for small tolerances. Table 4 shows the number of remaining data points when the tolerance δ ranges from 25 to 200 in Car and Starkey and from 5 to 25 in Truck. The numbers of data points in GSTD (both original and after simplification) are summarized in Table 5.

4.2 Effects of Varying m, e, and τ

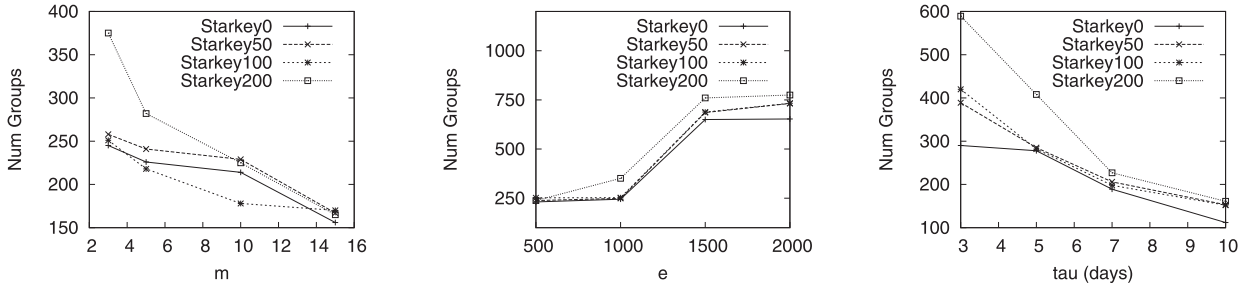
We proceed to consider the number of groups returned when varying cardinality m, distance threshold e, and lifetime τ. We use Starkey with tolerance values being 0 (original), 50, 100, and 200 meters. Starkey0, thus, serves as the ground truth. Fig. 8 shows the result.

As the cardinality threshold m increases, the number of discovered groups decreases quickly with all simplification tolerances because more animals are needed to form clusters. For Starkey0 and Starkey50, very few groups are discovered when $m > 10$, whereas for Starkey100 and Starkey200, a marked decrease is observed even before $m = 10$. It is also observed that Starkey50 displays the pattern that is most similar to Starkey0, whereas Starkey200 deviates the most from Starkey0. The four collections behave similarly when varying distance threshold e—as e increases, groups are allowed to be less dense, and more groups are found.

Finally, all collections exhibit a decreasing trend as parameter τ is increased.

4.3 Comparing GD and GD+

To observe the practical implication of the observation in Lemma 3.2, which is exploited in GD+, we ran GD and GD+ on Car and counted the number of candidate groups. Fig. 9 shows that the running time of GD is around 10 times that of GD+. Fig. 10 shows that the number of checked candidates by GD+ is reduced by up to three orders of magnitude. Note that both figures have a log-scaled y-axis.



(a) Varying m (b) Varying e (c) Varying τ

Fig. 8. Effect of varying m, e, and τ on groups identified.

We were only able to run GD on Car as GD is rather inefficient because of frequent trie updates when the memberships of clusters changes. On the other data sets, GD exhausted the Java heap space. In conclusion, Lemma 3.2 greatly improves the performance of the GroupDiscovery framework.

4.4 Effect of Varying θ

Fig. 11a (y-axis logscaled) shows the effect of θ on the number of pruned candidate groups. It is observed that with the increase of the θ value, fewer candidate groups are pruned. When θ is small, more candidate groups having common objects are treated as similar groups and are thus discarded by CheckCandidate. Among the three data sets, Car has the least pruned similar candidate groups, whereas Starkey has the most pruned similar candidate groups. Recall that for Car and Truck, we disregard the dates to increase the number of similar trajectories to find groups, while in Starkey, we take dates into account when tracking the same groups for three years. Thus, many more similar groups are found in Starkey due to recurring patterns. In Truck, the routes that the vehicles follow are more fixed than in Car, so more similar groups are found in Truck. Note that in both Truck and Starkey, many similar candidate groups are found along the way even when θ is large (0.8). Also note that no candidate groups are pruned when $\theta = 1.0$ for the three data sets.

Fig. 11b shows that for each data set, the average similarity among the top-k groups actually increases with the increase of θ, though the difference between any two average similarity values is not significant in the same data set. It is also observed that the average similarity value depends highly on the data set.

There is a tradeoff between the total score of the groups in the top-k list and the average similarity. To have more diversity in the top-k list, we need to decrease the θ value, which may result in pruning some high-scoring candidates in the top-k list because they are similar to other group candidates in the top-k list. Fig. 11c shows that with the increase of θ, the total score ratio (normalized by total value when $\theta = 1$) increases, and the average similarity among the top-k also increases, i.e., the diversity decreases.

4.5 Effect of Varying α

We conduct experiments on the effect of α on the average cardinality and duration of the results, which can be seen in Fig. 12. The average duration of the result drops greatly when α increases from 0 to 0.2, and then decreases slowly. In contrast, the average cardinality of the result increases slowly from 5.53 to 6.28 with the increase of α.

Recall the score definition in Section 2.1. Generally, small α values favor groups of longer durations, whereas large α values favor groups of larger sizes.

4.6 Effect of Varying k on Runtime

Fig. 13a shows the effect on the runtime of GD+ of varying k when using Starkey. The figure shows that when k is smaller than 15, the running time is relatively low. But when k exceeds 15, the running time increases markedly. The reason is that many candidates have already been pruned based on the upper bound on group scores and $R.minScore$. Thus, the use of pruning based on the lower bound on group scores is effective, and GD+ is able to exploit small k values to obtain better performance.

4.7 Comparing Top-k Results

We next consider the effect of simplification on the top-k results produced on Starkey. We compare the results obtained from the original and from default simplified data sets. We apply the Precision@k ranking performance metric

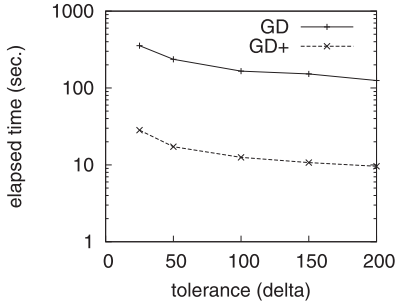


Fig. 9. Running time versus tolerance.

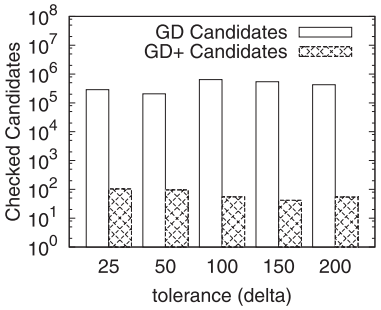
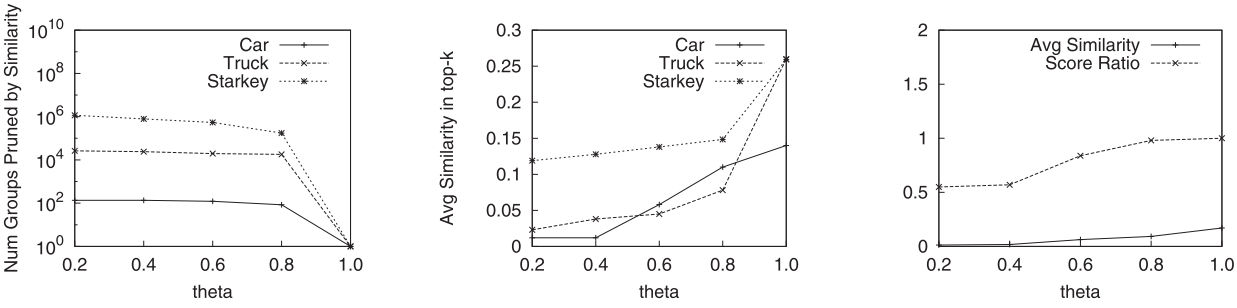


Fig. 10. Checked candidates versus tolerance.



(a) Pruned by Similarity (b) Avg. Similarity in top-k (c) Avg Similarity vs. Total Score

Fig. 11. Effect of varying θ.

that measures the fraction of the number of groups that are found in both original and simplified data sets versus total number of found groups (k). The metric Precision@k is widely used in information retrieval.

We vary the cardinality threshold m from 5 to 15 (Fig. 13b) and the distance threshold e from 500 to 2,000 meters (Fig. 13c), while using the simplification tolerance values 50, 100, and 200. When varying m, the precision ranges from 61 to 89 percent. The results are affected only little by simplification for $m = 7$ and $m = 10$. The highest drop in precision occurs for $m = 15$ and $\delta = 200$. Next, when varying e, the precision for different simplification tolerances ranges from 35 to 95 percent. The highest precisions are obtained when $\delta = 50$ and $e = 2{,}000$.

It is also observed that varying the distance threshold affects the results more than varying the cardinality threshold. The results from varying the distance threshold are also less stable.

4.8 Comparing GD+ and Convoy

We proceed to consider the runtime performance of the GD+ algorithm when varying the polyline simplification δ, comparing with the Convoy algorithm [17]. To be fair, we run both algorithms in an offline setting, because Convoy can only be applied offline. The Douglas-Peucker polyline simplification algorithm [9] is used to simplify the trajectories for both algorithms. The results are shown in Fig. 14.

As δ increases, fewer data points remain in the trajectories, which in turn improves the runtimes of both algorithms. The GroupDiscovery framework achieves speedups because fewer events need to be handled. The situation for the Convoy algorithm is more complex. This algorithm has a filter step that applies DBScan to line segments, and it has a refinement step that applies DBScan to data points. In both steps, with fewer sampling points, a line segment may have more time-overlapping line segments for each clustering, but the total number of clustering performed is reduced.

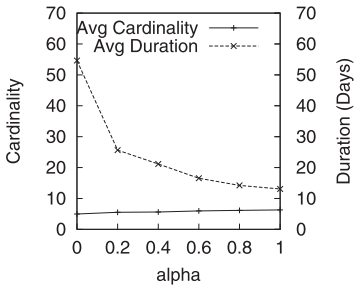


Fig. 12. Average cardinality and duration versus α.

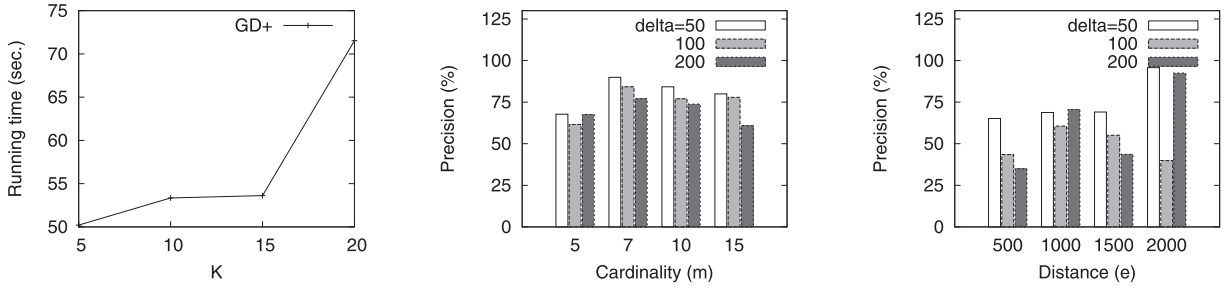
On Car and Starkey, GD+ outperforms Convoy for all tolerances. However, on Truck, GD+ only runs faster than Convoy when the tolerance is high. When applied to Truck, GD+ finds much larger numbers of groups than does Convoy for small tolerance values, as shown in Fig. 15. When the tolerance value is high, although GD+ still finds more groups, the filter step in Convoy takes more time because of larger numbers of candidates, so GD+ outperforms Convoy.

We also studied the effect of simplification on the number of discovered groups in both GD+ and Convoy. We first ran GD+ with $\delta = 0$ on each data set to obtain the total numbers of groups, which is treated as the *base-line*. Then, we applied simplification with various δ values and ran both GD+ and Convoy on the simplified data sets. For each tolerance value, the groups obtained by the two algorithms are compared with each other and with the baseline. It is observed that the groups returned by Convoy were consistently subsets of those returned by GD+. Fig. 15 shows the ratio of the number of groups found with different tolerance values with the baseline.

It is seen that more groups are found than with Convoy. As discussed before, GD+ is able to find patterns that are missing by Convoy because of sampling independence. The reduction in the number of sample points due to simplification affects both Convoy and GD+. We observe that GD+ consistently finds increasing numbers of groups for all data sets as the tolerance is increased. The number of convoys found also increases with the tolerance for Truck and Starkey. The main reason is that the smoothing effect of trajectory simplification results in clusters of longer duration. However, the number of convoys in Car decreases for small tolerance values. This is mainly because with small tolerance values, the number of sampled points decreases significantly, which affects the Convoy algorithm very much. For both Car and Truck, the numbers of groups and convoys found are very sensitive to simplification because cars and trucks are constrained by a road network, and the smoothing effect of simplification is significant.

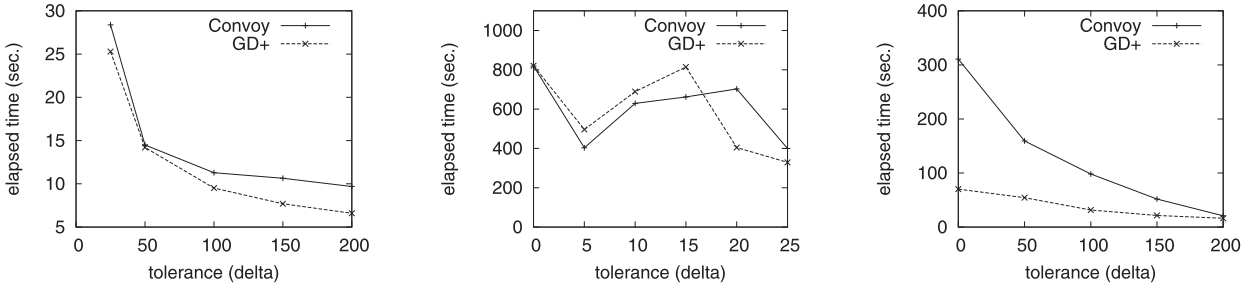
Next, we compare GD+ and Convoy on the synthetic data sets. Fig. 16 shows the result, when the number of trajectories varies from 5k to 25k. It is observed that GD+ consistently outperforms Convoy. The performance gap between GD+ and Convoy increases with the number of trajectories.

The filtering step in Convoy relies on time-domain partitioning and segment clustering. With a larger number



(a) Effect of Varying k on Runtime (b) Precision@10, Varying m (c) Precision@10, Varying e

Fig. 13. Top-k results.



(a) Car (b) Truck (c) Starkey

Fig. 14. Effect of simplification tolerance on efficiency.

of trajectories, it is more likely to have many trajectories near each other during certain time periods. Thus, the filtering step in Convoy is less effective, resulting in more candidates to be checked by the time-consuming refining step. In contrary, thanks to the event-driven design, GD+ always processes events when necessary. So it is less affected by the increased number of trajectories.

5 Conclusion and Future Work

In this paper, we have proposed the concept of *trajectory group* that is different from related notions in previous works and that enables the prioritized discovery of interesting moving object clusters from trajectories. We have also proposed the first framework that satisfies the four requirements explained in Section 1. Techniques based on the continuous clustering of moving objects using an effective pruning strategy are proposed to efficiently discover such groups. A scoring function enables the ranking of the discovered groups according to their size and duration. The effectiveness and efficiency of our schemes are studied using real data sets.

To reduce the number of events in the Continuous Cluster module and, thus, to further improve performance, trajectories should be as smooth as possible. To this end, future work can employ shared prediction-based tracking [8], [24] when collecting locations points.

Tries are used widely due to their simplicity and efficiency. Although the computational overhead is substantial, using a trie for storing and representing clusters is promising, and future work that aims to reduce the computational overhead is in order.

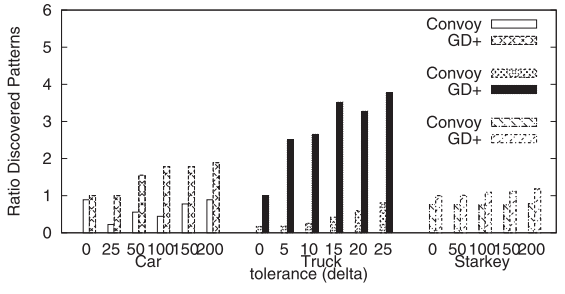


Fig. 15. Effect of simplification tolerance on error.

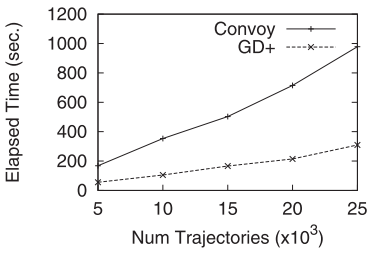


Fig. 16. Effect of number trajectories.

Acknowledgments

Xiaohui Li and Kian-Lee Tan would like to acknowledge the support of NExT Research Center funded by MDA, Singapore, under the research grant: WBS:R-252-300-001-490. Vaida Čeikutė and Christian S. Jensen were supported in part by the Geocrowd Initial Training Network funded by the European Commission as an FP7 - People Marie Curie Action.

References

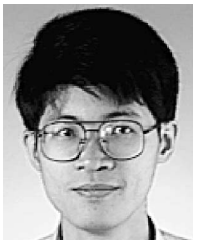
[1] G. Al-Naymat, S. Chawla, and J. Gudmundsson, "Dimensionality Reduction for Long Duration and Complex Spatio-Temporal Queries," *Proc. ACM Symp. Applied Computing (SAC)*, pp. 393-397, 2007.

[2] H.H. Aung and K.-L. Tan, "Discovery of Evolving Convoys," *Proc. 22nd Int'l Conf. Scientific and Statistical Database Management (SSDBM)*, pp. 196-213, 2010.
[3] H.H. Aung and K.-L. Tan, "Finding Closed Memos," *Proc. 23rd Int'l Conf. Scientific and Statistical Database Management (SSDBM)*, pp. 369-386, 369.
[4] R. Benetis, C.S. Jensen, G. Karčiauskas, and S. Šaltenis, "Nearest and Reverse Nearest Neighbor Queries for Moving Objects," *The VLDB J.*, vol. 15, no. 3, pp. 229-249, 2006.
[5] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle, "Reporting Flock Patterns," *Computational Geometry Theory Applications*, vol. 41, pp. 111-125, Nov. 2008.
[6] L. Chen and R. Ng, "On the Marriage of Lp-Norms and Edit Distance," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB)*, pp. 792-803, 2004.
[7] L. Chen, M.T. Özsu, and V. Oria, "Robust and Fast Similarity Search for Moving Object Trajectories," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05)*, pp. 491-502, 2005.
[8] A. Civilis, C.S. Jensen, and S. Pakalnis, "Techniques for Efficient Road-Network-Based Tracking of Moving Objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 5, pp. 698-712, May 2005.
[9] D. Douglas and T. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Line or Its Character," *The Am. Cartographer*, vol. 10, no. 42, pp. 112-122, 1973.
[10] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," *Proc. Second Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, pp. 226-231, 1996.
[11] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. 20th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS)*, pp. 102-113, 2001.
[12] J. Gudmundsson and M. van Kreveld, "Computing Longest Duration Flocks in Trajectory Data," *Proc. 14th Ann. ACM Int'l Symp. Advances in Geographic Information Systems (GIS)*, pp. 35-42, 2006.
[13] J. Gudmundsson, M. van Kreveld, and B. Speckmann, "Efficient Detection of Motion Patterns in Spatio-Temporal Data Sets," *Proc. 12th Ann. ACM Int'l Workshop Geographic Information Systems (GIS)*, pp. 250-257, 2004.
[14] Y. Huang, C. Chen, and P. Dong, "Modeling Herds and Their Evolvements from Trajectory Data," *Proc. Fifth Int'l Conf. Geographic Information Science (GIScience)*, pp. 90-105, 2008.
[15] C.S. Jensen, L.H., S. Pakalnis, and J. Runge, "The INFATI Data," Aalborg Univ., TimeCenter TR-79, 2004.
[16] C.S. Jensen, D. Lin, and B.C. Ooi, "Continuous Clustering of Moving Objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 19, no. 9, pp. 1161-1174, Sept. 2007.
[17] H. Jeung, M.L. Yiu, X. Zhou, C.S. Jensen, and H.T. Shen, "Discovery of Convoys in Trajectory Databases," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1068-1080, 2008.
[18] P. Kalnis, N. Mamoulis, and S. Bakiras, "On Discovering Moving Clusters in Spatio-Temporal Data," *Proc. Ninth Int'l Conf. Advances in Spatial and Temporal Databases (SSTD)*, pp. 364-381, 2005.
[19] P. Laube and S. Imfeld, "Analyzing Relative Motion within Groups of Trackable Moving Point Objects," *Proc. Second Int'l Conf. Geographic Information Science (GIS)*, pp. 132-144, 2002.
[20] J.-G. Lee, J. Han, and K.-Y. Whang, "Trajectory Clustering: A Partition-and-Group Framework," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 593-604, 2007.
[21] Z. Li, B. Ding, J. Han, and R. Kays, "Swarm: Mining Relaxed Temporal Moving Object Clusters," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 723-734, 2010.
[22] N. Meratnia and R.A. de By, "Spatiotemporal Compression Techniques for Moving Point Objects," *Proc. Ninth Int'l Conf. Extending Database Technology (EDBT)*, pp. 765-782, 2004.
[23] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento, "On the Generation of Spatiotemporal Datasets," *Proc. Sixth Int'l Symp. Advances in Spatial Databases (SSD)*, 1999.
[24] D. Tiesyte and C.S. Jensen, "Recovery of Vehicle Trajectories from Tracking Data for Analysis Purposes," *Proc. Sixth European Congress and Exhibition Intelligent Transport Systems and Services*, June 2007.
[25] M.R. Vieira, P. Bakalov, and V.J. Tsotras, "On-Line Discovery of Flock Patterns in Spatio-Temporal Data," *Proc. 17th ACM SIGSPATIAL Int'l Conf. Advances in Geographic Information Systems (GIS)*, pp. 286-295, 2009.
[26] J.S. Vitter, "Random Sampling with a Reservoir," *ACM Trans. Math. Software*, vol. 11, pp. 37-57, Mar. 1985.
[27] M. Vlachos, G. Kollios, and D. Gunopulos, "Discovering Similar Multidimensional Trajectories," *Proc. 18th Int'l Conf. Data Eng. (ICDE)*, pp. 673-684, 2002.
[28] Y. Wang, E.-P. Lim, and S.-Y. Hwang, "Efficient Mining of Group Patterns from User Movement Data," *Data Knowledge Eng.*, vol. 57, pp. 240-282, June 2006.
[29] B.-K. Yi, H.V. Jagadish, and C. Faloutsos, "Efficient Retrieval of Similar Time Sequences under Time Warping," *Proc. 14th Int'l Conf. Data Eng. (ICDE)*, pp. 201-208, 1998.

Xiaohui Li received the bachelor's degree in computer science and the bachelor's degree in statistics from the National University of Singapore in 2008, and is currently working toward the PhD degree in the School of Computing, National University of Singapore. He was a visiting scholar at Aarhus University, Denmark from 2010 to 2011. His current research interest is moving objects and trajectory data management.

Vaida Čeikutė received the MS degree in computer science from Vilnius University, Lithuania, in 2009, and is currently working toward the PhD degree in the Department of Computer Science at Aarhus University, Denmark. Her research interests include trajectory pattern mining, geocontext in location-based services, and intelligent transportation systems.

Christian S. Jensen is a professor of computer science at Aarhus University, Denmark, and he was previously with Aalborg University for two decades. He recently spent a 1-year sabbatical at Google Inc., Mountain View, California. His research concerns data management and data-intensive systems, and it concerns primarily temporal and spatiotemporal data management. He has received several national and international awards for his research. He is a vice chair of ACM SIGMOD and an editor-in-chief of *The VLDB Journal*, and he has served on the editorial boards of the *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, and the *IEEE Data Engineering Bulletin*. He was the PC chair or cochair for SSTD 2001, EDBT 2002, VLDB 2005, MobiDE 2006, MDM 2007, DMSN 2008, TIME 2008, ACM SIGSPATIAL GIS 2011, APWeb 2012, and ICDE 2013. He is a fellow of the ACM and the IEEE, a member of the Royal Danish Academy of Sciences and Letters, the Danish Academy of Technical Sciences, and the EDBT Endowment, and a trustee emeritus of the VLDB Endowment.



Kian-Lee Tan received the PhD degree in computer science in 1994 from National University of Singapore (NUS). He is a professor of computer science in the School of Computing, National University of Singapore. His current research interests include multimedia information retrieval, query processing and optimization in multiprocessor and distributed systems, database performance, and database security and privacy. He has published numerous papers in conferences such as SIGMOD, VLDB, ICDE and EDBT, and journals such as the *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, and *VLDB Journal*. He is a member of the ACM. He is a co-editor-in-chief of *The VLDB Journal* and an associate editor of the *IEEE Transactions on Knowledge and Data Engineering* and the *WWW Journal*. He was a technical program cochair of VLDB '2010, and ICDE '2011.