Reporting Flock Patterns

Marc Benkert[1⋆], Joachim Gudmundsson[2], Florian Hübner[1], and Thomas Wolle[2]

[1] Department of Computer Science, Karlsruhe University, P.O. Box 6980, D-76128 Karlsruhe, Germany. {mbenkert, florianh}@ira.uka.de
[2] National ICT Australia Ltd⋆⋆, Locked Bag 9013, Alexandria NSW 1435, Australia. {joachim.gudmundsson,thomas.wolle}@nicta.com.au

Abstract. Data representing moving objects is rapidly getting more available, especially in the area of wildlife GPS tracking. It is a central belief that information is hidden in large data sets in the form of interesting patterns. One of the most common spatio-temporal patterns sought after is flocks. A flock is a large enough subset of objects moving along paths close to each other for a certain pre-defined time. We give a new definition that we argue is more realistic than the previous ones, and we present fast approximation algorithms to report flocks. The algorithms are analysed both theoretically and experimentally.

1 Introduction

Data related to the movement of objects is becoming increasingly available because of substantial technological advances in position-aware devices such as GPS receivers, navigation systems and mobile phones. The increasing number of such devices will lead to huge spatio-temporal data volumes documenting the movement of animals, vehicles or people. One of the objectives of spatio-temporal data mining [12, 14] is to analyse such data sets for interesting patterns. For example, a group of 25 elks in Sweden was equipped with GPS-GSM collars. The GPS collar acquires a position every half hour and then sends the information to a GSM-modem where the positions are extracted and stored. Analysing this data gives insight into entity behaviour, in particular, migration patterns. There are many other examples where spatio-temporal data is collected [1, 13]. The analysis of moving objects also has applications in sports (e.g. soccer players [7]), in socio-economic geography [4] and in defence and surveillance areas.

The input is a set P of n moving point objects $p_1, \ldots, p_n$ whose locations are known at τ consecutive time steps $t_1, \ldots, t_\tau$, i.e. the trajectory of each object is a polygonal line, see Fig. 1a. We will call moving point objects *entities* from now on, and assume their velocity between two consecutive time steps is constant.

There is some research on data mining of moving objects (e.g. [9, 15, 16, 18]) in particular, on the discovery of similar directions or clusters. Kalnis et al. detect moving clusters over many time steps [8]. However, their definition of

⋆ Supported by grant WO 758/4-2 of the German Science Foundation (DFG).
⋆⋆ Funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

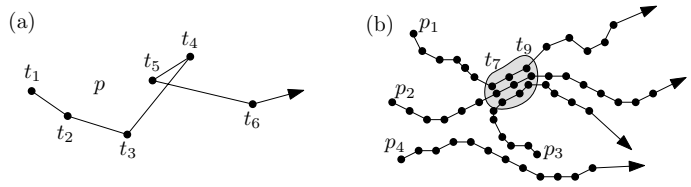

Fig. 1. (a) A polygonal line describing the movement of an entity p in the time interval $[t_1, t_6]$. (b) A flock for p_1, p_2, p_3 in the time interval $[t_7, t_9]$.

clusters does not lead to an approach that in general finds flocks. Verhein and Chawla [18] used associated data mining to detect patterns in spatio-temporal sets. They partitioned space into cells and then defined a cell to be a *source,sink* or *thoroughfare* depending on the number of objects entering, exiting or passing through the cell.

Laube and Imfeld [10] proposed a different approach in 2002 - the REMO framework (RElative MOtion) which defines similar behaviour in groups of entities. They define a collection of spatio-temporal patterns based on similar direction of motion or change of direction. Laube et al. [11] extended the framework by not only including direction of motion, but also location itself. They defined several spatio-temporal patterns, including *flock*, *leadership*, *convergence* and *encounter*, and gave algorithms to compute them efficiently. In [11] they developed an algorithm for finding the largest flock pattern (maximum number of entities) using the higher-order Voronoi diagram with running time $\mathcal{O}(\tau(nm^2 + n\log n))$, they also proved that the detection problem can be answered in $\mathcal{O}(\tau(nm + n\log n))$ time. Applying the paper by Aronov and Har-Peled [2] to the problem gives a $(1+\varepsilon)$-approximation with expected running time $\mathcal{O}(\tau n/\varepsilon^2 \log^2 n)$. Gudmundsson et al. [6] showed that if the disk is $(1+\varepsilon)$-approximated then the detection problem can be solved in $\mathcal{O}(\tau(n/\varepsilon^2 \log 1/\varepsilon + n\log n))$ time.

However, the above algorithms only consider each time step separately, that is, given $m \in \mathbb{N}$ and $r > 0$ a flock is defined by at least m entities within a circular region of radius r and moving in the same direction at some point in time. We argue that this is not enough for most practical applications, e.g. a group of animals may need to stay together for days or even weeks before they define a flock. Therefore we propose the following definition of a flock:

Definition 1. *(m, k, r)-flock$_A$ - Given a set of n trajectories where each trajectory consists of $\tau - 1$ line segments, a flock in a time interval $I = [t_i, t_j]$, where $j - i + 1 \geqslant k$, consists of at least m entities such that for every point in time within I there is a disk of radius r that contains all the m entities. Note that $m, k \in \mathbb{N}$ and $r > 0$ are given constants.*

Gudmundsson and van Kreveld [5] recently showed that (in the discrete model, see Section 2.1) computing the longest duration flock and the largest subset flock is NP-hard to approximate within a factor of $\tau^{1-\varepsilon}$ and $n^{1-\varepsilon}$ respectively. They also give a 2-radius approximation algorithm for the longest duration flock with running time $\mathcal{O}(n^2\tau \log n)$.

We describe efficient approximation algorithms for reporting and detecting flocks, where we let the size of the region deviate slightly from what is specified. Approximating the size of the circular region with a factor of $\Delta > 1$ means that a disk with radius between r and Δr that contains at least m objects may or may not be reported as a flock while a region with a radius of at most r that contains at least m entities will always be reported. We present several approximation algorithms, for example, a $(2+\varepsilon)$-approximation with running time $T(n) = \mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ and a $(1+\varepsilon)$-approximation algorithm with running time $\mathcal{O}(1/(m\varepsilon^{2k}) \cdot T(n))$.

Our aim is to present algorithms that are efficient not only with respect to the size of the input (which is τn) but also try to keep the dependency on k and m as small as possible. For most of the practical applications we have seen; m will be between a couple of entities to a few hundreds or even thousands, and k is expected to be between 5 and 30 for most applications.

The paper is organised as follows. In Section 2 we show a discrete version of the definition of a flock and prove that it is equivalent to the original definition. Three approximation algorithms (all derived from a general approach) are presented in Section 3. In the final section we discuss the implementations and experimental results. Note that due to space constraints proofs are omitted.

1.1 The skip quadtree and the computational model

One of the main tools used in this paper is the skip-quadtree by Eppstein et al. [3]. A small modification to the skip-quadtree results in the following lemma:

Lemma 1. *Insertion, deletion and search in the modified d-dimensional skip quadtree using a total of $\mathcal{O}(dn)$ space can be done in $\mathcal{O}(d \log n)$ time. An $(1+\delta)$-approximate range counting query for any fat convex region of complexity $\mathcal{O}(d)$ can be answered in time $T(n) = \mathcal{O}(d^2(\log n + 1/\delta^{d-1}))$, where $\delta > 0$ is a given constant.*

The standard practice [3] in computational geometry using quadtrees is that certain operations can be done in constant time. In arithmetic terms, the computations needed to perform point location, range queries or nearest neighbour queries in a quadtree, involve finding the most significant binary digit at which the coordinates of two points differ. This can be done using a constant number of machine instructions if we have a most-significant-bit instruction, or by using floating point or extended precision normalisation.

2 Approximate flocks

The input is a set P of n trajectories $p_1, \ldots, p_n$, where each trajectory p_i is a sequence of τ coordinates in the plane $(x_1^i, y_1^i), (x_2^i, y_2^i), \ldots, (x_\tau^i, y_\tau^i)$, and (x_j^i, y_j^i) is the position of entity p_i at time t_j. We will assume that the movement of an entity from its position at time t_j to its position at time t_{j+1} is described by the straight-line segment between the two coordinates, and that the entity moves along the segment with constant velocity.

2.1 An equivalent definition of flock

We will give an alternative and algorithmically simpler definition of a flock.

Definition 2. *(m, k, r)-flock$_B$ - Given a set of n trajectories where each trajectory consists of τ line segments a flock in a time interval $[t_i, t_j]$, where $j-i+1 \geqslant k$ consists of at least m entities such that for every discrete time step t_ℓ, $i \leqslant \ell \leqslant j$, there is a disk of radius r that contains all the m entities.*

Lemma 2. *If the entities move with constant velocity along the straight line segment between two consecutive time steps then flock$_A$ and flock$_B$ are equivalent.*

Note that the centre of a disk does not have to coincide with one of the positions of the entities. In the remainder of this paper we refer to Definition 2 whenever we talk about flocks. Definition 2 immediately suggests a new approach; for each time interval $[t_i, t_{i+k-1}]$ check whether there is a set of m entities $F = \{p_1, \ldots, p_m\}$ that can be covered by a disk of radius r at each discrete time step in $[t_i, t_{i+k-1}]$. We will show how this observation will allow us to develop an approximation algorithm.

2.2 The general approach

When developing an algorithm for this problem one of the main hurdles that we encountered was to detect flocks without having to keep track of all the objects in a potential flock. That is, when we consider a specific time step, the number of potential flocks can be very large and the number of objects that one needs to keep track of for each potential flock might be $\Omega(n)$. In general this problem occurs whenever one attempts to develop a method that processes the input time step by time step. In this paper we avoid this problem by transforming the trajectories into higher dimensional space. Note that the gain is that we only need to count the number of points in a region, instead of keeping track of the actual objects. This might seem like overkill but both the theoretical and the experimental bounds supports this approach, at least as long as k is fairly small.

The basic idea builds upon the fact that a polygonal line with d vertices in the plane can be modelled as a single point in $2d$ dimensions. The trajectory of an entity p in the time interval $[t_i, t_j]$ is described by the polygonal line $p(i, j) = \langle (x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j) \rangle$, which corresponds to a point $p'(i, j) = (x_i, y_i, x_{i+1}, y_{i+1}, \ldots, x_j, y_j)$ in $2(j - i + 1)$-dimensional space.

The first step when checking whether there is a flock in the time interval $[t_i, t_{i+k-1}]$ is to map the polygonal lines of all entities to $\mathbb{R}^{2k}$. Equivalence 1 gives the key characterisation of flocks. First, we define an (x, y, i, r)-*pipe* which is an unbounded region in $\mathbb{R}^{2k}$. Such a pipe contains all the points that are only restricted in two of the $2k$ dimensions (namely in dimensions i and $i + 1$) and when projected on those two dimensions lie in a circle of radius r around the point (x, y). Formally, a (x, y, i, r)-pipe is the following region:

$$\big\{(x_1, \ldots, x_{2k}) \in \ \mathbb{R}^{2k} \mid \ (x_i - x)^2 + (x_{i+1} - y)^2 \leqslant r^2\big\}.$$

Equivalence 1 *Let $F = \{p_1, \ldots, p_m\}$ be a set of entities and $I = [t_1, t_k]$ a time interval. Let $\{p'_1, \ldots, p'_m\}$ be the mapping of F to $\mathbb{R}^{2k}$ w.r.t. I. It holds that: F is a (m, k, r)-flock $\iff \exists x_1, y_1, \ldots, x_k, y_k : \forall p \in F : p' \in \bigcap_{i=1}^{k}(x_i, y_i, 2i - 1, r)$-pipe.*

To see that this equivalence holds we observe the following: for each time step $t_i \in I$ the disk with radius r and centre (x_i, y_i) contains the entity positions $p_1^i, \ldots, p_m^i$. We will show that approximation algorithms can be obtained by performing a set of range counting queries in higher dimensional space.

3 Approximation algorithms

We now give approximation algorithms where the radius r is approximated. A Δ-approximation (with $\Delta > 1$) here means that every (m, k, r)-flock will be reported, an $(m, k, \Delta r)$-flock may or may not be reported, while no (m, k, r')-flock where r' exceeds Δr will be reported.

Method 1: A $(\sqrt{8}+\varepsilon)$-approximation algorithm (box). Using the general idea discussed in Section 2.2 we will develop a $(\sqrt{8}+\varepsilon)$-approximation algorithm. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we will do the following computations. For simplicity our first method uses a $2k$-dimensional box to approximate the region of a potential flock.

For each entity p let p' denote the mapping of p to $\mathbb{R}^{2k}$ with respect to I. Construct a skip quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Then, for each point $p' \in P'$ and an appropriately chosen $\delta > 0$ perform a $(1 + \delta)$-approximate range counting query in T where the query range $Q(p')$ is a $2k$-dimensional cube. As we do not know the centre of a potential flock we choose to query around p' and any flock that contains p is within distance $2r$ from p. Hence, our query box has side length $4r$ and centre at p'. Roughly spoken, we $(1+\delta)$-approximate the $2k$-dimensional cube which is itself a $\sqrt{8}$-approximation for the query region (see Fig. 2a). Every counting query containing at least m entities corresponds to an $(m, k, (\sqrt{8} + \varepsilon)r)$-flock as stated in Lemma 3. Note that the same flock may be reported several times.

Lemma 3. *Method 1 is a $(\sqrt{8}+\varepsilon)$-approximation algorithm and requires $\mathcal{O}(\tau n)$ space and $\mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ time.*

Method 2: A $(2 + \varepsilon)$-approximation algorithm (pipes). The algorithm is similar to the above algorithm. The main difference is that we will use the intersection of k pipes as the query regions instead of the $2k$-dimensional box. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we will do the following computations.

For each entity p let p' denote the mapping of p to $\mathbb{R}^{2k}$ with respect to I. Construct a skip quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Then, for each point $p' \in \mathcal{P}'$ perform a $(1 + \varepsilon)$-approximate range counting query in T where

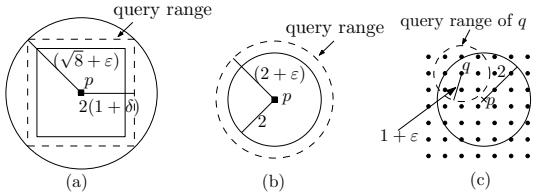

Fig. 2. Illustration of the approximative range of methods 1,2 and 3 for $r = 1$. The dashed region is the query region.

the query range $Q(p')$ is the intersection of the k pipes $(x_i, y_i, 2i-1, 2r)$ where (x_i, y_i) is the position of entity p at time step t_i (see Fig. 2b).

Recall that since the query region is convex and fat we can apply Lemma 1. The definition of fatness we use was introduced by van der Stappen [17].

Definition 3 ([17]). *Let $\alpha > 1$ be a real value. An object s is α-fat if for any d-dimensional ball D whose centre lies in s and whose boundary intersects s, we have* $\text{volume}(D) \leqslant \alpha \cdot \text{volume}(s \cap D)$.

Lemma 4. *The intersection of d pipes $(x_i, y_i, 2i-1, 2r)$, $1 \leqslant i \leqslant k$, in $2d$-dimensional space is a bounded convex 4^d-fat region whose boundary consists of $\mathcal{O}(d)$ surfaces of quadratic complexity.*

Lemma 5. *Method 2 is a $(2+\varepsilon)$-approximation algorithm and requires $\mathcal{O}(\tau n)$ space and $\mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ time.*

Remark 1. A comparison between Lemmas 3 and 5 shows that even though the approximation factor of the second method is smaller the running time is identical. However, this is a theoretical bound, in practice we chose to implement the second method using a compressed quadtree, because for using a skip-quadtree, we need to compute the volume between a d-dimensional cell and the intersection of the k pipes, which is possible in theory but hard in practice. Consequently, the experiments performed with methods 1 and 2 use different data structures.

Method 3: A $(1+\varepsilon)$-approximation algorithm. We use the same approach as above but instead of querying only the input points in $\mathbb{R}^{2k}$ we will now query $\mathcal{O}(1/\varepsilon^{2k})$ sample points for each entity point. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we will do the following computations.

For each entity p let p' denote the mapping of p to $\mathbb{R}^{2k}$ with respect to I. Construct a skip quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Let Γ be the vertices of a regular grid in $\mathbb{R}^{2k}$ of spacing $\varepsilon \cdot r/2$. Each input point p'_i generates the sample set $\Gamma \cap D(p'_i)$ where $D(p'_i)$ is the $2k$-dimensional ball of radius $2r$ centred at p'_i. Clearly, this gives rise to $\mathcal{O}(1/\varepsilon^{2k})$ sample points for each entity (see Fig. 2c).

Next, we perform a $(1+\frac{\varepsilon}{2+\varepsilon})$-approximate range counting query in T for each sample point $(x_1, y_1, \ldots, x_k, y_k)$ where the query range is the intersection of the

k pipes $(x_i, y_i, 2i-1, (1+\varepsilon/2)r)$, $1 \leqslant i \leqslant k$. However, a necessary condition for a sample point q to induce an (m,k,r)-flock is that there are at least m entities in the disk $D(q)$ of radius $2r$ centred at q. During the processing of the sample points we can count how many entities indeed lie in $D(q)$ for each sample point q. As we generate at most $\mathcal{O}(n/\varepsilon^{2k})$ sample points, this means that we have to check at most $\mathcal{O}(n/(m\varepsilon^{2k}))$ candidate sample points for inducing a flock. Next we prove the approximation bound.

Lemma 6. *Method 3 is a $(1+\varepsilon)$-approximation algorithm and requires $\mathcal{O}(\tau n)$ space and $\mathcal{O}(\frac{\tau n k^2}{m\varepsilon^{2k}}(\log n + 1/\varepsilon^{2k-1}))$ time.*

4 Experiments

We used a Linux operated off-the-shelf PC with an Intel Pentium-4 3.6 GHz processor and 2 GB of main memory. The data structures and algorithms were implemented and compiled with the Gnu C++ compiler. Our point sets used in the experiments were created artificially. Each point coordinate of an input point is taken from the interval $[0, ..., 2^{13}]$ or $[0, ..., 2^{16}]$. The point sets differ in size (10,000 - 160,000 points; one algorithm was run with more than 1 million points), in length of the time interval (4 - 16 time steps) and also in the distribution of the points (uniformly random or clustered). In all point sets, 10% of the points were placed in such a way that they form (randomly positioned) flocks of $m = 50$ entities in a circle of radius $r = 50$. The distribution and density of the clusters were chosen not to considerably increase the number of flocks found by the algorithms. This makes a comparison between the results for clustered and uniformly randomly distributed point sets easier. Note that each generated data instance contains the coordinates of points for a certain number of time steps τ, and in the experiments on that instance, we always looked for (m,k,r)-flocks with $m = r = 50$ and $k = \tau$.

4.1 Methods

We compare the results of four methods called 'box', 'pipes', 'no-tree' and 'pruning'. The box and pipes method are explained in Section 3 and use a skip-quadtree or a compressed quadtree, respectively. The no-tree method (which was implemented for comparison) is a 2-approximation and does not use a tree structure. It has two nested loops (each running over all input points), the outer one specifying a potential flock centre and the inner one computing the distance between a point and the potential flock centre. If there are enough points within a ball (around the potential flock centre) of double flock-radius then we found a flock. The pruning method takes advantage of the fact that all points not involved in flocks of length $k^* < k$ cannot be involved in flocks of length k. It works as follows: At first, we compute flocks of length $k = 4$ using the box method. Then we build a new tree containing only those points that were contained in

flocks during the first step. This drastically reduces the number of points. We then again perform the box method on the new tree for the entire length $k = \tau$.

A set of entities can have many flocks and even one single entity can be involved in several flocks, e.g. a flock involving $m + 1$ entities trivially contains $m + 1$ flocks of cardinality m. We must specify what we want to find and report in a given data set, see [6] for a discussion. The general approach described in Section 3 has the following disadvantage: As every entity is tested, a flock consisting of exactly m elements can be reported up to m times. We chose to use an approach in the experiments which guarantees a high level of correctness while bounding the number of flocks that an entity may simultaneously belong to. The idea is that when a flock is found every query point within the query region will be marked, so that no query will be performed with those marked points as centres. Using a simple packing argument it follows that the maximal number of flocks an entity can be part of during a time step is bounded by $\mathcal{O}(2^{2k})$. The additional time that we have to spend updating the tree is $\mathcal{O}(nk \log n)$ per time step, thus $\mathcal{O}(\tau nk \log n)$ in total. The number of reported flocks is bounded by $2^{2k} \cdot n$ per time step.

4.2 Results

We run the experiments with a series of generated point-sets for each combination of point-set characteristics. The results were very similar for fixed characteristics and hence the tables below show the numbers for only one collection of point-sets with the specified characteristics. The results of the algorithms for $\varepsilon = 0.05$ are depicted in Table 1, where the coordinates of the points are chosen from the interval $[0, ..., 2^{16}]$. The columns below 'input' specify the number of points and the number of time steps, and the columns below 'uniformly' and 'clustered' show the number of flocks found (our algorithms also output the size and the centre of those flocks) and the running times (in seconds) needed when performing the box-, pipes- and no-tree-algorithm on the corresponding input. We also performed the same experiments on point-sets where the coordinates where chosen from $[0, ..., 2^{13}]$. Table 2 shows those results. The results for the method with pruning are given in Table 3. Because of the similarity of the results for a different number of time steps, we only report the results for 16 time steps in that table. Note that we artificially inserted flocks and report their numbers (indicated in *italics* if it deviates from the number of inserted flocks) merely to ensure and verify that our methods indeed find them. The dependencies of the running times are a more important result than the number of flocks.

4.3 Discussion

Flat trees in high dimensions. One obvious observation is that the running times of our algorithms are increasing with the number of time steps (i.e. with the number of spatial dimensions d). Recall that an internal node of an quadtree has 2^d children. Using 16 time steps means 32 dimensions which translates to more than 4 billion quadrants, i.e. children of an internal node (in our approach

input		uniformly						clustered					
		box		pipes		no-tree		box		pipes		no-tree	
size	τ	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	0	20	0	20	5	20	0	20	1	20	5
10K	8	20	2	20	1	20	5	20	1	20	0	20	5
10K	16	20	2	20	1	20	6	20	1	20	0	20	5
20K	4	40	1	*41*	0	40	21	40	0	40	1	40	20
20K	8	40	7	40	5	40	21	40	1	40	0	40	22
20K	16	40	13	40	10	40	25	40	3	40	2	40	25
40K	4	80	0	80	1	80	83	80	1	80	0	80	83
40K	8	80	32	80	22	80	87	80	2	80	2	80	87
40K	16	80	62	80	44	80	99	80	8	80	7	80	101
80K	4	160	3	*163*	3	160	332	160	3	160	2	160	332
80K	8	160	129	160	88	160	347	160	6	160	4	160	346
80K	16	160	244	160	182	160	392	160	30	160	29	160	392
160K	4	320	8	*321*	10	320	1326	320	8	320	5	320	1327
160K	8	320	441	320	316	320	1391	320	20	320	15	320	1384
160K	16	320	986	320	768	320	1576	320	102	320	93	320	1564

Table 1. Results for $\varepsilon = 0.05$ and point-sets with coordinates from $[0, ..., 2^{16}]$.

we only store non-empty children in a list, which reduces storage space but increases time complexity). In an experiment with 160K points in 32 dimensions it is not likely that many of the random points (not in flocks) fall into the same quadrant. Therefore the tree is very flat, which results in high running times.

Error value ε. When performing a range query, ε influences the approximate region to be queried. One could expect that a larger value of ε leads to shorter running times and more flocks, because the descent in the tree can be stopped earlier. However, apart from marginal fluctuations, this behaviour could not be observed in our experiments. Our trees in the experiments are rather sparsely filled. Hence, the squares corresponding to most of the leaves in the tree (which correspond to single points in a point set) are still quite large compared to the approximated flock radius $(1 + \varepsilon)r$. Furthermore, it often seems that the point sets are too sparse to find any random flocks. Therefore we refrained from reporting results for different ε and only used $\varepsilon = 0.05$.

Number of flocks. Most of the times the algorithms found exactly as many flocks as were artificially inserted. A few times more flocks were found but only in instances with a small number of time steps, which is reasonable as it is more likely for random points to form flocks only for a small number of time steps. In one case more than 1300 flocks were found which indicates that for that instance and for that characteristic the distribution of the points and clusters reached a limit were the clusters are dense enough to often create random flocks. In some of our experiments we observed that the algorithms found less flocks than were inserted. This can happen if two flocks are close to each other and fall into one query region and hence will be counted as one flock by the algorithm.

Coordinate space $[0, ..., 2^{13}]$ *vs.* $[0, ..., 2^{16}]$. The experiments with coordinate space $[0, ..., 2^{16}]$, i.e. where each coordinate of a point is in $[0, ..., 2^{16}]$, were comparatively much faster than those with coordinate space $[0, ..., 2^{13}]$. One explanation is that the query region is relatively larger in the coordinate space $[0, ..., 2^{13}]$. Also, in a bigger underlying coordinate space it is more likely that the query region falls into a single quadrant of a quadtree. Due to the sparseness of the point-sets the algorithms are likely to find just a single point in that quadrant. On the other hand in a smaller underlying space the query region might intersect more quadrants, which results in more subsequent queries.

input		uniformly						clustered					
		box		pipes		no-tree		box		pipes		no-tree	
size	τ	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	1	20	0	20	5	20	2	20	1	20	4
10K	8	20	8	20	6	20	6	20	2	20	2	20	6
10K	16	20	14	20	11	20	6	20	5	20	11	20	6
20K	4	40	1	40	4	40	20	40	2	40	1	40	20
20K	8	40	52	40	35	40	22	40	6	40	4	40	22
20K	16	40	83	40	58	40	25	40	17	40	44	40	25
40K	4	80	4	80	15	80	83	*81*	6	80	2	80	83
40K	8	80	237	80	166	80	87	80	16	80	21	80	87
40K	16	80	347	80	244	80	99	80	55	80	177	80	99
80K	4	160	10	160	57	160	333	*206*	16	160	8	160	332
80K	8	160	932	160	696	160	348	160	45	160	77	160	348
80K	16	160	1411	160	1124	160	394	160	164	160	594	160	395
160K	4	320	29	320	201	320	1326	*1317*	42	320	27	320	1331
160K	8	320	3179	320	2658	320	1393	320	124	320	238	320	1392
160K	16	320	6015	320	4226	320	1575	320	692	320	2306	320	1576

Table 2. Results for $\varepsilon = 0.05$ and point-sets with coordinates from $[0, ..., 2^{13}]$.

Uniformly vs. clustered. We can observe that our tree-based algorithms almost always perform better on the clustered point-sets. This behaviour could be expected because, as we have seen from the experiments in general, uniformly distributed points result in quadtrees that are rather flat, i.e. have only a very small depth (especially for higher dimensions). But it is a 'good balance' between height and width of a tree that allows fast query times. Clustered data sets are more likely to create trees that are faster descended by the algorithms. The no-tree method (which is not using a tree) is not affected by the two different types of data.

No-tree vs. box vs. pipe. The no-tree method's running times are quadratic in the number of points and not influenced by the number of time steps, as expected. On the other hand the box and pipes algorithms are strongly influenced by the number of time steps and the number of points. A large query region in combination with a small coordinate space causes their behaviour to become

similar (although with a big overhead) to the no-tree method. The difference between the box and pipes method is caused by the different data structure they use. That is why the pipe method is almost always faster than the box method.

Pruning. Table 3 shows the impressive impact of the pruning step. Depending on the density and distribution, even some point-sets with more than 1 million points can be dealt with in a couple of minutes. Furthermore, we observed that the number of time steps has almost always no effect on the running times. An exception to this are the clustered point sets with many points and with coordinates in $[0, ..., 2^{13}]$, where we experienced much longer running times. (Because of space restrictions, we only give the numbers for 16 time steps.) This can be explained by noting that after the pruning step it is likely that the remaining points form a flock also for more time steps (as intended). Therefore, almost every query to the datastructure leads to finding a flock and hence, the number of queries is drastically decreased. For the clustered point sets with coordinates in $[0, ..., 2^{13}]$, however, the probability of random flocks is higher, because the query region is comparatively large. The fact that the pruning method sometimes finds less flocks than the box method can be explained by noting that the pruning method performs two runs of the box method each of which can handle the points in a different order. Therefore the second run of the box method can encounter points which will not belong to any flock.

input		coordinates from $[0, ..., 2^{13}]$				coordinates from $[0, ..., 2^{16}]$			
		uniformly		clustered		uniformly		clustered	
		pruning		pruning		pruning		pruning	
size	τ	flocks	time	flocks	time	flocks	time	flocks	time
10K	16	20	0	20	1	20	1	20	0
20K	16	40	1	40	2	40	1	40	0
40K	16	80	3	80	6	80	2	80	2
80K	16	160	11	160	15	160	3	160	3
160K	16	320	30	320	45	320	9	320	9
320K	16	*639*	82	*633*	303	640	26	640	25
640K	16	*1271*	194	*1268*	1796	1280	75	1280	75
1280K	16	*2501*	533	*2507*	9213	2560	249	2560	246

Table 3. Results for pruning method, $\varepsilon = 0.05$.

5 Conclusions and open problems

This paper is a first step towards practical algorithms for finding spatio-temporal patterns, such as flocks, encounters and convergences. Future research does not only include more efficient approaches to compute these patterns but also more complicated patterns, e.g. hierarchical patterns or repetitive patterns. In this paper we have presented different algorithms for finding flock patterns and analysed them theoretically as well as experimentally. From experiments we have seen that our algorithms can perform very well. However, their running times

depend very much on the characteristics of the input point-sets, which motivates more research and experiments, preferably on real-world data.

References

1. Wildlife tracking projects with GPS GSM collars, 2006. `http://www.environmental-studies.de/projects/projects.html`.
2. B.Aronov and S. Har-Peled. On approximating the depth and related problems. In *Proc. of 16th ACM-SIAM Symposium on Discrete Algorithms*, pp. 886–894, 2005.
3. D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proc. 21st ACM Symposium on Computational Geometry*, pp. 296–305, 2005.
4. A.U. Frank, J.F. Raper, and J.-P. Cheylan, editors. *Life and motion of spatial socio-economic units*. Taylor & Francis, London, 2001.
5. J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in spatio-temporal data. Manuscript, April 2006.
6. J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal sets. In *Proceedings of the 13th International Symposium of ACM Geographic Information Systems*, 2004.
7. S. Iwase and H. Saito. Tracking soccer player using multiple views. In *Proc. of the IAPR Workshop on Machine Vision Applications (MVA02)*, pp. 102–105, 2002.
8. P. Kalnis, N. Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *Proceedings of the 9th International Symposium on Spatial and Temporal Databases (SSTD05)*, vol. 3633, Lecture Notes in Computer Science, pp. 364–381. Springer-Verlag, 2005.
9. G. Kollios, S. Sclaroff, and M. Betke. Motion mining: discovering spatio-temporal patterns in databases of human motion. In *Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2001.
10. P. Laube and S. Imfeld. Analyzing relative motion within groups of trackable moving point objects. In *GIScience 2002*, vol. 2478, Lecture Notes in Computer Science, pp. 132–144. Springer, Berlin, 2002.
11. P. Laube, M. van Kreveld, and S. Imfeld. Finding REMO – detecting relative motion patterns in geospatial lifelines. In *Developments in Spatial Data Handling: Proceedings of the 11th International Symposium on Spatial Data Handling*, pp. 201–214, 2004.
12. H.J. Miller and J. Han, editors. *Geographic Data Mining and Knowledge Discovery*. Taylor & Francis, London, 2001.
13. Porcupine caribou herd satellite collar project. `http://www.taiga.net/satellite/`.
14. J. Roddick, K. Hornsby, and M. Spiliopoulou. An updated bibliography of temporal, spatial, and spatio-temporal data mining research. In *TSDM 2000*, vol. 2007, Lecture Notes in Artificial Intelligence, pp. 147–163. Springer, Berlin, 2001.
15. C.-B. Shim and J.-W.Chang. A new similar trajectory retrieval scheme using k-warping distance algorithm for moving objects. In *Proc. of the 4th International Conference on Advances in Web-Age Information Management, (WAIM 2003)*, vol. 2762, Lecture Notes in Computer Science, pp. 433–444. Springer, Berlin, 2003.
16. N. Sumpter and A. J. Bulpitt. Learning spatio-temporal patterns for predicting object behaviour. *Image Vision and Computing*, 18(9):697–704, 2000.

17. A. F. van der Stappen. *Motion Planning amidst Fat Obstacles.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1994.
18. F. Verhein and S. Chawla. Mining spatio-temporal association rules, sources, sinks, stationary regions and thoroughfares in object mobility databases. In *Proc. of the Workshop on Temporal Data Mining: Algorithms, Theory and Applications*, 2005.