On-Line Discovery of Flock Patterns in Spatio-Temporal Data

Marcos Vieira
University of California
Riverside, CA 92507, USA
mvieira@cs.ucr.edu

Petko Bakalov
ESRI
Redlands, CA 92373, USA
pbakalov@esri.com

Vassilis Tsotras
University of California
Riverside, CA 92507, USA
tsotras@cs.ucr.edu

ABSTRACT

With the recent advancements and wide usage of location detection devices, large quantities of data are collected by GPS and cellular technologies in the form of trajectories. While most previous work on trajectory-based queries has concentrated on traditional range, nearest-neighbor and similarity queries, there is a increasing interest in queries that capture the "aggregate" behavior of trajectories as groups. Consider for example, finding groups of moving objects that move "together", i.e. within a predefined distance to each other, for a certain continuous period of time. Such queries typically arise in surveillance applications, e.g. identify groups of suspicious people, convoys of vehicles, flocks of animals, etc. In this paper we first show that the on-line flock discovery problem is polynomial and then propose a framework and several strategies to discover such patterns in streaming spatio-temporal data. Experiments with real and synthetic trajectorial datasets show that the proposed algorithms are efficient and scalable.

1. INTRODUCTION

Recent advances in the area of location detection devices (RFID, GPS, etc.) and their widespread use have enabled the creation of complex tracking and situational awareness systems which continuously monitor the position of moving objects of interest. Examples include *AccuTracking*, *tracNET24*, Path Intelligence's *FootPath*, InSTEDD's *GeoChat* and many others. This abundance of information, generated by those systems, motivates the need to develop efficient techniques for answering interesting queries about the past behavior of the moving objects like discovering similarity patterns among the object trajectories.

The existing methods on querying trajectories are mainly focused on answering simple single predicate range or nearest neighbor queries. Examples include queries like "find all moving objects that were in area A at 10 a.m. (in the past)" or "find the car which drove as close as possible to the location B during the time interval (10am:1pm)". Recently a new group of similarity search querying methods have emerged. The result of a similarity search query is a trajectory closest to the query trajectory according to some metric distance (e.g. Euclidean, Dynamic Time Warping, etc.). Common to all the above methods is that the query answer is validated per trajectory. That is, a trajectory is reported to the user if its individual behavior satisfies the query predicate(s). In other words, all the above queries focus on the behavior of a trajectory as a single object and thus cannot be used to discover group patterns between the trajectories.

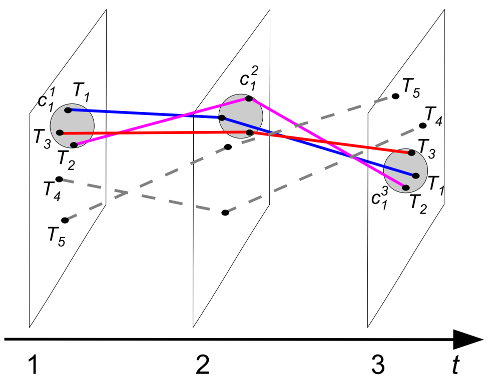

Figure 1: A flock pattern example: $\{T_1, T_2, T_3\}$

Recently there has been increased interest in querying patterns capturing "collaborative" or "group" behavior between moving objects. This includes queries like moving clusters [13, 10], convoy queries [12] and flocks patterns [5, 8]. Such queries discover groups of moving objects that have a "strong" relationship in the space for a given time duration. The difference between all those patterns is the way they define the relationship between the moving objects. In this paper we consider the discovery of *flock* patterns among the moving objects, i.e., the problem of identifying all groups of trajectories that stay "together" for the duration of a given time interval. We consider moving objects to be "close" together if there exists a disk with given radius that covers all moving objects in the pattern (see Figure 1). A trajectory satisfies the above pattern as long as "enough" other trajectories are contained inside the disk for the specified time interval; that is, the answer is based not only on a given trajectory's behavior but also on the ones near it. Such patterns are useful in security and monitoring applications, for example to potentially identify suspicious behavior within large number of people ("Identify all groups of five or more people that were always within a disk of 100 feet in the last 30 minutes") or to study patterns of animal behavior [1, 2] (e.g. migration of sharks, whales, birds, etc.).

The example in Figure 1 shows a flock pattern containing 3 trajectories $\{T_1, T_2, T_3\}$ that are within a query defined disk for 3 consecutive time instances. Note that the location of the disk can freely "move" in the 2-dimensional space in order to accommodate all three moving objects and its center does not need to coincide with any moving object location for a given time instance. This makes the discovery of flock patterns difficult because there is an infinite number of possible placements of the disk at any time instance. It is that difficulty that makes the existing methods for flock pattern discovery [5, 8] suffer from severe limitations. Such methods either find approximate solutions, or can be applied only for a single time instance of the problem (i.e. the solution does not support the minimum time duration in the query). To the best of our knowledge, our work is the first one to present exact solutions for reporting flock patterns in polynomial time. It is also the first one that does so for online environments. Our work is also different than clustering-based approaches (since clusters are not restricted to a specific shape); flocks are also different than convoy discovery [12]. More details of the previous methods are discussed in Section 2.

We start by providing a complexity analysis for the on-line flock problem. Our analysis reveals that polynomial time solution can be found through identifying a discrete number of locations to place the center of the flock disk inside the spatial universe. The number of such possible locations is polynomial on the total number of moving objects. Based on this analysis we propose several evaluation algorithms which can be used to find flock patterns in polynomial time. The first algorithm is based on time-joins, i.e., merging the results from one time instance to another. The other four algorithms use the *filter-and-refinement* paradigm with the purpose of reducing the total number of candidates and thus the overall computation cost of the algorithm. We evaluate our solutions using several real and synthetic moving object datasets.

The rest of the paper is organized as follows: Section 2 highlights related work while Section 3 formally defines the online flock pattern and provides a complexity analysis on the problem. Section 4 describes the proposed algorithms for flock pattern Discovery. Section 5 presents the performance evaluation of our proposed algorithms and Section 6 concludes the paper.

2. RELATED WORK

Related work can be classified to (i) research on clustering moving objects, (ii) work on discovering convoys between trajectories and (iii) previous work on flock discovery. Various clustering algorithms have been proposed for static spatial datasets, with different strategies ranging from partitioning (e.g. k-medoids [18]), to hierarchical (e.g. *BIRCH* [19] and *CURE* [9]) and density-based (e.g. *DBSCAN* [6]). The *DBSCAN* algorithm is a representative in its category where it works for arbitrary-shaped clusters based on their "density". This method utilizes a distance *eps* and minimum number of points *minPts* parameters to identify dense areas. Points that have more than *minPts* within *eps* radius are considered as a dense area, and then each of these points are processed recursively. Points that do not have at least *minPts* points in their *neighborhood* and are not "reachable" from a dense area are labeled as noisy points. This process of expanding clusters by analyzing each point in the dataset stops when all the objects are analyzed (i.e. all objects assigned to a cluster).

Clustering for moving objects was examined in [13], where the *DBSCAN* algorithm is performed for every time instance of the dataset. Then clusters that have been found for two consecutive time instances $t-1$ and t are joined. The clusters can be joined only if the number of common objects among them are above the predefined parameter θ. This process is applied each time for all time instances in the dataset. Other works on clustering moving objects also include [10, 7, 17, 16, 15]. In [10] techniques were proposed to incrementally update clusters of moving objects based on the cluster centers. The object movements were used to predict the cluster evolution over time. The *MONIC* framework described in [7] deals with transitions in moving clusters, e.g. disappearance and splitting. [17] presented the *microclustering* technique that groups moving objects that are not only close to each other at a specific time instance, but are also expected to move together in the "near" future. Recently, [15, 16] proposed to segment trajectories into line segments. Then line segments are grouped together to build the clusters. Time is not consider in [16], which makes some line segments to be clustered together even though they are not "close" when time is considered. Nevertheless, approaches for clustering moving objects cannot solve the flock pattern query since clusters do not assume any shape restriction.

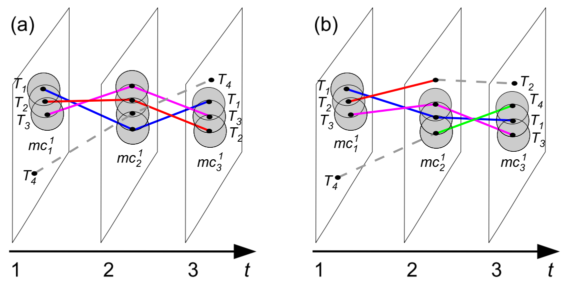

Figure 2: Clustering vs. flock patterns

Related to discovering collaborative behavior between trajectories is the work of finding *convoy patterns* in trajectory archives [12]. A convoy query is defined as a "dense" cluster of trajectories that stay "together" at least for a predefined continuous time. This type of query has four parameters: *eps* and *minPts* (used by the *DBSCAN* algorithm), θ' (threshold used to join clusters), and δ' (minimum duration time). There are however three major differences between flock and convoy queries: **(1)** they use different criteria when joining the moving object clusters for time instances t and $t-1$; **(2)** convoy queries employ clustering algorithms, and therefore no strong relationship among *all* elements are inforced; **(3)** convoy queries does not require the *same* set of moving objects to stay in a cluster all the time for the specified minimum duration. Flocks instead require that objects cannot join or leave the cluster in order to be considered part of the *same* flock.

For example, in Figure 2(a) convoy query returns trajectories $\{T_1, T_2, T_3\}$ for $\theta = 3$ and for 3 time instances, while in Figure 2(b) it does not return nothing. For the moving cluster, if $\theta = 1$ then moving clusters return nothing in both Figure 2(a) and (b). On the other hand, if $\theta = 1/2$ then it returns $\{T_1, T_2, T_3\}$ in Figure 2(a) and $\{T_1, T_3, T_4\}$ in Figure 2(b), but the last one is not a convoy query. Both examples return results based on the density of the objects, but for the flock pattern it would return nothing in either examples. The reason is that in both examples the objects belong to dense areas but they do not have "strong" interaction among them.

Flock pattern query was first introduced in [5, 14], without the notion of minimum lasting time. Later [8] introduced the minimum duration as a parameter of the pattern. Unlike the convoy patterns in a flock the cluster has a predefined shape – a disk with radius r. A set of moving objects is considered a flock if there is a disk with radius r which covers all of them and there are at least some predefined number of objects in the cluster. It is shown in [8] that the discovery of the "longest" duration flock pattern is an *NP-hard* problem. As a result, [8] presents only approximation algorithms.

To the best of our knowledge our paper is the first which proposes a polynomial time solution to the flock problem with a predefined time duration. Moreover our algorithms can be applied in a streaming environment for online discovery of the flock patterns.

3. PRELIMINARIES

We assume that object O_{id} is uniquely identified by identifier id. Its movement is represented by a trajectory T_{id} which is defined as an ordered sequence of n multidimensional points $T_{id} = \{p(t_1), p(t_2), \ldots, p(t_n)\}$. Here t_i is a timestamp and $p(t_i)$ is the location of object O_{id} in the two dimensional space $\mathbb{R}^2$ as recorded at timestamp t_i ($t_i \in \mathbb{N}$, $t_{i-1} < t_i$, and $0 < i \leq n$). For simplicity when we discuss the current time instance, t_i is omitted, and we just use p_{id} to denote the object location.

Given two object locations $p_a^{t_i}$ and $p_b^{t_i}$ in a specific time instance t_i from trajectories T_a and T_b respectively, $d(p_a^{t_i}, p_b^{t_i})$ denotes the L_p distance between p_a and p_b. Even though our methods apply to any family of L_p metric distances, for ease of illustration in the rest of the paper we assume the Euclidean distance. A flock pattern query $Flock(\mu, \epsilon, \delta)$ is defined as follows:

Definition 1. *Given are a set of trajectories $\mathcal{T}$, a minimum number of trajectories $\mu > 1$ ($\mu \in \mathbb{N}$), a maximum distance $\epsilon > 0$ defined over the distance function d, and a minimum time duration $\delta > 1$ ($\delta \in \mathbb{N}$). A flock pattern $Flock(\mu, \epsilon, \delta)$ reports all sets $\mathcal{F}$ of trajectories where: for each set f_k in $\mathcal{F}$, the number of trajectories in f_k is greater than μ ($|f_k| \geq \mu$) and there exist δ consecutive time instances such that for every $t_i \in \delta$, there is a disk with center $c_k^{t_i}$ covering all $f_k^{t_i}$ points. That is: $\forall T_j \in f_k, \forall t_i \in f_k, \forall f \in \mathcal{F} : d(p_j^{t_i}, c_k^{t_i}) \leq \epsilon/2$*

The $c_k^{t_i}$ is called the center of the flock f_k at time t_i. In the above definition, a flock pattern can be viewed as a "tube" shape formed by the centers c and expanded with diameter ϵ, and having length δ (consecutive time instants) such that there are least μ trajectories which stay inside the tube all the time, as shown in Figure 3.

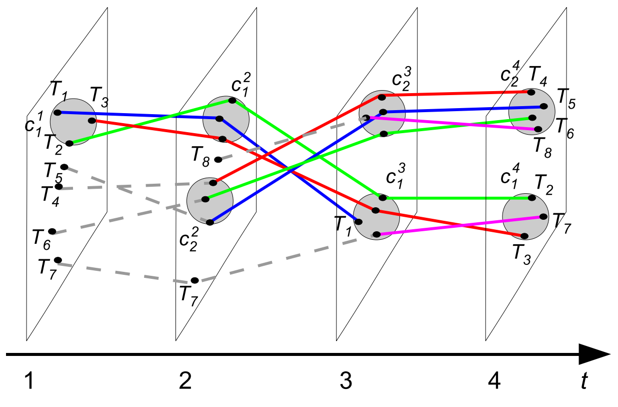

Figure 3: Flock pattern example

Having this formal definition we proceed with the complexity analysis of the flock pattern. The major challenge in this type of queries is the fact that the center of the flock pattern $c_k^{t_i}$ may not belong to any of the trajectories. Hence we cannot iterate over the discrete number of trajectory locations stored in the database and check if each one of them is a center of a flock or not. Since any point in the spatial domain can be a center of a flock there is an infinite number of possible locations to test.

Nevertheless, we show using the following Theorem that there is a limited and discrete number of locations where we can look for flocks among the infinite number of options.

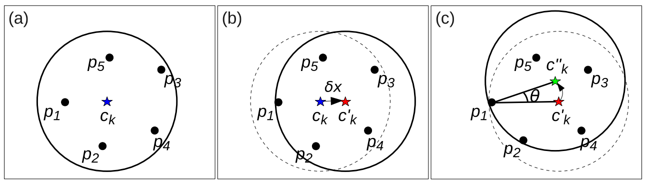

Figure 4: Finding disks to cover set of points

Theorem 1. *If for a given time instance t_i there exist a point in the space $c_k^{t_i}$ such that:*

$$\forall T_j \in f, d(p_j^{t_i}, c_k^{t_i}) \leq \epsilon/2$$

then there exists another point in the space $c'^{t_i}_k$ such that

$$\forall T_j \in f, d(p_j^{t_i}, c'^{t_i}_k) \leq \epsilon/2$$

and there are at least trajectories $T_a \in f$ and $T_b \in f$ such that

$$\forall T_j \in \{T_a, T_b\}, d(p_j^{t_i}, c'^{t_i}_k) = \epsilon/2$$

Theorem 1 states that if there is a disk $c_k^{t_i}$ with diameter ϵ that covers all trajectories in the flock f at time instance

t_i then there exists another disk with the same diameter but with different center $c'^{t_i}_k$ that also covers all trajectories covered by the first one and has at least two common points on its circumference. Theorem 1 can be easily proved by construction.

Proof Sketch. Assume that we have a disk with diameter ϵ and center c_k that covers all trajectories in the flock at given time instance t_i as shown in Figure 4(a). Assume for simplicity that there is no trajectory point on the circumference of the disk defined by c_k and ϵ, i.e. $\forall T_j \in f, d(T_j, c_k) < \epsilon/2$. We can find another disk with the same properties but with different center by using a combination of translation and rotation of the disk with center c_k. As a first step of the construction the center of the disk c_k is moved along the x axis until the first of the trajectory points inside lies on the circumference of the disk. For example in in Figure 4(b) the first point which falls on the circumference after the horizontal move of the disk center is p_1. The new center of the disk is point c'_k. All points in the flock are covered by the new disk with center c'_k and diameter ϵ. Otherwise, there would be a contradiction to the assumption that p_1 is the first point on the circumference. The next step of the construction rotates the new disk using as pivot the first point on the circumference (p_1). The disk is rotated until another point falls on its circumference. In the example of Figure 4(c) the disk is rotated until point p_2 is on the circumference of disk c''_k. All points in the flock are still covered by the new disk with center c''_k and diameter ϵ (otherwise there will be a contradiction to the assumption that p_2 is the first one to be on the circumference of the disk during the rotation process). The new disk has at least two points on its circumference (points p_1 and p_2) $\square$

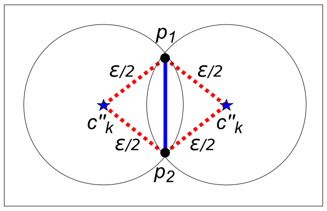

Figure 5: Disks for $\{p_1, p_2\}$, $d(p_1, p_2) \leq \epsilon$

Theorem 1 has great impact on the search for flock patterns because it limits the number of locations inside the spatial domain where to look for flocks. For a database of $|\mathcal{T}|$ trajectories there are $|\mathcal{T}|^2$ possible pairs of point combinations at any given time instance). For each such pair there are exactly two disks with radius $\epsilon/2$ that have those points on their circumference (Figure 5). We test those disks to find if they have the required minimum number of μ trajectories inside. For each time instance of the time-interval δ we have to perform $2|\mathcal{T}|^2$ tests for flock pattern. The total number of possible flock patterns that need to be tested is $2|\mathcal{T}|^{2\delta}$. As a result, the flock problem with fixed time duration has polynomial time complexity $O(|\mathcal{T}|^{|\delta|})$.

4. REPORTING FLOCK PATTERNS

In this section we describe a grid-based structure and some optimizations in order to efficiently compute the flock disk and report flocks. We also describe five on-line algorithms to process spatio-temporal data in an incremental fashion.

The grid-based structure employed for all proposed algorithms is based on grid cells with edges of ϵ distance. Each trajectory location $p^{t_i}_{id}$ reported for a specific time instance t_i is inserted in a specific grid cell. The cell is determined by its components' location latitude and longitude. Thus, each location is inserted in only one cell. The total number of cells in the index is thus affected by the trajectory distribution in the each specific time instance t_i and the ϵ. The smaller the value of ϵ, the larger number of grid cells are needed. In our implementation, grid cells that are empty, i.e. there is no trajectory location in them, are not allocated. Other structures (e.g. k-d-trees) could be employed for organizing all trajectory locations in each cell grid. However, since for small ϵ the number of locations within each cell is relatively small, and given its access simplicity we used a list for each cell. The organization of this index is shown in Figure 6.

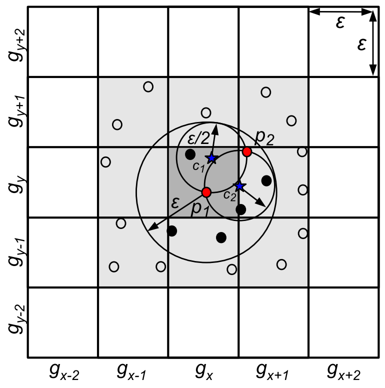

Figure 6: A grid-based index example.

Once the grid structure is built for t_i, disks can be processed using the Algorithm 1. For each grid cell $g_{x,y}$, only the 9 adjacents grid cells, including itself, are analyzed. Algorithm 1 first process every point in $g_{x,y}$ and every point in $[g_{x-1,y-1}...g_{x+1,y+1}]$ in order to find pair of points p_r, p_s whose distances satisfy: $d(p_r, p_s) \leq \epsilon$. Because all cells in the grid index have ϵ distance, there is no need to analyze points further away of the range $[g_{x-1,y-1}...g_{x+1,y+1}]$ cells for points in a particular cell $g_{x,y}$. Pairs that have not been processed yet and are within ϵ to each other are further used to compute the two disks c_1 and c_2. In case that the pairs are exactly at distance $d(p_r, p_s) = \epsilon$, c_1 and c_2 have the same center and only one has to be further processed.

It should be noted that not all points in $[g_{x-1,y-1}...g_{x+1,y+1}]$ have to be "paired" with each point in $g_{x,y}$: only those that have distance $d(p_r, p_s) \leq \epsilon$. Another optimization involves the points that have to be checked whether they are inside each disk computed in the previous step. Figure 7 illustrates these situations. For each point $p_r \in g_{x,y}$ (point p_1 in Figure 7(a)), a range query with radius ϵ is performed over all 9 grids $[g_{x-1,y-1}...g_{x+1,y+1}]$ to find points that can be "paired" with p_r, that is $d(p_r, p_s) \leq \epsilon$ holds. The result of such range search is stored in the list $\mathcal{H}$ that is used to check for each disk computed. For those valid pairs, at most 2 disks are generated. For each of them, points in the list $\mathcal{H}$ are checked if they are inside the disk (Figure 7(b)). Disks that have less than μ points are further discarded and only the ones that $|c_k| \geq \mu$ holds are kept. In Figure 7(c) disk c_1 is discarded and c_2 is considered a valid disk. Because we are interested

Algorithm 1 Computing disks in grid-based index

```
91:  $\mathcal{C} \leftarrow \emptyset$
92:  $Index.Build(\mathcal{T}[t_i], \epsilon)$                 ▷ build index for $\mathcal{T}[t_i]$
93:  for each non-empty cell $g_{x,y} \in Index$ do
94:      $P_r \leftarrow g_{x,y}$                              ▷ points in central grid
95:      $P_s \leftarrow [g_{x-1,y-1} \dots g_{x+1,y+1}]$      ▷ points in 9 grids
         centered in $g_{x,y}$
96:      if $|P_s| \geq \mu$ then                  ▷ cells have enough trajectories
97:          for each $p_r \in P_r$ do
98:              $\mathcal{H} \leftarrow Range(p_r, \epsilon)$    ▷ $d(p_r, p_s) \leq \epsilon, p_s \in P_s$
99:              for each $p_j \in \mathcal{H}$ do      ▷ compute disks in $\mathcal{H}$
910:                 if not computed $\{p_r, p_j\}$ yet then
911:                     compute disks $\{c_1, c_2\}$ defined by
                         $\{p_r, p_j\}$ and diameter $\epsilon$
912:                     for each disk $c_k \in \{c_1, c_2\}$ do
913:                         $c \leftarrow c_k \cap \mathcal{H}$      ▷ points inside disk
914:                         if $|c| \geq \mu$ then          ▷ disk qualifies
915:                             $\mathcal{C}.Add(c)$  ▷ keep maximal sets
                                 only
916: return $\mathcal{C}$                                   ▷ sets of maximal disks
```

only in maximal instances of flock patterns, a valid disk is further checked whether another disk has a superset of instances that the current disk has just computed (line 15 of Algorithm 1). In this particular case, disks that have subset of instances are discarded and only those ones stored in $\mathcal{C}$ that have the maximal instances are returned by Algorithm 1.

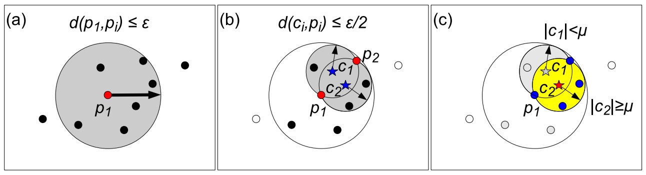

Figure 7: Steps on finding flocks for time t

The process that Algorithm 1 employs to keep only the maximal disks is based on the center of the disk and the total number of common elements that each disk has. Disks are checked only with the ones that are "close" to each other, that is, disk c_1 is checked with c_2 only if $d(c_1, c_2) \leq \epsilon$. If $d(c_1, c_2) > \epsilon$, we can safely state that they do not have any elements in common. To efficient process the operations described above, we store disks in $\mathcal{C}$ using a k-d-tree where the center of each disk along with its radius $\epsilon/2$ are stored. When checking for a particular entry c_1, we only need to check entries in the k-d-tree that "intersect" with the new one. Only those disks that cannot be pruned are further verified to check their contents. Because we store entries that belong to each disk in a binary tree, we can efficiently check if one disk has supersets/subsets elements than the other disk. Therefore, we only need to count common elements in both disks by scanning each entry in each disk once. If the cardinality of common elements are $|c_1 \cap c_2| = |c_1|$ then c_1 is subset of c_2 disk, or they have all common elements when $|c_1| = |c_2|$. Therefore, c_1 can be discarded and only c_2 is kept in $\mathcal{C}$. When $|c_1 \cap c_2| = |c_2|$, c_2 can be discarded. Otherwise we can safely say that one is not maximal than the other disk and we have to keep both c_1 and c_2 in $\mathcal{C}$.

In the following subsection we describe the basic flock pattern evaluation algorithm which combines the candidate disks generated for each time instance into flock patterns. Later in this section we describe four variations of the basic algorithm which use different filtering heuristics in order to reduce the number of candidate disks which have to be analyzed.

4.1 The Basic Flock Evaluation Algorithm

In the basic flock pattern evaluation algorithm *BFE*, we generate the candidate disks for every time instance t_i, starting with the first one t_1 and moving one time instance at a time. Every candidate disk generated in given time instance t_i is analyzed and joined with potential flocks generated in the previous time instance t_{i-1}. Only those potential flocks that are successfully augmented with disk in the current time instance are kept for further processing in the next time instance. This method reports flock patterns as soon as they satisfy the temporal constraint δ (e.g. we have at least δ candidate disks successfully joined in a flock).

As it was mentioned in previous section, we use a grid-based index to find disks for the current time instance t_i. For the first time instance t_1, all disks returned by the grid-based index are stored as potential flocks (we can view a candidate disk as a partial flock with length 1) in the list of candidate flocks for this time instance $\mathcal{F}^{t_i}$. In the following time instances all disks returned by the grid-based index are stored in their candidate flock lists $\mathcal{F}^{t_i}$ and then "joined" with the candidate flocks from the previous time instance $\mathcal{F}^{t_{i-1}}$. The "join" condition used for this operation is $|c \cap f| \geq \mu$, i.e. the total number of common elements between the candidate flock and the disk has to be greater or equal to μ in order to be joined. If the condition is satisfied then we move the join result into the list of candidate flocks for the current time instance t_i. A flock is found if there are at least δ join operations applied over the candidate flock (line 12), i.e. $u.t_{end} - u.t_{start} = \delta$. In this case, the flock pattern is immediately reported to the user and its $u.t_{start}$ attribute is updated and reinserted in $\mathcal{F}^{t_i}$ to be further joined with other disks in the following time instance.

It should be noted that $\mathcal{F}^{t_i}$ only maintains potential flocks starting at some previous time instance $t_{start} > t_i - \delta$ and ending in the current time instance $t_{end} = t_i$. Entries that cannot be joined in the next time instance are discarded and not transferred into the list of candidate flocks for the next time instance.

One advantage of the *BFE* Algorithm is that for each time instance being processed, the algorithm store only the trajectory IDs in $\mathcal{F}^{t_i}$. There is no need to keep the actual locations of moving objects in $\mathcal{F}^{t_i}$ since they do not participate in the join condition. Another advantage is that trajectory locations for each time instance are processed only once, that is, there is no need to buffer trajectory data for a time window with lenght δ like our other algorithms explained later in this section.

4.2 Filtering Heuristics

The number of candidate disks in a given time instance can be quite large and the cost to join those candidate disks in a flock pattern can be quite expensive. In order to improve the performance of the basic join algorithm we propose a

Algorithm 2 *BFE*: Basic Flock Evaluation Algorithm

```
91: $\mathcal{F}^{t_0} \leftarrow \emptyset$    ▷ initialize partial result set
92: for each new time instance $t_i$ do
93:     $\mathcal{L} \leftarrow \mathcal{T}[t_i]$    ▷ reported location for trajectories in time $t_i$
94:     $\mathcal{C} \leftarrow Index.Disks()$    ▷ set of disks for $t_i$
95:     $\mathcal{F}^{t_i} \leftarrow \emptyset$    ▷ holds potential flocks up to $t$
96:     for each $c \in \mathcal{C}$ do    ▷ join phase
97:         for each $f \in \mathcal{F}^{t_{i-1}}$ do    ▷ previous potential flocks
98:             if $|c \cap f| \geq \mu$ then    ▷ at least $\mu$
99:                 $u \leftarrow c \cap f$
910:                $u.t_{start} \leftarrow f.t_{start}$    ▷ set the initial time
911:                $u.t_{end} \leftarrow t$    ▷ set the final time
912:                if $(u.t_{end} - u.t_{start}) = \delta$ then    ▷ found a flock
913:                    report flock pattern $u$ from $u.t_{start}$ to $u.t_{end}$
914:                    update $u.t_{start}$    ▷ shift the time
915:                $\mathcal{F}^{t_i} \leftarrow \mathcal{F}^{t_i} \cup u$    ▷ add potential flock $u$
916:        $\mathcal{F}^{t_i} \leftarrow \mathcal{F}^{t_i} \cup c$    ▷ add disk $c$ to $\mathcal{F}^{t_i}$
```

set of four different heuristics used to limit the number of generated candidate disks. These heuristics are described next.

4.2.1 Top Down Evaluation

The first heuristic is a "Top Down Evaluation" (*TDE*). It differs from the basic algorithm in the fact that the construction of the flocks is not done in a bottom-up approach (by extending flock patterns one candidate disk at a time until they become at least δ time instances long) but in a top down fashion. Here we compare the candidate disks for time instances which are δ time instances apart. This is based on the assumption that the difference between the candidate disks in two consecutive time instances will be small (thus resulting in a large number of short flocks which still have to be kept as candidates until it becomes clear that they do not have the required length), while the differences between candidate disks from time instances which are δ time instances apart will be significant (and will result in smaller set of candidate flocks).

This heuristic buffers trajectory locations for time window w which has length δ time instances. It also performs a different strategy on joining the candidate disks in this time window w. First the algorithm calculates the candidate disks $\mathcal{C}^1$ for the first time instance $t_{i-\delta+1}$ in the window w. Then, disks for the last time instance t_i in w are calculated and joined with the ones in $\mathcal{C}^1$. The candidate flocks for time window w generated as a result of this step are then verified using the the basic flock pattern evaluation algorithm.

4.2.2 The Pipe Filter Evaluation

Our second heuristic, the *Pipe Filtering Evaluation* (*PFE*), also employs the filter and refine paradigm. It first filters all trajectories that have at least μ objects within distance ϵ of them for a duration of at least δ time instances. Then in a refinement step performed over the trajectories returned by the filtering step we search for flock patterns using the basic flock pattern evaluation algorithm. Figure 8 illustrates a pipe for trajectory T_2 with radius ϵ.

Algorithm 3 *TDE*: Top Down Evaluation Algorithm

```
91: for each new time instance $t_i$ do
92:     let $\mathcal{L}$ be trajectories in windows size $|w| = \delta$, from $t_{i-\delta}$ to $t_i$
93:     $\mathcal{C}^1 \leftarrow Index.Disks(\mathcal{L}[1])$    ▷ set of disks for $t_{i-\delta}$
94:     $\mathcal{C}^w \leftarrow Index.Disks(\mathcal{L}[w])$    ▷ set of disks for $t_i$
95:     $\mathcal{F} \leftarrow \emptyset$
96:     $\mathcal{U} \leftarrow \emptyset$
97:     for each $c^1 \in \mathcal{C}^1$ do    ▷ join phase
98:         for each $c^w \in \mathcal{C}^w$ do    ▷ join with flocks in $t_{i-1}$
99:             if $|c^1 \cap c^w| \geq \mu$ then    ▷ there are at least $\mu$ locations
910:                $\mathcal{U} \leftarrow \mathcal{U} \cup \{c^1 \cap c^w\}$    ▷ add it
911:    for each $u \in \mathcal{U}$ do    ▷ refinement phase
912:        $\mathcal{L}' \leftarrow u$    ▷ trajectories in $u$ from $t_{i-\delta}$ to $t_i$
913:        $\mathcal{F}^1 \leftarrow u^1$    ▷ results for $w = 1$
914:        for $t \leftarrow 2$ to $|w| - 1$ do    ▷ forward-join phase
915:            $\mathcal{C}^t \leftarrow Index.Disks(\mathcal{L}'[t])$    ▷ disks for $\mathcal{L}'[t]$
916:            $\mathcal{F}^t \leftarrow \emptyset$
917:            for each $c \in \mathcal{C}^t$ do
918:                for each $f \in \mathcal{F}^{t-1}$ do    ▷ join phase
919:                    if $|c \cap f| \geq \mu$ then
920:                        $\mathcal{F}^t \leftarrow \mathcal{F}^t \cup \{c \cap f\}$    ▷ add it for the next step
921:            if $|\mathcal{F}^t| = 0$ then break    ▷ there is a gap
922:        for each $f \in \mathcal{F}^{w-1}$ do
923:            for each $c^w \in \mathcal{C}^w$ do    ▷ join with flocks in $w$
924:                if $|f \cap c^w| \geq \mu$ then    ▷ there are at least $\mu$ locations
925:                    $\mathcal{F} \leftarrow \mathcal{F} \cup \{f \cap c^w\}$    ▷ add it
926:    report flocks $\mathcal{F}$    ▷ report flocks from $t_{i-\delta}$ to $t_i$
```

The Pipe Filtering algorithm, first builds a grid-based index for the first time instance $t_{i-\delta}$ in the w window. Then, for each trajectory location T_j in $t_{i-\delta}$ a range search is issued (line 5). The purpose of this query is to examine how many other object locations are within distance ϵ from the trajectory being processed If the cardinality of the result set is greater or equal than the threshold μ, then we continue with the same check for time instances $t_{i-\delta+1}$ to t_i (lines 8-10). If the total number of trajectories inside the "pipe" for given trajectory T_j is $|\mathcal{U}| \geq \mu$, then the trajectory qualifies and it is stored in the list of candidates $\mathcal{M}$ (lines 11-12), to be further processed in the refinement step (lines 13-23) of the algorithm.

The refinement step employs the basic flock pattern evaluation algorithm. The difference however is that now it process only the trajectory locations returned as a result of the filtering step $\mathcal{M}$ instead of using the whole trajectory database.

This approach is beneficial in cases where a large number of trajectories will be pruned by the pipe filtering step and the computationally expensive candidate disk generation and flock construction will be performed over a limited subset of trajectories.

4.2.3 The Continuous Refinement Evaluation

As the name implies, the *Continuous Refinement* heuristic continuously refines the set of trajectories which can participate in a flock pattern. This approach uses the candidate

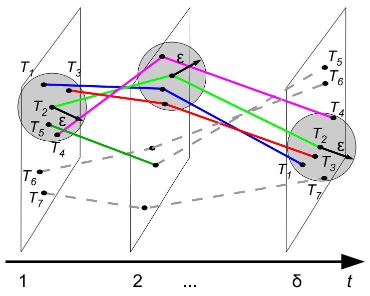

Figure 8: Pipe filtering for T_2, δ and radius ϵ.

Algorithm 4 *PFE*: Pipe Filter Evaluation Algorithm

```
91: for each new time instance $t_i$ do
92:     $\mathcal{F} \leftarrow \emptyset$
93:     let $\mathcal{L}$ be trajectories in windows size $|w| = \delta$, from $\mathcal{T}[t_{i-\delta}]$ to $\mathcal{T}[t_i]$
94:     for each $T_j \in \mathcal{L}$ do                          ▷ filter phase
95:         $\mathcal{L}' \leftarrow Index.Range(T_j, \epsilon)$   ▷ range in $\mathcal{L}[1]$ for $T_j$ and radius $\epsilon$
96:         if $|\mathcal{L}'| \geq \mu$ then ▷ $\mathcal{L}'$ has enough entries for $t_{i-\delta}$
97:             $\mathcal{U} \leftarrow \emptyset$
98:             for each $T_k \in \mathcal{L}'$ do                 ▷ pipe query
99:                 if $\forall t_i \in w, p_k^{t_i} \in T_k, p_j^{t_i} \in T_j, d(p_k^{t_i}, p_j^{t_i}) \leq \epsilon$ then
910:                    $\mathcal{U} \leftarrow \mathcal{U} \cup T_k$   ▷ $T_k$ is in the pipe $T_j$ and $\epsilon$
911:            if $|\mathcal{U}| \geq \mu$ then
912:                $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{U}$   ▷ add pipe $\mathcal{U}$ to $\mathcal{M}$
913:    for each $m \in \mathcal{M}$ do                             ▷ refinement phase
914:        $\mathcal{F}^1 \leftarrow Index.Disks(m^1)$             ▷ disks for $w = 1$
915:        for $t \leftarrow 2$ to $|w|$ do                        ▷ refinement phase
916:            $\mathcal{C} \leftarrow Index.Disks(m^t)$           ▷ disks for $t$ and $m$
917:            $\mathcal{F}^t \leftarrow \emptyset$
918:            for each $c \in \mathcal{C}$ do
919:                for each $f \in \mathcal{F}^{t-1}$ do           ▷ join phase
920:                    if $|c \cap f| \geq \mu$ then
921:                        $\mathcal{F}^t \leftarrow \mathcal{F}^t \cup \{c \cap f\}$   ▷ add it for the next step
922:            if $|\mathcal{F}^t| = 0$ then break
923:        $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}^t$          ▷ store the flocks
924:    report flocks $\mathcal{F}$         ▷ report flocks from $t_{i-\delta}$ to $t_i$
```

disk generation step for time instance t_i as a filtering step for time instance t_{i+1}. Only trajectories that are associated with the candidate disk in time t_i are analyzed in t_{i+1}. This approach can be used in cases where the selectivity of the candidate disks is high, e.g. there exists a relatively small number of candidate disks and the number of trajectories in them is low.

In its first step, the continuous refinement evaluation finds disks $\mathcal{C}^1$ using locations $\mathcal{L}[1]$ for time instance $t_{i-\delta}$. Then, for each disk $c^1 \in \mathcal{C}^1$, all trajectories associated with it are further processed from time instance $t_{i-\delta+1}$ to t_i (lines 8-16).

At the first time instance, disks $\mathcal{C}^1$ for time instance $t_{i-\delta}$ are stored in $\mathcal{F}^1$ (potential flocks of length 1 - line 7). Then, each instance of c^1 is further processed to compute disks and is "merge-joined" with the previous ones stored in $\mathcal{F}^t$. If $\mathcal{F}^t$ has no potential flock at time t, then the processing of c^1

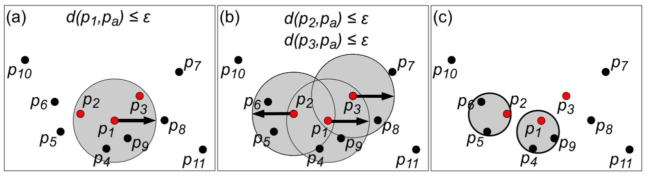

Figure 9: *CFE* steps to find flock patterns

can be discarded. After this second step, flock patterns are reported from time $t_{i-\delta}$ to t_i.

Algorithm 5 *CRE*: Continuous Refinement Evaluation Algorithm

```
91: for each new time instance $t_i$ do
92:     let $\mathcal{L}$ be trajectories in windows size $|w| = \delta$, from $t_{i-\delta}$ to $t_i$
93:     $\mathcal{C}^1 \leftarrow Index.Disks(\mathcal{L}[1])$     ▷ set of disks for $w = 1$
94:     $\mathcal{F} \leftarrow \emptyset$
95:     for each $c^1 \in \mathcal{C}^1$ do                          ▷ join phase
96:         let $\mathcal{L}'$ be the trajectories in $c^1$ with length $w$
97:         $\mathcal{F}^1 \leftarrow c^1$                           ▷ results for $w = 1$
98:         for $t \leftarrow 2$ to $|w|$ do                         ▷ forward-join phase
99:             $\mathcal{C}^t \leftarrow Index.Disks(\mathcal{L}'[t])$   ▷ disks for $\mathcal{L}'[t]$
910:            $\mathcal{F}^t \leftarrow \emptyset$
911:            for each $c \in \mathcal{C}^t$ do
912:                for each $f \in \mathcal{F}^{t-1}$ do           ▷ join phase
913:                    if $|c \cap f| \geq \mu$ then
914:                        $\mathcal{F}^t \leftarrow \mathcal{F}^t \cup \{c \cap f\}$   ▷ add it for the next step
915:            if $|\mathcal{F}^t| = 0$ then break                ▷ there is a gap
916:        $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}^t$          ▷ store the flocks
917:    report flocks $\mathcal{F}$         ▷ report flocks from $t_{i-\delta}$ to $t_i$
```

4.2.4 The Cluster Filtering Evaluation

The last proposed heuristic, *Cluster Filtering Evaluation* (*CFE*), has two phases. The first phase applies the *DBSCAN* clustering algorithm with parameters *eps*=ϵ and *minPts*=μ for for each time instance t_i. Clusters reported for a given time instance t_i by the *DBSCAN* algorithm are further joined with clusters found in the previous time instance t_{i-1}. The join criteria is that the clusters should have at least μ trajectories in common. If a cluster u can be augmented in this way for δ consecutive time instances ($u.t_{end} - u.t_{start} = \delta$), then it is saved as a candidate which has to be verified in the second phase, using the basic flock pattern evaluation algorithm (lines 13-23 of Algorithm 6).

Figure 9 illustrates the steps performed by the 6 algorithm. In Figure 9(a), the *DBSCAN* is applied to a specific object location p_1 with parameters *eps*=ϵ and *minPts*=μ. Then, in Figure 9(b), shows the propagation of the *DBSCAN* algorithm over p_1's neighbors. Object locations that do not belong to any cluster are discarded. The final two clusters reported by the *DBSCAN* algorithm in Figure 9(c) are then further processed in the refinement step of the 6 Algorithm.

5. EXPERIMENTAL EVALUATION

In order to evaluate the proposed algorithms, we run several experiments with various real and synthetic trajectorial datasets under different parameters. In particular we show the results for three real – *Trucks*, *Buses*, *Cars* and

Algorithm 6 *CFE*: Clustering Filtering Evaluation Algorithm

```
91: $\mathcal{I}^{t_i} \leftarrow \emptyset$                         ▷ density-clusters up to $t_i$
92: for each new time instance $t_i$ do
93:     $\mathcal{L} \leftarrow \mathcal{T}[t_i]$                  ▷ trajectory locations in time $t_i$
94:     $\mathcal{Q} \leftarrow DBSCAN(\mathcal{L}, \mu, \epsilon)$     ▷ cluster $\mathcal{L}$ using $DBSCAN$
95:     $\mathcal{U} \leftarrow \emptyset$                          ▷ results for $t_i$
96:     for each $q \in \mathcal{Q}$ do                      ▷ join phase
97:         for each $f \in \mathcal{I}^{t_{i-1}}$ do ▷ join with clusters in $t_{i-1}$
98:             if $|q \cap f| \geq \mu$ then     ▷ there are at least $\mu$ locations
99:                 $u \leftarrow \{q \cap f\}$
910:                $u.t_{start} \leftarrow f.t_{start}$   ▷ set the initial time
911:                $u.t_{end} \leftarrow t$             ▷ set the final time
912:                if $(u.t_{end} - u.t_{start}) = \delta$ then   ▷ potential flock
913:                    $\mathcal{F}^1 \leftarrow Index.Disks(u^1)$   ▷ disks for $w = 1$
914:                    for $t \leftarrow 1$ to $|w|$ do     ▷ refinement phase
915:                        $\mathcal{C} \leftarrow Index.Disks(m^t)$ ▷ disks for $t$ and $m$
916:                        $\mathcal{F}^t \leftarrow \emptyset$
917:                        for each $c \in \mathcal{C}$ do
918:                            for each $f \in \mathcal{F}^{t-1}$ do   ▷ join phase
919:                                if $|c \cap f| \geq \mu$ then
920:                                    $\mathcal{F}^t \leftarrow \mathcal{F}^t \cup \{c \cap f\}$
921:                        if $|\mathcal{F}^t| = 0$ then break  ▷ gap in time
922:                    $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}^t$          ▷ store the flocks
923:                    update $u.t_{start}$             ▷ shift the time
924:                $\mathcal{U} \leftarrow \mathcal{U} \cup u$
925:        $\mathcal{U} \leftarrow \mathcal{U} \cup q$                    ▷ add set $q$ to $\mathcal{U}$
926:    $\mathcal{I}^{t_i} \leftarrow U$                    ▷ $\mathcal{U}$ holds clusters up to $t_i$
```

Caribous – and one synthetic – *SG* – dataset. The *Trucks* and *Buses* [3] datasets represents 112,203 and 66,096 locations for 276 and 145 moving trucks and buses, respectively, in the metropolitan area of Athens, Greece. The *Cars* [11] represents 134,263 locations for 183 private cars moving in Copenhagen, Denmark. The *Caribous* [4] trajectorial dataset represents migration movements of 43 caribous, containing 15,796 locations, in Canada. We also used the *SG* dataset which was synthetic generated using the road network of Singapore. For this synthetic dataset, 50,000 moving objects with different velocities were randomly placed in the road network generating 2,548,084 locations.

Table 1: Parameters values for each dataset

	μ [**default**]	ϵ [**default**]	δ [**default**]
Trucks	4, 6,...,20 [**5**]	0.8, 0.9,...,1.5 [**1.2**]	4, 6,...,20 [**10**]
Buses	4, 6,...,20 [**5**]	0.4, 0.5,...,1.1 [**1.2**]	4, 6,...,20 [**10**]
Cars	4, 6,...,20 [**5**]	0.8, 0.9,...,1.5 [**1.2**]	4, 6,...,20 [**10**]
Caribous	2, 3,...,10 [**5**]	0.1, 0.2,...,0.8 [**1.6**]	4, 6,...,20 [**10**]
SG	4, 6,...,20 [**5**]	2.2, 2.6,...,5.0 [**3.4**]	4, 6,...,20 [**10**]

In our experiments we tested several values for the μ, ϵ and δ parameters. The ranges of values for each dataset are shown in Table 1, where bold values represent default values. For instance, for the *Trucks* when the μ varies from 4 to 20, with increments of 2, the default values for ϵ and δ are 1.2 and 10, respectively. The total number of patterns discovered are shown in Table 2 (we only show the minimum and maximum values). For instance, for the *Trucks* dataset the minimum value of patterns discovered for μ=4 is 309, ϵ=0.8 is 3,741, and δ=20 is 2,045 (when the parameters are very selective in the query).

Table 2: Number of flock patterns discovered

Varying	μ		ϵ		δ	
	min	*max*	*min*	*max*	*min*	*max*
Trucks	309	14,935	3,741	15,608	2,045	23,222
Buses	0	2,988	16	1,021	55	1,730
Cars	62	18,451	3,218	23,440	3,149	24,211
Caribous	124	9,480	5,292	6,915	3,364	4,598
SG	0	1,304	53	741	112	385

Figures 10-14 show the results when varying μ (first column), ϵ (second column) and δ (third column). All plots represent the total time in seconds to process the whole dataset, including building the indexes. As it can be seen, when increasing μ, decreasing ϵ, or decreasing δ, the total time to discover patterns for each algorithm also decreases. This is explained by the pruning behavior of the algorithms in their early stages (grid-based index) and thus less flock candidates have to be maintained.

For the *Trucks* and *Cars* datasets, the *CRE* and *PFE* algorithms have the best performance among all other algorithms. The large difference in performance appears when the total number of patterns reported increases (for small μ and big ϵ). This is because more intermediate results are maintained by the other three methods (*BFE*, *PFF* and *CRF*) and thus more time is needed in order to manipulate them. The same behavior occurs for big values of δ, but only for the *PFF* and *CRF* algorithms. This is due to the fact that these two algorithms aggregate trajectories in windows w before computing the disks for each timestamp, different from the other methods. For the *Buses* dataset, the *BFE*, *PFE* and *CRE* algorithms have the same performance behaviour.

For the *Caribous* dataset the *BFE* algorithm has the best performance, closely followed by the *PFE* and *CRE*. The *BFE* algorithm performed well in this dataset because the other algorithms do not prune almost any trajectories in their filtering phases. This can be corroborated by the total number of flocks discovered in Table 2 (the values are quite large for this dataset with 15,796 locations).

The *SG* dataset derived a very interesting observation. The *BFE* algorithm is by far the best algorithm for this dataset. The main reason for this behavior is that even for each timestamp the total number of potential flocks is big (see Table 3), the other four algorithms try to join flocks for two consecutive timestamps, while most of them are not possible (see the number of flocks reported in Table 2). Therefore, the first filter phase of the *BFE* has a high pruning capability than the other algorithms for the *SG* dataset. We should remark that in the real datasets, trajectories follow similar

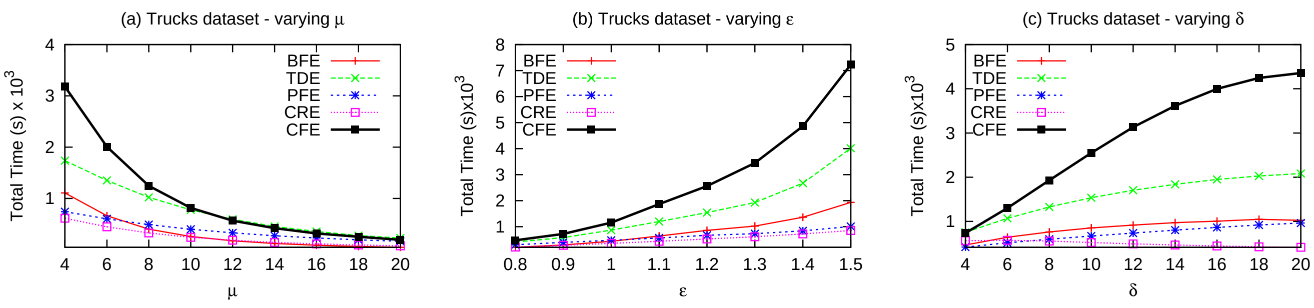

Figure 10: Total time (s) when varying (a) μ, (b) δ and (c) ϵ for the *Trucks* dataset

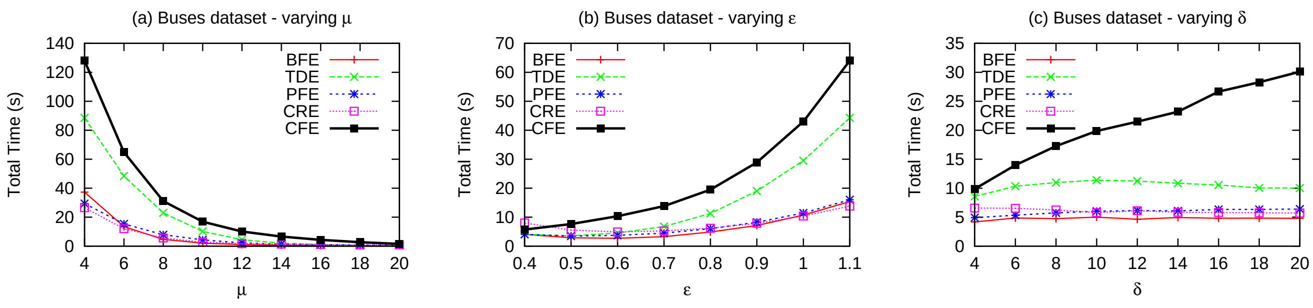

Figure 11: Total time (s) when varying (a) μ, (b) δ and (c) ϵ for the *Buses* dataset

Table 3: Number of disks per time

Varying	μ		ϵ		δ	
	min	*max*	*min*	*max*	*min*	*max*
Trucks	505	1,257	812	1,547	1,237	1,237
Buses	7	236	27	183	105	105
Cars	72	294	142	387	279	279
Caribous	393	235	587	342	309	309
SG	1,343	12,894	1,232	2,916	10,934	10,934

patterns, while for the *SG* dataset, objects are close to each other but they do not tend to follow similar patterns for consecutive timestamps.

For most of the datasets, the *CRF* algorithm had the worst performance among all algorithms. This is due to the fact that the filtering (clustering) step employed by it does not prune many trajectories as expected. In fact, because not so many trajectories are pruned to the second phase, the difference among its performance and the others are related to running its filter step.

In the next set of experiments, Table 3 shows minimum/maximum number of disks computed for the *BFE* algorithm. Similar results were obtained for the other four algorithms. The results show that even for big values of μ, ϵ and δ, the range number of disks computer per timestamp are small compared with the number of trajectories.

The last set of experiments examine the performance of the grid-based index with the brute force (processing the dataset with no use of indexes) and R-tree approaches. Figure 15 shows the total time required to build the grid-based and R-tree indexes and the total time used to process the *Trucks* dataset with those two indexes and with no index. The time to build the grid-based index is half the time for the R-tree, but with similar times to process the dataset. On the other hand, the combined time to build the index and to process the dataset is always smaller than the brute-force approach. This means that the time of building the grid-based index pays off when using it in the next phases.

6. CONCLUSIONS

Recently there has been increased interest in queries that capture the collaborative behavior of spatio-temporal data (e.g. convoys, flocks, etc.) In particular, a flock contains a group of at least μ moving objects all of them “enclosed” by a disk of diameter ϵ for at least δ consecutive time periods. Discovering flock patterns on line is useful for several applications ranging from tracking suspicious activities to migrations of animals. Previous related works either cannot apply on finding flock patterns, work only for archived datasets and/or find approximate results. We first show that flock discovery under a fixed time duration can be solved in polynomial time. We then present a framework that uses a lightweight grid-based structure in order to efficiently and incrementally process the trajectory locations. Using this framework we provide various on-line flock discovery algorithms. Experiments on real and synthetic trajectorial datasets show that our methods can efficiently report flock patterns even for large datasets and for different variations of the flock parameters (μ, ϵ and δ). As future work we will examine cost models to enable the user pick the most efficient algorithm based on the data distribution.

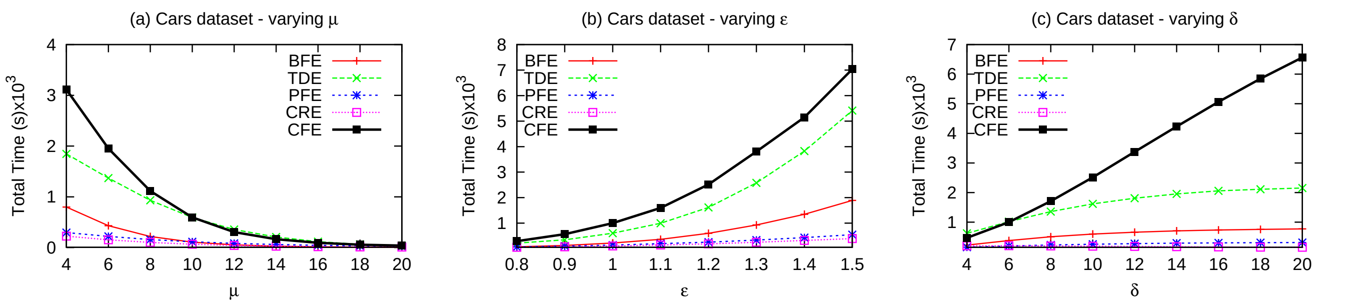

Figure 12: Total time (s) when varying (a) μ, (b) δ and (c) ϵ for the *Cars* dataset

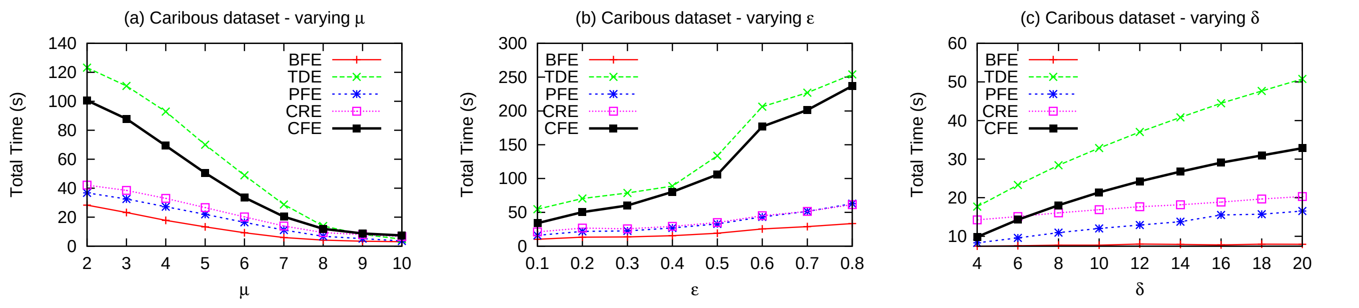

Figure 13: Total time (s) when varying (a) μ, (b) δ and (c) ϵ for the *Caribous* dataset

7. REFERENCES

[1] www.environmental-studies.de.

[2] whale.wheelock.edu.

[3] www.rtreeportal.org.

[4] www.taiga.net/satellite.

[5] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. In *Conf. on Annual European Symp.*, 2006.

[6] M. E. et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[7] M. S. et al. Monic: modeling and monitoring cluster transitions. In *KDD*, 2006.

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

[9] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *SIGMOD*, pages 73–84, 1998.

[10] C. Jensen, D. Lin, and B. Ooi. Continuous clustering of moving objects. *IEEE TKDE*, 19(9):1161–1174, 2007.

[11] C. S. Jensen. Daisy. www.daisy.aau.dk.

[12] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 2008.

[13] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.

[14] P. Laube, M. Kreveld, and S. Imfeld. Finding REMO: Detecting relative motion patterns in geospatial lifelines. *Dev. in Spatial Data Handling*, 2004.

[15] J.-G. Lee, J. Han, X. Li, and H. Gonzalez. Traclass: Trajectory classification using hierarchical region-based and trajectory-based clustering. *PVLDB*, 2008.

[16] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, 2007.

[17] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *ACM SIGKDD*, pages 617–622, 2004.

[18] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, 1994.

[19] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD*, 1996.

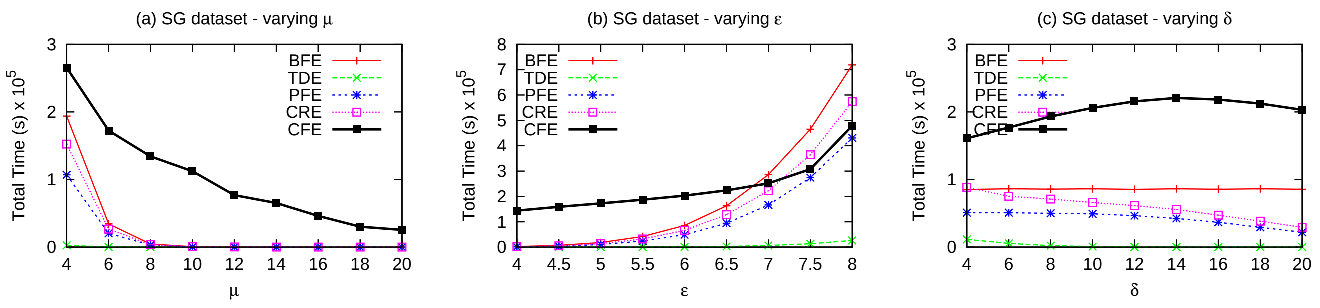

Figure 14: Total time (s) when varying (a) μ, (b) δ and (c) ϵ for the *SG* dataset

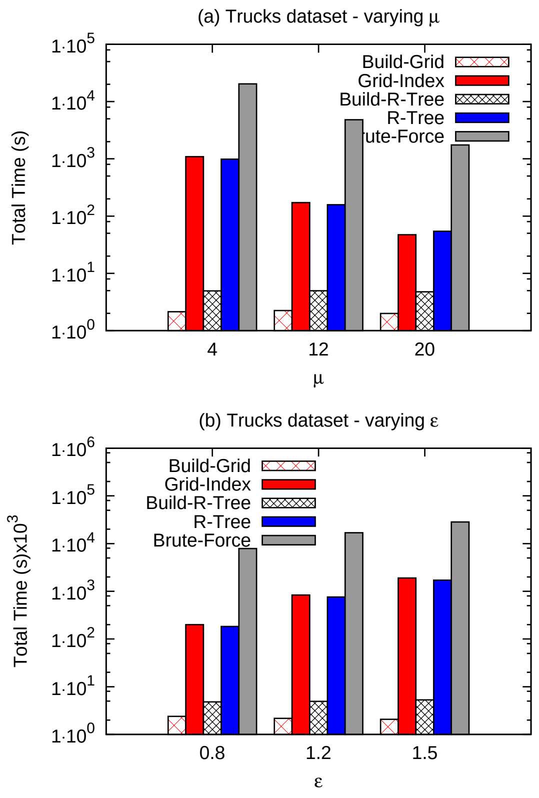

Figure 15: Indexes comparison for *Trucks*