On Discovery of Gathering Patterns from Trajectories

Kai Zheng[1], Yu Zheng[2], Nicholas Jing Yuan[2], Shuo Shang[3]

[1] School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, Australia, kevinz@itee.uq.edu.au
[2] Microsoft Research Asia, Beijing, China, {yuzheng, nichy}@microsoft.com
[2] Department of Computer Science, Aalborg University, sshang@cs.aau.dk

***Abstract*—The increasing pervasiveness of location-acquisition technologies has enabled collection of huge amount of trajectories for almost any kind of moving objects. Discovering useful patterns from their movement behaviours can convey valuable knowledge to a variety of critical applications. In this light, we propose a novel concept, called gathering, which is a trajectory pattern modelling various group incidents such as celebrations, parades, protests, traffic jams and so on. A key observation is that these incidents typically involve large congregations of individuals, which form durable and stable areas with high density. Since the process of discovering gathering patterns over large-scale trajectory databases can be quite lengthy, we further develop a set of well thought out techniques to improve the performance. These techniques, including effective indexing structures, fast pattern detection algorithms implemented with bit vectors, and incremental algorithms for handling new trajectory arrivals, collectively constitute an efficient solution for this challenging task. Finally, the effectiveness of the proposed concepts and the efficiency of the approaches are validated by extensive experiments based on a real taxicab trajectory dataset.**

I. Introduction

The increasing availability of location-acquisition technologies including telemetry attached on wildlife, GPS set on cars, WLAN networks, and mobile phones carried by people have enabled tracking almost any kind of moving objects, which results in huge volumes of spatio-temporal data in the form of trajectories. Such data provides the opportunity of discovering usable knowledge about movement behaviour, which fosters ranges of novel applications and services [1]. For this reason, it has received great attention to perform data analysis on trajectories. In this paper, we move towards this direction and address one particular challenge to do with discovering the so-called *gathering* patterns from trajectories in an efficient manner.

Informally, a gathering represents a group event or incident that involves congregation of objects (e.g., vehicles, people, animals). Examples of gatherings may include celebrations, parades, large-scale business promotions, protests, traffic jams and other public gatherings. A gathering is expected to imply something unusual or significant happening. As such, the detection of gatherings over trajectories can help in sensing, monitoring and predicating non-trivial group incidents in everyday life.

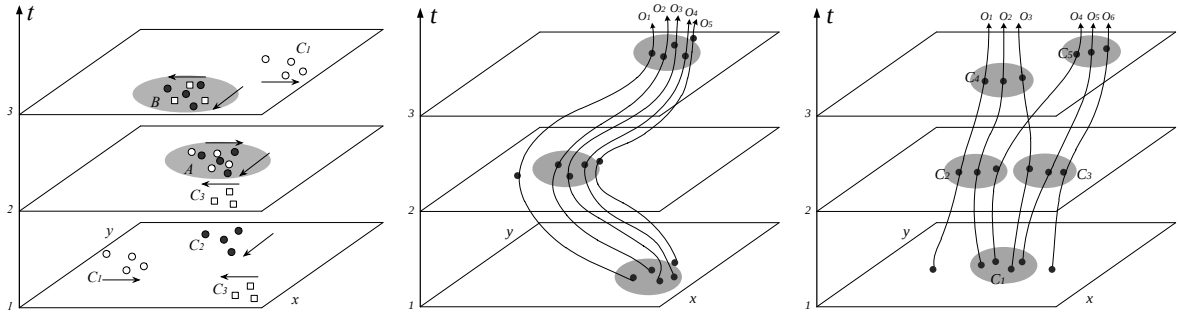
However, discovering the gatherings from trajectories is not an easy task, where challenges are two-fold. First, how to define the concept of gathering appropriately such that it intuitively captures the properties of the above mentioned events, while being rigid from algorithmic aspect in the mean time. Second, how to develop a solution that can discover gatherings from large scale trajectories efficiently, and more importantly, handle new data arrivals in an incremental manner. In the sequel, we will elaborate the two challenges and brief our contributions for addressing them respectively.

A. Challenge 1: Appropriate Model

To get some inspirations on how to choose the appropriate model for gatherings, we first review some related concepts in previous work. The problem of dense area detection or density query[2][3] has been proposed with the objective of identifying *where* and *when* there are regions of high density. But the dense area cannot be adopted to model gatherings due to their limitations in two aspects. First, previous work typically identifies dense areas by overlaying a fixed grid on the geographical space, which might not correspond to the real shape of congregation in a gathering. Although this issue can be tackled to some extent by using a grid with finer granularity, the exponential increase in complexity makes this solution computationally infeasible. Second, a more intrinsic problem lies in that, the only criterion of a dense area is whether its a congregation of individuals exceeds a given threshold, regardless of whether the individuals within the area share common behaviours. Consider Figure 1a in which there are three groups of objects moving towards different directions. At $t = 2$, group c_1 and c_2 encounter and form a dense area A. After that, c_2 departures with c_1 and meets c_3 at $t = 3$, resulting in another dense area B. From this example, we can see that dense areas may be the places where individuals come by each other coincidently (e.g. major road intersections), since it does not take into account the common movements of the objects inside this area.

On the other hand, there also exist some concepts with the aim to discovering a group of objects that move together for a certain time period, such as *flock* [4][5][6][7][8], *con-*

This work was done when the first author was performing an internship in Microsoft Research Asia



(a) Dense area (b) Flock, convoy and swarm (c) Gathering

Fig. 1: Comparison of the related concepts

voy [9][10] and *swarm* [11]. These concepts, which we refer to as *group patterns*, can be distinguished based on how the "group" is defined and whether they require the time period to be consecutive. Specifically, a flock is a group of objects that travel together within a disc of some user-specified size for at least k consecutive timestamps. A major drawback is that a circular shape may not reflect the natural group in reality, which may result in the so-called lossy-flock problem [9]. To avoid rigid restrictions on the sizes and shapes of the group patterns, the convoy is proposed to capture generic trajectory pattern of any shape and extent by employing the density-based clustering. Instead of using a disc, a convoy requires a group of objects to be density-connected to each other during k consecutive time points. While both flock and convoy have strict requirement on consecutive time period, Li et al [11] propose a more general type of trajectory pattern, called swarm, which is a cluster of objects lasting for at least k (possibly non-consecutive) timestamps. Figure 1b illustrates these concepts. Let $k = 2$, the group $\langle o_2, o_3, o_4\rangle$ is a flock from t_1 to t_3. Though o_5 is an obvious company of the group, it cannot be included due to the fixed size of disc employed by the flock definition. But the convoy can include o_5 into the group since $\langle o_2, o_3, o_4, o_5\rangle$ is density-based connected from t_1 to t_3. It is also easy to see that all the five objects form a swarm during the non-consecutive time period $\{t_1, t_3\}$.

However, using the group patterns to model gatherings is also problematic, since they all require the group to contain the same set of individuals during its lifetime. This is unrealistic since in a practical group event such as business promotion, members joining and leaving the event frequently is inevitable. But the critical part is, though each individual may only stay for a short while, they can collectively make the event last for a long time. It is worth mentioning that, there exists another notion of group pattern, called *moving cluster* [12], which does not retain the same set of objects in its lifetime. But they require any two groups in consecutive timestamps to share (large) enough number of common objects, which is still hard to be satisfied by a group event in practice. Besides, two consecutive groups in a moving cluster can be far away from each other, while a gathering usually occurs within a relatively stable area.

Contributions. Given the insights from the above concepts, we regard a gathering as a dense and continuing group of individuals. Besides, the shape and location of the group do not change too fast, since the mobility of individuals in this group is low. Unlike the group patterns, there is no requirement for coherent membership in the gathering, i.e., members can enter and leave this group any time. However, we do desire some *dedicated members* who can commit a certain time period, though may not be consecutive, to participate the group event.

The above observations can be summarized with five key attributes, which should be possessed by the appropriate model of the gathering.

1) **Scale.** A gathering typically involves a relatively large number of individuals.
2) **Density.** Those individuals form a dense group.
3) **Durability.** It should last for a certain time period continuously.
4) **Stationariness.** The geometric properties (e.g., shape, location) of the group is relatively stable.
5) **Commitment.** At any time of the gathering, there exist several dedicated members who stick to the group for a certain time (possibly non-consecutive).

In this paper we first propose a concept called *crowd*, which captures the first four attributes. Specifically, a crowd is a sequence of density-based clusters of objects' locations which lasts for at least k_c timestamps. In order to restrict the geometrical changes of the clusters at consecutive timestamps, we adopt the widely-used Hausdorff distance [13] to measure the distance between two clusters. Then we further define the *gathering* pattern as a special kind of crowd that additionally satisfies the fifth attribute. Formally, each cluster of a gathering should contain at least m_p so-called *participators*, which refer to the objects appearing in at least k_p clusters of this gathering. These concepts can be illustrated with Figure 1c. Let $k_c = 3$, the two sequences of clusters $\langle c_1, c_2, c_4\rangle$ and $\langle c_1, c_3, c_4\rangle$ form two crowds. $\langle c_1, c_2, c_5\rangle$ ($\langle c_1, c_3, c_5\rangle$) is not a crowd since c_5 is too far away from $c_2(c_3)$. Let $k_p = 2, m_p = 3$, then only $\langle c_1, c_2, c_4\rangle$ is a gathering since it contains three participators all the time. We will re-visit this example with more explanations in Section II.

B. Challenge 2: Efficient Discovery Algorithm

Now another question is: can we simply apply or extend the algorithms for group pattern mining to discover the gathering patterns? Apparently the solutions for detecting flocks cannot work since they can only find the group within a fixed disc. The moving cluster algorithm [12] repeatedly appends a cluster of the next timestamp as long as it shares enough common objects with the current cluster. The CuTS algorithm [9] firstly clusters the simplified trajectories to obtain convoy candidates, and then applies the moving cluster algorithm to get the correct results. Both of them do not apply to our problem, since we do not require any two consecutive clusters to share common objects. Last, the *ObjectGrowth* algorithm [11] basically tries to enumerate all subsets of the object set and checks if it is a swarm. To keep the computation complexity tractable, they propose apriori pruning, backward pruning and forward closure checking to reduce the search space significantly. But we cannot borrow these techniques either, since the gathering pattern does not have the *downward closure* property, as will be demonstrated later in Section III.

Naturally, the solution for discovering the gatherings from trajectories can be divided into two phases: finding all the crowds over the trajectory data and validate each crowd to see if it is a gathering. Both of them can raise efficiency issues.

For the crowd discovery phase, one challenge is how to find all the pairs of clusters, the Hausdorff distances between which do not exceed a given threshold, at consecutive time points efficiently. Given the quadratic complexity of Hausdorff distance computation and enormous clusters at each time instant, testing all possible pairs in the brute-force manner will render the discovery process prohibitively time consuming. In addition, we also need to take care of the redundancy problem since many crowds have containment relationship. In such cases, should we output all of them or does it suffice to just keep a subset of them?

As for the second phase, a major issue is what action should be taken whenever a crowd fails to be a gathering. Obviously, it is not wise to validate all the subsequences of the crowd due to the exponential complexity. Therefore, a smarter algorithm that examines only a few subsequences yet guarantees the correct results is desired.

Besides, a practical trajectory database often needs to be updated with new collection of trajectory data periodically. With the growth of the database size, it will become computationally infeasible eventually, if we perform the whole process from scratch whenever new trajectory data arrive. So it is of high importance to devise incremental algorithm that can handle database updates efficiently.

Contributions. To speed up the crowd discovery process, we explore different spatial indexing techniques, namely R-tree and grid index, to organize the clusters at each time point. By this means, a large portion of cluster pairs can be safely pruned and the left candidates can also be refined at lower costs without knowing the exact Hausdorff distances. Besides, with the observation of *downward closure property* of crowds, we propose an efficient growth-style algorithm to produce the *closed crowds* only, i.e., the ones with no super-crowds.

In the gathering detection phase, we propose a *test-and-divide* algorithm, which splits the whole crowds into subsequences by removing the *invalid clusters*, i.e., the ones with not enough participators, and tests each subsequence recursively. Since repetitively counting the occurrences of a large number of objects in a long crowd can still be lengthy, we build *bit vector signature* for each object in the crowd and apply the fast bit operations to count its occurrence. More importantly, the bit vector signatures only need to be constructed once and can be re-used by all the recursive procedures.

At last, corresponding to the database updates, we propose an incremental algorithm that checks only a small subset of crowds or crowd candidates in the old database to see if they can be extended into longer crowds with the new trajectory data. Further, once an existing crowd is extended to a new crowd, we are also able to speed up the test-and-divide algorithm by taking advantage of the old gatherings that have already been found in the crowd.

The remainder of this paper is organized as follows. We define the necessary concepts and formulate the focal problem of this paper in Section II. Efficient solutions for discovering gatherings on archived and new arrivals of trajectory data are presented in Section III. Section IV reports the experimental observations, followed by a brief review of related work in Section V. Section VI concludes the paper.

II. Problem Definition

In this section, we will present the definitions of all necessary concepts used throughout the paper, and formally state the focal problem to be solved. The list of major symbols and notations in this paper is summarized in the following table.

TABLE I: Table of notations

Notation	Definition
$\mathcal{O}_{DB}$	moving object database
$\mathcal{T}_{DB}$	time domain of the database
o	the trajectory of a moving object
t	a time point in $\mathcal{T}_{DB}$
$o(t)$	the location of object o at time t
c_i	a snapshot cluster at time t_i
C_i	the set of snapshot clusters at time t_i
Cr	a crowd
$Cr.\tau$	the lifetime of a crowd
$d_H(P, Q)$	the Hausdorff distance between point sets P and Q
δ	the variation threshold in the definition of crowd
k_c	the lifetime threshold of a crowd
m_c	the support threshold of a crowd
$Par(Cr)$	the participator set of a crowd Cr
k_p	the lifetime threshold of a participator
m_p	the support threshold of a gathering
$B(o)$	the bit vector signature of an object o

Let $\mathcal{O}_{DB} = \{o_1, o_2, ..., o_n\}$ be the set of all moving objects in the database and $\mathcal{T}_{DB} = \{t_1, t_2, ..., t_m\}$ be the time domain, where each t_i is a time point. The *trajectory* of a moving object

o is represented by a polyline that is given as a finite sequence of timestamped locations during a closed time interval $[t_1, t_n]$, i.e., $o = \langle (p_1, t_1), (p_2, t_2), ..., (p_n, t_n) \rangle$, where $p_i \in \Re^2$ is the geo-spatial position sampled at $t_i \in \mathcal{T}_{DB}$. For simplicity, we use $o.\tau$ to denote the lifespan of o and $o(t_i)$ to refer to the location of o at time instant t_i.

In our paper, we consider a practical trajectory database model, which assumes each trajectory may have different lengths and sampling rates (i.e., they are not synchronized in temporal aspect). Therefore, some trajectories may not have a sampled location for a given time instant t_i. In this case, we apply linear interpolation to create the virtual points p_i for those trajectories.

Now we adopt the notion of density-based clustering [14] to define the snapshot cluster. Given a distance threshold ϵ and a set of points S, the ϵ-neighborhood of a point p is defined as $N_\epsilon(p) = \{q \in S | D(p, q) \leq \epsilon\}$, where $D(\cdot)$ is the Euclidean distance between two points. A point p is *directly density-reachable* from a point q w.r.t. a given distance threshold ϵ and an integer m if $p \in N_\epsilon(q)$ and $|N_\epsilon(q)| \geq m$. A point p is called *density-reachable* from q if there is a chain of points $p_1, p_2, ..., p_n$ in S s.t. $p_1 = q, p_n = p$, and p_{i+1} is directly density-reachable from p_i. Then a point p is said to be *density-connected* to a point q if there exists a point $x \in S$ s.t. both p and q are density-reachable from x.

Definition 1 (Snapshot Cluster): Given a trajectory set of moving objects $\mathcal{O}_{DB}$, a distance threshold ϵ, and an integer m, the *snapshot cluster* c_t at timestamp t is a non-empty subset of objects $\mathcal{O} \subseteq \mathcal{O}_{DB}$ satisfying the following conditions:

1) $\forall o_p, o_q \in \mathcal{O}$, $o_p(t)$ is density-connected to $o_q(t)$ w.r.t. ϵ and m.
2) $\mathcal{O}$ is maximal, i.e., if $o_q \in \mathcal{O}$ and $o_p(t)$ is density-reachable from $o_q(t)$ w.r.t. ϵ and m, then also $o_p \in \mathcal{O}$.

A snapshot cluster is a group of objects with arbitrary shape and size, which are density-connected to each other at a given timestamp. Following the notion in DBSCAN [14], such snapshot clusters are spatially maximal so that no two of them with the same timestamp overlap in their objects. In the sequel we will abbreviate the snapshot cluster to cluster and omit the parameters m, ϵ when no ambiguity can be caused.

Definition 2 (Crowd): Given a trajectory set of moving objects $\mathcal{O}_{DB}$, a support threshold m_c, a variation threshold δ, and a lifetime threshold k_c, a *crowd* Cr is a sequence of snapshot clusters at consecutive timestamps, i.e., $Cr = \langle c_{t_a}, c_{t_{a+1}}, ..., c_{t_b} \rangle$, which satisfies the following requirements:

1) The lifetime of Cr, denoted by $Cr.\tau$, is not less than k_c, i.e., $Cr.\tau = b - a + 1 \geq k_c$.
2) There should be at least m_c objects at any time, i.e., $\forall a \leq i \leq b, |c_{t_i}| \geq m_c$.
3) The distance between any consecutive pair of snapshot clusters is not greater than δ, i.e., $Dist(c_{t_i}, c_{t_{i+1}}) \leq \delta, \forall a \leq i \leq b - 1$.

Besides, a subsequence (supersequence) of a crowd Cr is called a sub-(super-)crowd of Cr, if it is also a crowd. Cr is said to be closed if it has no super-crowd.

Since a snapshot cluster is essentially a set of points, we adopt the Hausdorff distance [13] to measure how far two clusters are from each other. Hausdorff distance is a widely used metric for point sets in the community of computer vision and image processing. Given two sets of points P and Q, their Hausdorff distance $d_H(P, Q)$ is defined as

$$d_H(P, Q) = \max\{\max_{p \in P} \min_{q \in Q} d(p, q), \max_{q \in Q} \min_{p \in P} d(p, q)\}$$

Informally, the Hausdorff distance is the longest distance one can be forced to travel by an adversary who chooses a point in one of the two sets, from where you must travel to the other set. As such, two clusters are close in the Hausdorff distance if their locations and shapes are similar with each other, which is exactly what we expect for the stationariness property as mentioned in Section I.

Essentially, the concept of crowd captures all the properties of a gathering except the last one, i.e., it has no restriction on the membership. Before defining the gathering, we introduce the notion of participator first.

Definition 3 (Participator): Given a crowd Cr, an object o is called a *participator* of Cr iff it appears in at least k_p snapshot clusters of Cr. Let $Cr(o)$ denote the set of snapshot clusters in Cr that contains object o, i.e., $Cr(o) = \{c_t \mid c_t \in Cr, o(t) \in c_t\}$. Then the participators of Cr are the object set $Par(Cr) = \{o \mid |Cr(o)| \geq k_p\}$.

Note that a participator needs not to stay in the crowd for continuous k_p timestamps. As long as an object occurs in the crowd for long enough time, it is regarded as a participator. This kind of flexibility allows an individual to enter and leave a crowd multiple times, which is an usual phenomenon.

Definition 4 (Gathering): A crowd Cr is called a *gathering* iff there exists at least m_p participators in each snapshot cluster of Cr, i.e., $\forall c_t \in Cr$, $|\{o \mid o(t) \in c_t, o \in Par(Cr)\}| \geq m_p$. A gathering is said to be closed if there is no super-crowd of Cr that is also a gathering.

Example 1: Let's consider Figure 1c again with $k_p = 2, m_p = 3$. To make our explanation clearer, we list the occurrence of each object in both crowds in Table II. The participators are highlighted with bold symbols, and the bottom row shows the number of participators in each cluster. Then it is easy to see that $\langle c_1, c_2, c_4 \rangle$ satisfies the support threshold at every time instant, while $\langle c_1, c_3, c_4 \rangle$ only has three participators in c_1.

TABLE II: Occurrences of the objects in the crowds

object	c_1	c_2	c_4	#	object	c_1	c_3	c_4	#
$\boldsymbol{o_1}$		–	–	2	o_1			–	1
$\boldsymbol{o_2}$	–	–	–	3	$\boldsymbol{o_2}$	–		–	2
$\boldsymbol{o_3}$	–		–	2	$\boldsymbol{o_3}$	–	–	–	3
$\boldsymbol{o_4}$	–	–		2	o_4	–			1
o_5	–			1	$\boldsymbol{o_5}$	–	–		2
o_6				0	o_6		–		1
# Par.	3	3	3		# Par.	3	2	2	

Problem Statement. Given a trajectory set of moving objects $\mathcal{O}_{DB}$, two support thresholds m_c, m_p, two lifetime

thresholds k_c, k_p, and a variation threshold δ, our goal is to find all the closed gatherings from $\mathcal{O}_{DB}$.

III. Discovering Closed Gathering

In this section, we will present our framework for discovering all closed gatherings from a trajectory database. Basically, our framework consists of three phases: snapshot clustering, crowd discovery and gathering detection. In the first phase, we perform density-based clustering on the trajectories of objects at each time point in $\mathcal{T}_{DB}$ to find all the snapshot clusters. To reduce the cost incurred by clustering, we can apply the techniques in [9], which simplifies the original trajectories first by the Douglas-Peucker algorithm and then perform clustering on the line segments. Each cluster of line segments contains the objects that are possible to form a snapshot cluster at some time point. Finding snapshot clusters on such a set of objects is much more efficient than on the whole object set directly. The details of this phase are omitted due to space limitation, and it finally outputs a database of snapshot clusters $C_{DB} = \{C_{t_1}, C_{t_2}, ..., C_{t_n}\}$.

The second phase aims to find all the closed crowds from C_{DB}, while the third phase will validate each closed crowd to see if it is or contains closed gathering(s). In the next two subsections, we will elaborate our proposed techniques for improving the performance of these two phases respectively. The last subsection will discuss how to handle the new data arrivals more efficiently.

A. Crowd Discovery

It is easy to observe that the crowd satisfies the *downward closure* property, which means any l-length subsequence of a crowd ($l \geq k_c$) is also a crowd, making it redundant to output all the sub-crowds. More importantly, gatherings detected from a non-closed crowd is not guaranteed to be closed since there may exist longer gatherings in its super-crowds. Therefore, instead of finding all the crowds, we only discover the closed crowds in this phase. At first glance, this needs to check every supersequence for a crowd in order to decide whether it is closed. But actually, according to the following lemma, checking the supersequence of a crowd by appending one more snapshot cluster suffices.

Lemma 1: Given a crowd $Cr = \{c_{t_i}, c_{t_{i+1}}, ..., c_{t_j}\}$, if $\nexists c_{t_{j+1}} \in C_{t_{j+1}}$, s.t. appending $c_{t_{j+1}}$ to Cr will generate a new crowd, then Cr is a closed crowd. Otherwise, Cr is not closed.

Based on this lemma, we can discover the closed crowds by incrementally appending the snapshot clusters at the next time point to the current set of crowd candidates (denoted as $\mathcal{V}$). Algorithm 1 outlines this process. At each timestamp, we check the last cluster of each crowd candidate to see if it can be extended by appending one more cluster. If so, the extended crowd candidates are inserted back to $\mathcal{V}$ as new candidates. Otherwise, we can conclude it is either a closed crowd (if the length is not smaller than k_c) based on Lemma 1, or not a crowd at all. Note that, at any timestamp the clusters (denoted by R) that cannot be appended to any existing crowd candidate should also be regarded as a new candidate, since it is possible to grow into a crowd later.

Algorithm 1: Discovering Closed Crowds

```
Input: C_DB, m_c, k_c, δ
1  V_cc ← ∅; // set of closed crowds
2  V ← ∅; // set of current crowd candidates
3  for t_i = t_1 to t_n do
4      R ← ∅;
5      for each crowd candidate Cr ∈ V do
6          c_{t_{i-1}} ← the last snapshot cluster of Cr;
7          C'_{t_i} ← RangeSearch(c_{t_{i-1}}, C_{t_i}, δ); // find the set of
           snapshot clusters that are within δ distance to Cr
8          R ← R ∪ C'_{t_i};
9          if C'_{t_i} = ∅ then // Cr cannot be extended
10             if Cr.τ ≥ k_c then
11                 V_cc ← V_cc ∪ Cr; // Cr is a closed crowd
12         else
13             for each c_{t_i} ∈ C'_{t_i} do
14                 if |c_{t_i}| ≥ m_c then
15                     Cr' ← append c_{t_i} to Cr;
16                     V ← V ∪ Cr';
17             Remove Cr from V;
18     Insert C_{t_i} \ R into V; // the snapshot clusters than cannot be
       appended to any current crowd candidate will become new crowd
       candidates
19 return V_cc;
```

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
					c_6^1		
		c_3^1	c_4^1	c_5^1			
c_1^1	c_2^1			c_5^2			
	c_2^2	c_3^2		c_5^3			
					c_6^2	c_7^1	c_8^1
					c_6^3		

(a)

time	$\mathcal{V}$	$\mathcal{V}_{cc}$
1	$\langle c_1^1 \rangle$	
2	$\langle c_1^1, c_2^1 \rangle$,$\langle c_1^1, c_2^2 \rangle$	
3	$\langle c_1^1, c_2^1, c_3^2 \rangle$,$\langle c_1^1, c_2^2, c_3^2 \rangle$, $\langle c_1^1, c_2^1, c_3^1 \rangle$	
4	$\langle c_1^1, c_2^1, c_3^1, c_4^1 \rangle$	
5	$\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^1 \rangle$, $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^2 \rangle$, $\langle c_5^3 \rangle$	
6	$\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_6^1 \rangle$, $\langle c_5^3, c_6^2 \rangle$, $\langle c_6^3 \rangle$	$\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^2 \rangle$
7	$\langle c_5^3, c_6^2, c_7^1 \rangle$, $\langle c_6^3, c_7^1 \rangle$	$\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^2 \rangle$, $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_6^1 \rangle$
8	$\langle c_5^3, c_6^2, c_7^1, c_8^1 \rangle$, $\langle c_6^3, c_7^1, c_8^1 \rangle$	$\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^2 \rangle$, $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_6^1 \rangle$
9		$\langle c_5^3, c_6^2, c_7^1, c_8^1 \rangle$, $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^2 \rangle$, $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_6^1 \rangle$

(b)

Fig. 2: Illustration of closed crowd discovery

Example 2: We use the example in Figure 2a to illustrate this discovery process. To keep its simplicity, we assume the two clusters in the same row or adjacent rows of the table are close to each other (i.e., their Hausdorff distance is not greater than δ). Let $k_c = 4$, the intermediate status of the

crowd candidate set $\mathcal{V}$ and result set $\mathcal{V}_{cc}$ at each timestamp is shown in Figure 2b

It is easy to see that the most costly part in Algorithm 1 is the procedure $RangeSearch()$, which looks for the clusters from the cluster set at current timestamp C_{t_c}, whose Hausdorff distance with c_i is not greater than δ. A naive implementation of this procedure is to calculate $d_H(c_i, c_j)$ for each $c_j \in C_{t_c}$. Apparently, a single calculation of $d_H(c_i, c_j)$ requires $O(|c_i||c_j|)$ time, and it should be performed over all pairs between current crowd candidates and the clusters at the current time point. This will make the overall computation prohibitively expensive for a large dataset. To address this issue, we will explore spatial indexing techniques to organize the clusters and speed up the search process.

1) Indexing Clusters with R-tree: Actually we do not need the exact Hausdorff distance between two clusters. Instead, it suffices to just know whether their distance is below or above δ. Let $M(c)$ denote the minimum bounding rectangle (MBR) of cluster c and $d_{min}(,)$ the minimum distance between two rectangles. The following lemma holds naturally.

Lemma 2: Given two clusters c_i and c_j, $d_{min}(M(c_i), M(c_j)) \leq d_H(c_i, c_j)$

Based on this lemma, we firstly retrieve a candidate set of clusters from C_{t_c} whose minimum distance with c_i is not greater than δ and then refine the candidates to get the final results. To support efficient candidate search, we index the MBRs of the clusters in C by a R-tree, and then perform a window query against the R-tree, in which the window is the enlarged MBR of c_i by δ. Obviously, clusters contained in the nodes not overlapping with the window are not candidates.

However, d_{min} is rather a loose lower bound for the Hausdorff distance since the latter is the maximum of minimum distance from one cluster to the other. The following lemma provides a tighter lower bound for the Hausdorff distance.

Lemma 3: Let $M.l_a$ denote the a-th side of a rectangle M ($a = 1, 2, 3, 4$). Define the distance function d_{side} to be

$$d_{side}(M(c_i), M(c_j)) = \max_{a \in [1,4]} d_{min}(M(c_i).l_a, M(c_j))^1$$

Then we have $d_{side}(M(c_i), M(c_j)) \leq d_H(c_i, c_j)$.

Proof: Let p_a be the point of the cluster c_i that lies on the side $M(c_i).l_a$. Naturally, $d_{min}(M(c_i).l_a, M(c_j)) \leq d_{min}(p_a, M(c_j))$. From the definition of Hausdorff distance, $d_{min}(p_a, M(c_j)) \leq d_H(c_i, c_j)$ since $p_a \in c_i$. As such, $d_{min}(M(c_i).l_a, M(c_j)) \leq d_H(c_i, c_j)$, $\forall a \in [1, 4]$. By taking their maximum, d_{side} still lower bounds d_H. ∎

To retrieve candidates in the R-tree by utilizing d_{side}, we need slight modifications to the aforementioned window query process as follows. First we enlarge each side of $M(c_i)$ by δ to obtain four rectangles, denoted by $r_a, a = 1, 2, 3, 4$. During the traversal of R-tree, a node needs to be further examined only if it intersects with all the four rectangles.

[1] d_{min} here is used to compute the minimum distance between a side and a rectangle since the side can be regarded as a degenerated rectangle

2) Indexing Clusters with Grid: Indexing clusters with R-tree, though improving the discovery performance by ruling out many disqualifying clusters, still suffers from three major drawbacks. First, an R-tree needs to be constructed and maintained for each time point, which may incur high cost. Second, since the density-based clusters may have arbitrary shapes, rectangular bounding box cannot always capture the distribution of points in a cluster, which will affect its pruning effect. Third, the brute-force refinement is still needed to evaluate the Hausdorff distances for those candidate clusters. To address them, we propose a grid-based index for clusters. As we shall see shortly, the grid index is easier to construct since the clusters at all timestamps can share the same grid structure. More effective pruning can be performed as the composition of grid cells can approximate the shape of a cluster better. Besides, a smarter refinement algorithm can be devised by utilizing the grid index, which is able to validate a candidate without calculating the exact Hausdorff distance.

To start, we partition the whole space by a grid, each cell of which is a square with the side length equal to $\frac{\sqrt{2}}{2}\delta$. Then for each time point t, we can build a grid index G_t with two kinds of data structures by scanning the set of clusters once, namely a cell list for each cluster $c.cl$ that keeps the cells occupied by the cluster, and an inverted list for each cell $g.inv$ that stores the clusters covering this cell. Before describing the algorithm, we define the *affect region* for a cell.

Definition 5 (Affect region): Given a cell g_{ab} locating at the a-th row and b-column of a grid G, its *affect region* is the set of cells whose minimum distance with g_{ab} is not greater than δ. More precisely, $AR(g_{ab}) = \{g_{ij} \in G \mid |i-a| \leq 2, |j-b| \leq 2,\ and\ |i-a| + |j-b| < 4\}$.

Intuitively, the affect region of a cell g may contain some point whose distance with a point in g is not greater than δ. Now given the query cluster c_i, (i.e., the last cluster of some crowd candidate) and grid index $G_{t_{i+1}}$ at the next timestamp, the procedure $RangeSearch()$ of Algorithm 1 works in the pruning-refinement style, stated as follows.

In the pruning phase, we select each cell g from $c_i.cl$ and find the clusters in $C_{t_{i+1}}$ whose cell list intersects with $AR(g)$. Easy to know that, only the clusters that overlap with the affect region of every cell in $c_i.cl$ can be the candidates, since otherwise there exists at least one point in the cluster that is farther away from c_i than δ.

In the refinement phase, we will validate each candidate to determine the final results. For a candidate c_j, we first perform a set join on $c_i.cl$ and $c_j.cl$ to get their common cells. The rational behind is that the distance between any two points within the same cell cannot be greater than δ. In other words, the Hausdorff distance between the subsets of c_i and c_j that fall inside their common cells will not exceed δ. In an extreme case, if $c_i.cl = c_j.cl$, we can immediately conclude $d_H(c_i, c_j) \leq \delta$. For this reason, we just need to check the cells in their difference set, i.e., $dif(c_i.cl, c_j.cl) = (c_i.cl \cup c_j.cl) \setminus (c_i.cl \cap c_j.cl)$. For each point p within $dif(c_i.cl, c_j.cl)$, assuming $p \in c_i$ without loss of generality, we calculate its minimum distance with c_j. Note

that we only need to calculate the distances between p and the points falling inside the affect region, since all the other points will definitely have distances with p greater than δ.

B. Gathering Detection

In this subsection, we will discuss the algorithm to detect closed gatherings on each closed crowd obtained from the last step. It seems that we can apply the similar methodology with the crowd discovery – incrementally extending a shorter cluster sequence into a longer one until it fails to be a gathering. However, the downward closure property does not hold for gatherings. In other words, a non-gathering cannot imply its supersequences also not being gatherings. To see this, consider a crowd with four clusters $c_1 = \{o_1, o_2, o_3\}$, $c_2 = \{o_1, o_2, o_4\}$, $c_3 = \{o_1, o_3, o_4\}$, $c_4 = \{o_2, o_3, o_4\}$, and let $k_p = 3, m_p = 2$. Obviously, neither the crowd $\langle c_1, c_2, c_3\rangle$ nor $\langle c_2, c_3, c_4\rangle$ is a gathering as the number of participators in $c_2(c_3)$ is less than m (only 1). But when we see their super-crowd $\langle c_1, c_2, c_3, c_4\rangle$, it is a gathering indeed. As such, for a gathering found so far, we have to check all the super-crowds in order to know if it is closed. Undoubtedly, this will incur high computation cost especially when the given crowd is a long sequence.

1) Test-and-Divide Algorithm: In the sequel, we propose a test-and-divide (TAD) algorithm that can detect all the closed gatherings in a given crowd efficiently. As shown in Algorithm 2, it starts from the whole closed crowd and tests if it is a gathering. If so, as we shall prove shortly, it is a closed gathering and can be returned immediately. Otherwise, we identifies the *invalid clusters*, which does not have enough participators, and divide the crowd into several subsequences by removing these clusters (some subsequences may not be crowds as their lengths are less than k). For each subsequence that is still a crowd, we repeat the above steps again since some objects may become non-participators now due to the deletion of invalid clusters. This procedure is performed recursively until no crowd can be found any more .

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
	o_1	o_1		o_1	o_1		
o_2	o_2	o_2	o_2			o_2	o_2
o_3	o_3		o_3		o_3	o_3	o_3
o_4		o_4	o_4	o_4	o_4	o_4	o_4
	o_5	o_5	o_5				
				o_6	o_6		

Fig. 3: Illustration of test-and-divide algorithm

Example 3: Consider a closed crowd illustrated in Figure 3, and let $k_c = k_p = 3$, $m_c = m_p = 3$. According to the TAD algorithm, we first apply the $Test()$ procedure on the whole crowd. It is easy to see that, the objects o_1, o_2, o_3, o_4, o_5 are participators w.r.t. the whole crowd. So c_5 is an invalid cluster as it only contains two participators (< 3). By removing c_5, we divide the original crowd into two sub-crowds $Cr_a = \langle c_1, c_2, c_3, c_4\rangle$ and $Cr_b = \langle c_6, c_7, c_8\rangle$. Again, we perform $Test()$ recursively on Cr_a and Cr_b respectively. For Cr_a, though o_1 changes to a non-participator, all the clusters still have enough number of participators, so we output Cr_a as a gathering. But for Cr_b, both o_1 and o_2 become non-participators, making all the three clusters invalid. Since we cannot get any more sub-crowds from Cr_b, the TAD algorithm will terminate.

Algorithm 2: Test and Divide (TAD)

```
Input: Cr, k_c, k_p, m_p
R ← ∅; // the set of closed gatherings
if Test(Cr,k_p,m_p) is true then // test if Cr is a gathering
    return Cr;
else
    C ← find the invalid clusters;
    S_cr ← Divide(Cr, C); // divide Cr by removing clusters in C
    for each Cr' ∈ S_cr do
        if Cr'.τ ≥ k_c then // if Cr' is still a crowd
            R ← R∪ TAD(Cr', k_c, k_p, m_p);
return R;
```

Theorem 1: The gatherings output by TAD are closed.

Proof: We can prove it by contradiction. Suppose at some stage of TAD, we get a sub-crowd $Cr = \langle c_i, c_{i+1}, ..., c_j\rangle$ that turns out to be a gathering. According to the work flow of TAD, the reason we get Cr is that both c_{i-1} and c_{j+1} are invalid clusters. On the other hand, if there exists any super-crowd of Cr such that it is also a gathering, then at least one of c_{i-1} and c_{j+1} should have enough participators, which is contradictory to the previous claim. Therefore Cr is a closed gathering. ■

2) Efficient Implementation with Bit Vector Signature: A straightforward implementation of TAD algorithm is to count the occurrence for each object in the crowd to see if it is a participator, and then check the number of participators for each cluster in the crowd. Obviously this requires $O(m \cdot Cr.\tau)$ time where m is the number of objects in Cr. Even worse, we have to perform the above operations repeatedly from scratch for each sub-crowd obtained.

For a more efficient implementation of TAD, we propose to construct a *bit vector signature* (BVS) for each object of Cr, and all the subsequent steps can be performed with fast bitwise operations. Specifically, given a crowd $Cr = \langle c_1, c_2, ..., c_n\rangle$, the BVS for an object $o \in Cr$ is an n-length bit vector with each bit representing the existence of o in the corresponding cluster.

The BVSs of all the objects in Cr can be constructed by a single scan of the crowd. More importantly, the BVSs only need to be built once and can be used for all the recursions of TAD. The BVS of each object in the crowd of Figure 3 is shown as follows.

$B(o_1)$	0 1 1 0 1 1 0 0
$B(o_2)$	1 1 1 1 0 0 1 1
$B(o_3)$	1 1 0 1 0 1 1 1
$B(o_4)$	1 0 1 1 1 1 1 1
$B(o_5)$	0 1 1 1 0 0 0 0
$B(o_6)$	0 0 0 0 1 1 0 0

Next, we elaborate how to implement the two procedures $Test()$ and $Divide()$ in Algorithm 2 by utilizing the BVS.

Test step. With the BVS of some object o, denoted by $B(o)$, the procedure $Test()$ essentially turns out to be counting the 1 bits in $B(o)$, which is also known as the Hamming weight [15] of a bit vector. While a naive method is to iterate over all the bits of $B(o)$, more efficient implementations have been well studied. One of the best solution known is based on adding the counts in a binary tree pattern [15], in which we first get the number of 1s in every 2-bit piece of $B(o)$, and then in every 4-bit piece,..., and so on so forth. The example below shows how we can get the Hamming weight of $B(o_1)$ in just 3 steps. Let $x = B(o_1)$,

1) Let $m1 = 01010101$,
   $x = (x\&m1) + ((x \gg 1)\&m1) = 01011000$
2) Let $m2 = 00110011$,
   $x = (x\&m2) + ((x \gg 1)\&m2) = 00100010$
3) Let $m4 = 00001111$,
   $x = (x\&m4) + ((x \gg 1)\&m4) = 00000100$

Now the decimal number of x is 4, which is exactly the number of 1s in $B(o_1)$. In the above operations, $m1, m2, m4$ are called *masks* and can be defined properly once the length of the bit vector is known. In general, for any bit vector with n bits, its Hamming weight can always be obtained in $\lceil \log_2(n) \rceil$ steps.

Divide step. In this step we will divide the crowd into a set of subsequences if it fails to be a gathering. Essentially this is to split the BVS of each object into a set of subvectors. But it is worth pointing out, there is no need to process the BVSs of non-participators since a non-participator of a crowd must remain a non-participator in any of its sub-crowds. We also do not have to split the BVS physically, instead of which we can just use a mask to extract the desired part from the original BVS. The mask is also a bit vector having the same length as the BVS. It sets to 1 in the bits corresponding the sub-crowd, and 0 in all the other bits. By performing the AND operation on the original BVS and the mask, we get a new BVS where the bits of the desired sub-crowd are kept while all other bits are zero. For example, in Figure 3 the mask to extract the crowds Cr_a and Cr_b are 11110000 and 00000111 respectively. As such, the $Divide()$ just needs to return a set of masks, which is more compact compared to the subsequences of a crowd, and pass it to the subsequent $Test()$ procedure. By this means, the $Test()$ procedure can use each mask to get the BVSs of the objects in the corresponding sub-crowd directly, thus avoiding the re-construction of BVSs for each sub-crowd.

C. Discovering Gathering Incrementally

We have discussed the efficient algorithms for discovering closed gatherings in a trajectory archive. But in real applications, trajectories are often received incrementally. As such, the latest batch of trajectory data should be appended to the database periodically (e.g., every day, week or month). Specifically, consider a trajectory database $\mathcal{O}_{DB}$ with the time domain $\mathcal{T}_{DB} = \{t_1, t_2, ..., t_n\}$. After a new batch of trajectories $\mathcal{O}_{new}$ with the time domain $\mathcal{T}_{new} = \{t_{n+1}, ..., t_u\}$ has been collected and appended to $\mathcal{O}_{DB}$, we obtain an updated database $\mathcal{O}'_{DB} = \mathcal{O}_{DB} \cup \mathcal{O}_{new}$ with the extended time domain $\mathcal{T}'_{DB} = \mathcal{T}_{DB} \cup \mathcal{T}_{new}$.

A great challenge posed by this incremental update is that, some closed crowds found in the old database may not be closed any more, since they may be extended with the clusters in $\mathcal{O}_{new}$. Consequently, a closed gathering is also possible to change if its resident crowd is extended. To get the correct results up-to-date, a straightforward solution is to directly apply the aforementioned techniques to find the gatherings in $\mathcal{T}'_{DB}$ from scratch. Obviously this approach becomes more expensive as the size of the database grows, which will make it intractable eventually. To address this issue, we propose an incremental algorithm that can produce the new closed gatherings efficiently by taking full advantage of the previously found crowds and gatherings in the old database.

1) Crowd Extension: First of all, the following lemma states that only some of the crowds (or crowd candidates) in the old database are extensible.

Lemma 4: Given a closed crowd $Cr = \{c_i, ..., c_j\}$ in $\mathcal{O}_{DB}$, if its last cluster is not at the most recent time point of $\mathcal{T}_{DB}$, i.e., $c_j \notin C_n$, where C_n is the cluster set at t_n, then Cr cannot be extended in $\mathcal{O}'_{DB}$.

Based on this lemma, we only need to consider the set CS of cluster sequences that end at t_n in $\mathcal{O}_{DB}$ to see if they can be extended into new crowds. These cluster sequences include closed crowds and the crowd candidates (whose lengths is still less than k) in the old database. To this end, we slightly modify Algorithm 1, such that it saves the crowd candidates and the closed crowds, which end at the last timestamp. Then, after the new set of trajectories $\mathcal{O}_{new}$ has been received and transformed into the cluster database, i.e., $\langle C_{n+1}, ..., C_u \rangle$, the process of Algorithm 1 can be resumed by setting the time cursor t_i to t_{n+1} and the current crowd candidate set $\mathcal{V}$ to CS.

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
					c_6^1				c_{10}^1		
		c_3^1	c_4^1	c_5^1						c_{11}^1	c_{12}^1
c_1^1	c_2^1			c_5^2				c_9^1	c_{10}^2		
	c_2^2	c_3^2		c_5^3							
					c_6^2	c_7^1	c_8^1	c_9^2			
					c_6^3						

(a)

time	$\mathcal{V}$	$\mathcal{V}_{cc}$
9	$\langle c_5^3, c_6^2, c_7^1, c_8^1, c_9^2 \rangle$, $\langle c_6^3, c_7^1, c_8^1, c_9^2 \rangle$, $\langle c_9^1 \rangle$	
10	$\langle c_9^1, c_{10}^2 \rangle$, $\langle c_{10}^1 \rangle$	$\langle c_5^3, c_6^2, c_7^1, c_8^1, c_9^2 \rangle$, $\langle c_6^3, c_7^1, c_8^1, c_9^2 \rangle$
11	$\langle c_9^1, c_{10}^2, c_{11}^1 \rangle$, $\langle c_{10}^1 c_{11}^1 \rangle$	$\langle c_5^3, c_6^2, c_7^1, c_8^1, c_9^2 \rangle$, $\langle c_6^3, c_7^1, c_8^1, c_9^2 \rangle$
12	$\langle c_9^1, c_{10}^2, c_{11}^1, c_{12}^1 \rangle$, $\langle c_{10}^1 c_{11}^1, c_{12}^1 \rangle$	$\langle c_5^3, c_6^2, c_7^1, c_8^1, c_9^2 \rangle$, $\langle c_6^3, c_7^1, c_8^1, c_9^2 \rangle$
13	save $\langle c_{10}^1 c_{11}^1, c_{12}^1 \rangle$, $\langle c_9^1, c_{10}^2, c_{11}^1, c_{12}^1 \rangle$ to CS for further possible extension	$\langle c_9^1, c_{10}^2, c_{11}^1, c_{12}^1 \rangle$, $\langle c_5^3, c_6^2, c_7^1, c_8^1, c_9^2 \rangle$, $\langle c_6^3, c_7^1, c_8^1, c_9^2 \rangle$

(b)

Fig. 4: Illustration of crowd extension

Example 4: Continuing with Example 2, a new set of clusters during the time period $\{t_9, t_{10}, t_{11}, t_{12}\}$ has been appended to the original database, as shown in Figure 4a. Based on Lemma 4, the two closed crowds of the old database $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^2\rangle$ and $\langle c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_6^1\rangle$ cannot be extended since they end before t_8. The only two cluster sequences we have to consider for extension are $\langle c_5^3, c_6^2, c_7^1, c_8^1\rangle$ that is already a crowd and $\langle c_6^3, c_7^1, c_8^1\rangle$ that is a crowd candidate. By initializing $\mathcal{V}$ to contain these two candidates, and setting the current time to be t_9, we resume the iteration of Algorithm 1. This process is illustrated in Figure 4b.

2) Gathering Update: Suppose a crowd $Cr_{old} = \{c_i, ..., c_n\}$ in $\mathcal{O}_{DB}$ has been extended into a new closed crowd $Cr_{new} = \{c_i, ..., c_n, c_{n+1}, ..., c_m\}$ in $\mathcal{O}'_{DB}$. Now our goal is to find the closed gatherings in Cr_{new}. Trivially, we can perform the TAD algorithm on Cr_{new} from scratch. But as some gatherings have already been detected for Cr_{old} already, it is possible to speed up the discovery process by making use of them wisely. This optimization will bring more benefits when Cr_{old} occupies a large portion of Cr_{new}.

As before, we firstly build the BVS for each object in Cr_{new}, and then run the $Test()$ procedure to detect the invalid clusters. The following lemma indicates that some original invalid clusters in Cr_{old} may become valid in Cr_{new}.

Lemma 5: Denote the set of invalid clusters of a crowd Cr as $IC(Cr)$. Then we have $IC(Cr_{new}) \cap Cr_{old} \subseteq IC(Cr_{old})$.

This is natural since some non-participators in Cr_{old} may turn to be participators because of the new clusters in Cr_{new}. In other words, the gatherings in Cr_{old} may expand or merge with their neighboring gatherings in Cr_{new}. However, if we find some invalid cluster c_j in Cr_{new} which also belongs to Cr_{old}, it is guaranteed that all the closed gatherings before t_j remain unchanged in Cr_{new}. More precisely, we have the following theorem,

Theorem 2: Given an invalid cluster $c_j \in IC(Cr_{new})$ with $j \leq n+1$, then any closed gathering $Gr \subset \langle c_i, ..., c_{j-1}\rangle$ remains closed in Cr_{new}.

Proof: Since c_j is an invalid cluster, we can only find closed gathering from $Cr_a = \langle c_i, ..., c_{j-1}\rangle$ and $Cr_b = \langle c_{j+1}, ..., c_m\rangle$. a). If $j = n+1$, Cr_a is actually Cr_{old}. So they have the same set of closed gathering. b). If $j < n+1$, then $c_j \in IC(Cr_{new}) \cap Cr_{old}$. From lemma 5, we know that $c_j \in Cr_{old}$. This means in Cr_{old}, the closed gatherings locate in $Cr_c = \langle c_i, ..., c_{j-1}\rangle$ and $Cr_d = \langle c_{j+1}, ..., c_n\rangle$. $Cr_a = Cr_c$, hence their closed gatherings are also the same. ■

Motivated by Theorem 2, we can improve the original TAD algorithm by utilizing the gatherings found in Cr_{old}. After a set of invalid clusters IC has been obtained in the test phase, we look for the "rightmost" invalid cluster before the timestamp t_{n+1}, i.e., $c_j \in IC(Cr_{new})(j \leq n+1)$, s.t. $\nexists c_{j'} \in IC(Cr_{new})$ that $j < j' \leq n+1$. Theorem 2 guarantees that the closed gatherings on $\langle c_i, ..., c_{j-1}\rangle$ remain the same as before, which have already been discovered. Therefore only the sub-crowds within $\langle c_{j+1}, ..., c_m\rangle$ need to be examined further since they may contain new or updated gatherings.

IV. Experiment

In this section, we conduct extensive experiments to evaluate the effectiveness and efficiency of our proposed concepts and algorithms based on a real trajectory dataset, which contains about 120K trajectories generated by over 33,000 taxis of Beijing in a period of 3 months (March, April and May in 2009) [16], [17]. After discretizing the time domain into the granularity of minute, we get 132,480 time points ($60 \times 24 \times 92$) in $\mathcal{T}_{DB}$. Then as a offline preprocessing step, we find the snapshot clusters for every minute by applying the DBSCAN [14] with the settings $m = 5, \epsilon = 200(meters)$. All the algorithms in the following experiments are implemented in C# and run on a computer with Intel Xeon Core 4 CPU (2.66GHz) and 8GB memory.

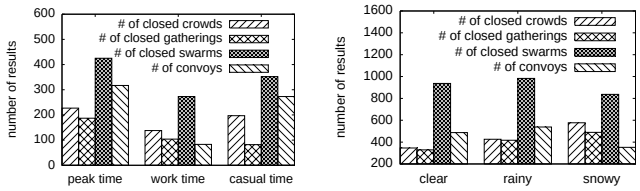
A. Effectiveness

Although the gathering is capable of modelling various group incidents as mentioned in Section I, in this part we use the traffic condition (e.g.,traffic jams) as a study case to evaluate the effectiveness of our proposals. Intuitively, a traffic jam can be captured by a gathering, since many vehicles with slow speeds form a dense area, and usually most of the vehicles stay within this area for a relatively long time. Essentially, GPS-equipped taxicabs can be viewed as ubiquitous mobile sensors of the city-wide traffic flows. For instance, Beijing has approximately 67,000 licensed taxis generating over 1.2 million occupied trips per day. This figure is around 4.2% of the total personal trips (35 million) within the Six Ring Road of Beijing City (reported by Beijing transportation bureau in July 2010), which is a significant sample reflecting the traffic condition of the city.

In the first experiment, we divide a day into three time periods, peak time (6am to 10am and 5pm – 8pm), work time (10am to 5pm) and casual time (8pm to 5am). Then we find all the closed crowds and gatherings from the trajectory set and group them by the time period with the setting $m_c = 15$, $\delta = 300m$, $k_c = 20$, $k_p = 15$ and $m_p = 10$. As comparison, we also search for the closed swarms and convoys from the trajectories with the settings $min_o = 15, min_t = 10$ (i.e., a group of 15 or more objects travelling together for a period of at least 10 time units). In cases a pattern crosses multiple time periods, we simply duplicate assign it to each of them. Figure 5a shows the average number of each pattern in a single day w.r.t. the time period. It is easy to see that, we can find the most gatherings during the peak time and much fewer for the rest. This observation is consistent with the traffic condition of Beijing, since it experiences severe traffic congestion during the rush hours every day. Interestingly, though there also exist many crowds in casual time, only a small portion turns out to be gatherings. This is because, many crowds are located around restaurants, shopping malls and other entertainment places, where taxicabs usually drop the passengers and leave quickly. As such, these crowds will not form gatherings since there are not enough participators. On the other hand, we can find more swarms and convoys in peak and casual time than in work time. To explain this, many taxicabs have common

destination areas during peak (e.g., CBD, residential suburbs) and casual time (e.g., entertainment places). On the contrary, the destinations of most taxicabs are widely distributed during work time, resulting in fewer swarms and convoys.

Next, we categorize the total 92 days into three groups according to the weather condition, namely clear, rainy and snowy. Then we compare the average number of each pattern in a single day with different weather conditions. As shown in Figure 5b, we can find the least number of gatherings in clear days and the most in snowy days. The reason is that, as the weather condition becomes worse for the traffic, vehicles tend to move more slowly which makes it easier to cause traffic jams. We also notice the great gap between the number of crowds and gatherings in snowy days. This may be caused by the large number of minor accidents on the roads, in which most vehicles around the accident can bypass it in a short time. We also note that the number of swarms seems quite insensitive to the weather, while there are fewer convoys in snowy days. A possible explanation is that vehicles try not to travel too closely to each other in snowy weather.



(a) Time of the day (b) Weather condition

Fig. 5: Effectiveness study

B. Efficiency

In this subsection, we study the efficiency of our proposed algorithms. In particular, we will measure the run time of the algorithms for discovering closed crowds, detecting closed gatherings and handling the database update incrementally in different parameter settings.

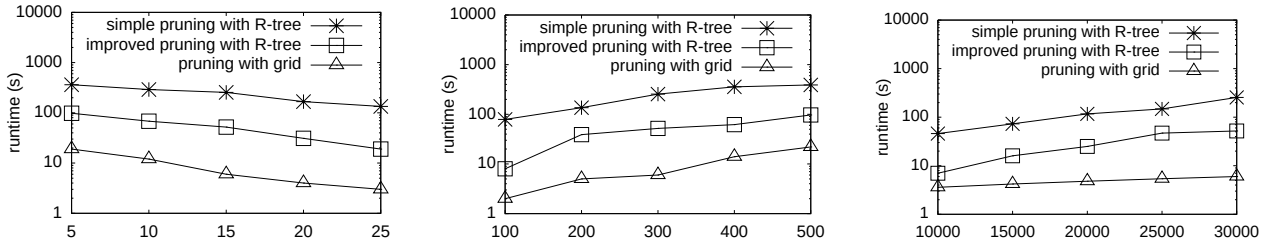
Performance of crowd discovery algorithm. In the first set of experiments, we compare the performances of three pruning schemes in the crowd discovery algorithm: a) SR, simple R-tree based pruning with d_{min}; b) IR, improved R-tree based pruning with d_{side}; c) GRID, grid-based pruning. The default parameters used in this set of experiments are: $|\mathcal{O}_{DB}| = 30,000$, $m_c = 15$, $\delta = 300m$, $k_c = 20$. Below we show the average runtime cost of searching for the closed crowds in a single day, i.e., $|\mathcal{T}_{DB}| = 1440$. It is worth mentioning that, since Algorithm 1 sequentially sweeps all the time points, the parameter k_c only affects the number of gatherings we can find, but has no impact on the time cost of the algorithm. So we omit the experiment studying k_c in the sequel.

As we can see from Figure 6, IR significantly improves the pruning effect of SR by using a tighter lower bound of d_H. GRID further enhances the performance of IR and outperforms SR by at least one order of magnitude constantly. Specifically, as shown in Figure 6a, the runtime costs of all algorithms decrease when m_c increases, since there are less number of clusters satisfying this support threshold at each time instant. As such, there are fewer candidates to consider when we attempt to expand the current crowd candidates. On the contrary, the performances of all methods deteriorate as δ increases (Figure 6b), since the search space increases when we look for the candidate clusters of the next timestamp. Finally, in Figure 6c we study the impact of database size by randomly choosing the subsets of the original dataset with different sizes. As expected, all the schemes need more time to complete on a larger database, since there tends to be more clusters at each time point. Interestingly, however, the grid-based pruning is relatively insensitive to the size of the database. This is due to the fact that, as the affect region of each cluster remains unchanged (since δ remains the same), the refinement cost increases slowly (we only refine the grids in the affect region) even though there may be more clusters considered as candidates.

Performance of gathering detection algorithm. We evaluate the performances of three algorithms for detecting closed gatherings from a given crowd: a) Brute-force method which will recursively test all the i-length sub-crowds ($i = n, n-1, ...$), until either it finds a gathering or no sub-crowd can be found ($i < k_c$); b) TAD algorithm; c) TAD*: TAD algorithm implemented with the bit vector signature. The default parameters used in this set of experiments are: $m_p = 11$ and $k_p = 14$. For each experiment, we run the algorithms on 1000 closed crowds that are randomly selected and record the average time cost.

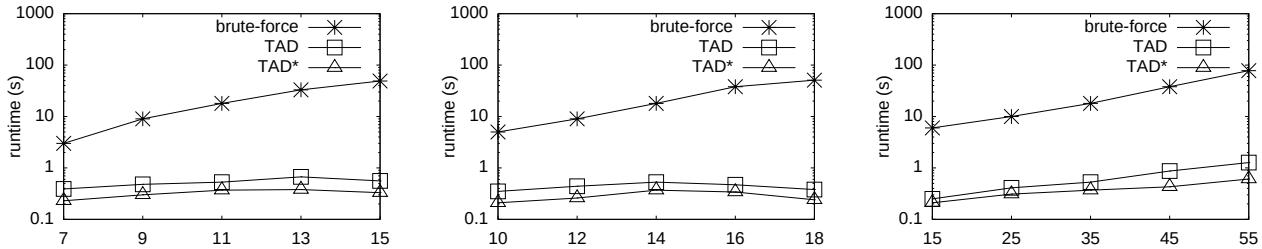
From Figure 7, it is easy to see that TAD outperforms the brute-force method by one to two orders of magnitude, and TAD* further improves TAD by about 30%. In Figure 7a, we show the performances of all three algorithms with the variation of m_p, i.e., the least number of participators for a cluster to be valid. As m_p increases, a cluster is more likely to be invalid. For this reason, the brute-force method has to check the shorter sub-crowds with more recursions until it finds a gathering. Although TAD and TAD* also have recursive procedures, they do not enumerate all the subsequences of a crowd. Interestingly, with the further increase of m_p, the time costs of TAD and TAD* turn to decrease. This is because too many invalid clusters in the original crowd will result in a large number of subsequences that are non-crowd, hence making the recursion terminate more quickly. Figure 7b shows the effect of the other parameter k_p, which is the least time period for a participator to stay within the crowd. As the previous experiment, there will be less valid clusters with greater k_p since the number of participators decreases. We omit the analysis for k_p due to its similarity with m_p.

We also investigate the runtime cost when the algorithms are performed on the crowds with different lengths ($Cr.\tau$), the results of which are shown in Figure 7c. As expected, the time cost of the brute-force method increases almost exponentially with $Cr.\tau$, since the number of subsequences is exponential to the length of a crowd. The performances of the other two



(a) Running time w.r.t. m_c (b) Running time w.r.t. δ (meter) (c) Running time w.r.t. $|\mathcal{O}_{DB}|$

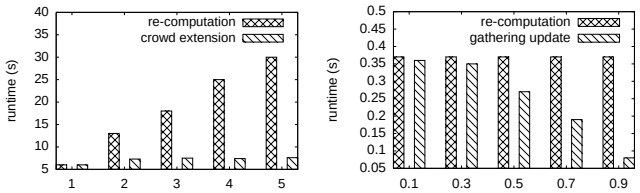
Fig. 6: Running time of closed crowd discovery



(a) Running time w.r.t. m_p (b) Running time w.r.t. k_p (min) (c) Running time w.r.t. $Cr.\tau$ (min)

Fig. 7: Running time of closed gathering detection

algorithms also deteriorate with the length of the crowd, but the changes are more smooth. In addition, TAD* exhibits more benefits on longer $Cr.\tau$, since using the BVSs for a longer sequence can save more computation time.



(a) $|\mathcal{T}_{DB}|$ (b) r

Fig. 8: Runtime costs of incremental vs. re-computation methods when handling new data arrival

Performance of incremental algorithms. Finally we analyse the performances of incremental algorithms for handling database update compared with the method of re-computation from scratch. To study the benefit of the crowd extension algorithm, we set the trajectories in one day randomly selected from the 92 days as the initial dataset, and then append the trajectories of the next consecutive 4 days (there are 1440 time points for each day), one at a time, to construct a newer dataset. Figure 8a illustrates the average runtime costs of different methods (with grid-based pruning) for discovering the closed crowds after each update. Not surprisingly, the time cost of re-computation method increases proportionally with the time domain. It is easy to predicate that, with the continuing increase of the database size, this method will become prohibitively expensive eventually. On the other hand, the consumed time of the crowd extension algorithm remains almost constant since it can skip processing of the most trajectories in the old dataset. Note that the computation on each successive update is a bit more expensive than on the initial dataset. The extra cost is caused by the processing of the crowd candidate set saved from the old dataset.

Next, we compare the time costs of re-computation and gathering update algorithm (both employing TAD* method) for detecting closed gatherings on the extended crowds. Figure 8b summarizes their average running time w.r.t. the ratio r between the lengths of the old crowd and the extended crowd. As expected, the variation of r does not affect the re-computation method, which applies the TAD* algorithm on the extended crowds from scratch. On the contrary, the gathering update algorithm can detect the closed gatherings faster when the old crowd occupies a larger portion of the new crowd, since it takes full advantage of the previously computed gatherings in the old crowd.

We also test the performance of the incremental algorithms w.r.t. all other parameters as in the experiments of Figure 6 and Figure 7. Since they exhibit the similar behaviours with the previous experiments, we omit the presentation of these results due to the space limitations.

V. Related Work

Most of the related work on group pattern mining has been discussed in Section I. In addition, Tang et al [18] recently study the problem of discovering travelling companions from streaming trajectories. The concept of travelling companion is essential the same as convoy [9]. But they focus on incremental discovery algorithms when new trajectory data

arrive continuously. In the sequel, we give a brief review about the research on dense area detection and trajectory clustering.

Dense area detection. Dense area detection was initially presented in the data mining community as the identification of the set(s) of regions, from spatio-temporal data, that satisfy a minimum density threshold. The STING method [19] is a fixed-size grid-based approach to generate hierarchical statistical information from spatial data. Density query for moving objects is first proposed in [2]. Its objective is to find regions with the density higher than a given threshold at a time point or for a period of time in the near future. They find the general density-based queries difficult to answer efficiently and hence turn to simplified queries. Specifically, they partition the data space into disjoint cells, and the simplified density query reports cells, instead of arbitrary regions, which satisfy the query conditions. Later, Jensen et al. [3] defines a more delicate types of density query with desirable properties to address the answer loss problem in [2]. Some other solutions to detect dense areas are based on the identification of *local maxima* by using the techniques from computer vision [20][21]. Common to all the above methods is that a fixed-size non-overlapping grid is employed to aggregate the values over the spatial dimensions, which might not correspond to the real shape of the underlying dense area. On the contrary, the gathering pattern in our work can capture the dense areas with arbitrary shapes.

Trajectory clustering. Trajectory clustering techniques aim to find groups of moving object trajectories that are close to each other and have similar geometric shapes. Gaffney et al. [22][23] propose trajectory clustering methods based on probabilistic modelling of a set of trajectories. As pointed out by Lee et al [24], distance measure based on whole trajectories may miss interesting common paths in sub-trajectories. Motivated by this, Lee et al. [24] designed a partition-and-group framework, which partitions trajectories into line segments and then build groups for those close segments. More recently, Li et al. [25] further study the efficient algorithms for maintaining and updating the clusters when trajectories are received incrementally. Different with the group pattern mining and our work, this category of proposals does not consider the temporal aspects of the trajectories. As such, moving objects whose trajectories are in the same cluster may not actually stay together temporally.

VI. Conclusion

In this paper, we study the problem of discovering closed gathering patterns from a large-scale trajectory database. Different from the earlier proposed concepts, such as flock, convoy and swarm, which aim to identify groups of moving objects travelling together for a certain time period, the gatherings are able to model a variety of non-trivial group events or incidents. Since the whole discovery process with straightforward solutions can be very lengthy in a practical dataset, we propose a series of techniques which address the indexing, searching and updating issues respectively. At last we validate the effectiveness and efficiency of our proposals by conducting extensive experiments based on a real and large-scale taxicab trajectory dataset.

Acknowledgement

This work was supported by ARC grants DP120102829 and DP110103423.

References

[1] Y. Zheng and X. Zhou, *Computing with spatial trajectories*. Springer, 2011.

[2] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. Tsotras, "On-line discovery of dense areas in spatio-temporal databases," *Advances in Spatial and Temporal Databases*, pp. 306–324, 2003.

[3] C. Jensen, D. Lin, B. Ooi, and R. Zhang, "Effective density queries on continuously moving objects," in *ICDE*, 2006, pp. 71–71.

[4] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle, "Reporting flock patterns," *Computational Geometry*, vol. 41, no. 3, pp. 111–125, 2008.

[5] M. Vieira, P. Bakalov, and V. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in *ACM GIS*, 2009, pp. 286–295.

[6] J. Gudmundsson and M. van Kreveld, "Computing longest duration flocks in trajectory data," in *ACM GIS*. ACM, 2006, pp. 35–42.

[7] J. Gudmundsson, M. van Kreveld, and B. Speckmann, "Efficient detection of motion patterns in spatio-temporal data sets," in *ACM GIS*, 2004, pp. 250–257.

[8] G. Al-Naymat, S. Chawla, and J. Gudmundsson, "Dimensionality reduction for long duration and complex spatio-temporal queries," in *ACM symposium on Applied computing*, 2007, pp. 393–397.

[9] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen, "Discovery of convoys in trajectory databases," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1068–1080, 2008.

[10] H. Jeung, H. Shen, and X. Zhou, "Convoy queries in spatio-temporal databases," in *ICDE*, 2008, pp. 1457–1459.

[11] Z. Li, B. Ding, J. Han, and R. Kays, "Swarm: Mining relaxed temporal moving object clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 723–734, 2010.

[12] P. Kalnis, N. Mamoulis, and S. Bakiras, "On discovering moving clusters in spatio-temporal data," *Advances in Spatial and Temporal Databases*, pp. 364–381, 2005.

[13] G. Rote, "Computing the minimum hausdorff distance between two point sets on a line under translation," *Information Processing Letters*, vol. 38, no. 3, pp. 123–127, 1991.

[14] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *SIGKDD*, vol. 96, 1996, pp. 226–231.

[15] D. Knuth, "The art of computer programming, volume 4, fascicle 1: Bitwise tricks & techniques; binary decision diagrams," 2009.

[16] J. Yuan, Y. Zheng, X. Xie, and G. Sun, "Driving with knowledge from the physical world," in *SIGKDD*, 2011, pp. 316–324.

[17] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang, "T-drive: driving directions based on taxi trajectories," in *GIS*, 2010, pp. 99–108.

[18] L. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. Hung, and W. Peng, "On discovery of traveling companions from streaming trajectories," in *ICDE*, 2012, pp. 186–197.

[19] W. Wang, J. Yang, and R. Muntz, "Sting: A statistical information grid approach to spatial data mining," in *VLDB*, 1997, pp. 186–195.

[20] D. Agarwal, A. McGregor, J. Phillips, S. Venkatasubramanian, and Z. Zhu, "Spatial scan statistics: approximations and performance study," in *SIGKDD*, 2006, pp. 24–33.

[21] M. Kulldorff, "A spatial scan statistic," *Communications in statistics: theory and methods*, vol. 26, no. 6, pp. 1481–1496, 1997.

[22] S. Gaffney, A. Robertson, P. Smyth, S. Camargo, and M. Ghil, "Probabilistic clustering of extratropical cyclones using regression mixture models," *Climate dynamics*, vol. 29, no. 4, pp. 423–440, 2007.

[23] S. Gaffney and P. Smyth, "Trajectory clustering with mixtures of regression models," in *SIGKDD*, 1999, pp. 63–72.

[24] J. Lee, J. Han, and K. Whang, "Trajectory clustering: a partition-and-group framework," in *SIGMOD*, 2007, p. 604.

[25] Z. Li, J. Lee, X. Li, and J. Han, "Incremental clustering for trajectories," in *Database Systems for Advanced Applications*, 2010, pp. 32–46.