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, fhuebner}@ira.uka.de
[2] NICTA Sydney[⋆⋆⋆], 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, where a pattern can be any configuration of some moving objects in a certain area and/or during a certain time period. 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 by the use of techniques from computational geometry we present fast algorithms to detect and report flocks. The algorithms are analysed both theoretically and experimentally.

Keywords: moving point objects, trajectories, spatio-temporal data, computational geometry

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 [16, 17] is to analyse such data sets for interesting patterns. For example, a herd of 25 moose in Sweden was equipped with GPS-GSM collars [5]. 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 (see e.g. [3, 10]). The analysis of moving objects also has applications in sports (e.g. soccer players [11]), in socio-economic geography [7] and in defence and surveillance areas.

We will model a set of moving objects by a set P of n moving point objects $p_1, \ldots, p_n$ whose locations are known at τ consecutive time-steps $t_1, \ldots, t_\tau$ that is, the trajectory of each object is a polygonal line that can self-intersect, see Fig. 1a. For brevity, we will call moving point objects *entities* from now on. It is assumed that the positions are sampled synchronously for all entities, and that an entity moves between two consecutive positions along a straight line with constant speed.

There is some research on data mining of moving objects (e.g. [12, 18–20]) in particular, on the discovery of similar directions or clusters. Verhein and Chawla [20] used associated data mining to detect patterns in spatio-temporal sets.

Laube and Imfeld [13] 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. [14]

[⋆] This article was originally published in 'Computational Geometry – Theory and Applications', 2007.

[⋆⋆] Supported by grant WO 758/4-2 of the German Science Foundation (DFG). Parts of this work were done while M. Benkert visited NICTA on DAAD grant D/05/08276.

[⋆⋆⋆] Funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

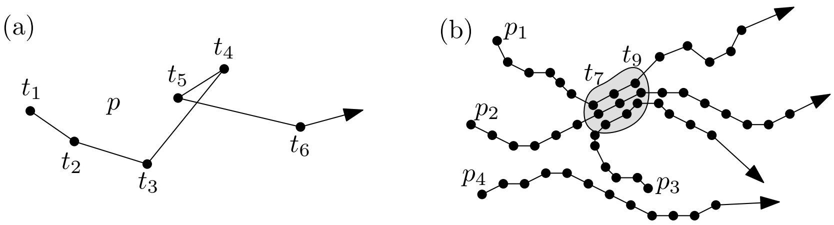

Fig. 1. (a) A polygonal line describing the movement of an entity p in the time interval $[t_1, t_6]$. (b) A flock for p_1, p_2, p_3 in the time interval $[t_7, t_9]$.

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.

Laube et al. [14] developed an $\mathcal{O}(\tau(nm_{\max}{}^2 + n\log n))$ time algorithm for finding the flock with the largest number of entities, denoted $m_{\max}$, using the higher-order Voronoi diagram. They also proved that the detection problem can be answered in $\mathcal{O}(\tau(nm + n\log n))$ time, i.e. deciding if there is a flock containing m entities. Gudmundsson et al. [9] showed that if the disk (i.e. the region in which the entities have to be in order to form a flock) 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 use a definition of a flock that considers 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 it is defined as a flock. Therefore we propose the following definition of a flock:

Definition 1. *(m,k,r)-flock$_A$ – Let $m, k \in \mathbb{N}$, and let $r > 0$ be a constant. Consider a set of trajectories, where each trajectory consists of τ 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.*

In this model, Gudmundsson and van Kreveld [8] recently showed that 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 with time-complexity $\mathcal{O}(n^2\tau\log n)$ for computing the longest duration flock.

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 algorithm running in $T(n) = \mathcal{O}(kn(2^k \log n + k^2/\varepsilon^{2k-1}))$ time and a $(1+\varepsilon)$-approximation algorithm with running in $\mathcal{O}(1/m\varepsilon^{2k}) \cdot T(n)$ time.

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

In this model a set of entities can have many flocks and even one single entity can be involved in several flocks. For example, 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 [9] for a discussion. One possibility is simply to detect whether a flock occurs. If so, we may want to report one example of a flock. Secondly, we may want to find all flocks that occur. Thirdly, we may want to report the largest size subset of entities that form a flock. In this paper we deal with the variant of finding all flocks, in Section 4 we will discuss the other variants briefly.

This paper is organised as follows. Next we give a brief description of the skip-quadtree structure used in this paper together with a description of the computational model used. In Section 2 we give a discrete version of the definition of a flock and prove that it is equivalent to the original definition provided that the entities move with constant velocity between consecutive time-steps. Furthermore, we describe our general approach to detect flocks. Then, in Section 3, we give three approximation algorithms which are all based on the general approach. In Section 4 we discuss different ways of pruning the set of flocks reported, and in the final section we discuss the implementations and the experimental results.

1.1 The computational model

One of the main tools used in this paper is the skip-quadtree presented by Eppstein, Goodrich and Sun [6] in 2005. As is nowadays standard [4, 6] in computational geometry, we assume that certain operations on quadtrees or octrees can be done in constant time. 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 two 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.

1.2 Skip-quadtree

In a later section, we will show that approximation algorithms can be obtained by performing a set of range counting queries in higher dimensional space. There are several data structures supporting this type of query: quadtrees, skip-quadtrees, octrees, kd-trees, range trees, BBD-trees, BAR-trees and so on (see e.g. [15]). For our requirements we could use either skip-quadtrees or BBD-trees, and since the implementation of the randomised skip-quadtree is very simple we chose to use the skip-quadtree.

The skip-quadtree uses the compressed quadtree as the bottom-level structure. The standard compressed quadtree for d dimensions uses $\mathcal{O}(2^d \cdot n)$ space and the worst-case height is $\mathcal{O}(n)$. We briefly describe the structure and show how to modify the structure so that it uses $\mathcal{O}(dn)$ space while the query time will increase by an $\mathcal{O}(d)$-factor.

The following is the original description of a compressed quadtree taken from [6]. Consider the standard quadtree T of the input set S with n points. We may assume that the centre of the root square (containing the set S) is the origin and the half side length for any square in T is a power of 2. Define an interesting square of a quadtree to be one that is either the root or that has at least two non-empty quadrants. Any quadtree square p in T containing at least two points contains a unique largest interesting square q in T. The compressed quadtree explicitly only stores the interesting squares, thus removing all the non-interesting squares and deleting their empty children. So for each interesting square p, they store the bounding box of p and 2^d bi-directed pointers, one for each d-dimensional quadrant. If the quadrant contains at least two points, the pointer goes to the largest interesting square inside the quadrant; if the quadrant contains one point, the pointer goes to that point; and if the quadrant is empty the pointer is NULL.

The above description of a compressed quadtree implies that the size of the tree is $\mathcal{O}(2^d \cdot n)$. We will modify the tree in the following way. Instead of storing information about which children contain points and which children are empty, we use a list that contains only the non-empty children. This improves the space complexity to $\mathcal{O}(dn)$, however this modification will increase the cost of a search in the tree since deciding if a child exists or not requires $\mathcal{O}(d)$ time.

The skip-quadtree supports $(1+\delta)$-approximate range (counting) queries, i.e. the query range Q is approximated by an extended query range Q_δ. The extended query range Q_δ consists of Q and all points within a distance $\delta \cdot w$ from Q, where w is the diameter of Q. The approximate query counts all points in Q, it either counts or does not count points in $Q_\delta \setminus Q$ and it does not count any point in $\mathbb{R}^d \setminus Q_\delta$.

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. A $(1+\delta)$-approximate range counting query for any convex region of complexity $\mathcal{O}(d)$ can be answered in time $T(n) = \mathcal{O}(d(2^d \log n + d^2/\delta^{d-1}))$, where $\delta > 0$ is a given constant.*

Proof. The time bounds stated regarding insertion, deletion and searching follow from the above discussion. Here we will focus on the bound stated for approximate range counting queries.

Let R be a query region and let A be the set of points outside R within distance $\delta \cdot w$ of R, where w is the diameter of R. That is, the union of R and A defines the extended query range. For example, if R is a d-dimensional ball of diameter w then A is the hyper-annulus of thickness $\varepsilon \cdot w$ surrounding R.

Consider a node ν of the skip-quadtree and let $s(\nu)$ be the d-dimensional cube represented by ν. We call a node ν in T *stabbing* if the d-dimensional region associated with ν intersects both R and $\overline{R \cup A}$.

In order to answer an approximate range query we only need to expand each stabbing node since all other nodes immediately can be included or excluded. At first glance the number of stabbing nodes can be $\mathcal{O}(n)$ since all the nodes in a tree path might be nested stabbing squares. However, Eppstein et al. [6] observed that one only needs to consider the smallest one in each path to answer the approximate range query. A *critical node* is a stabbing node whose child nodes are either not stabbing, or stabbing but cover less volume of R than p does. The d-dimensional cube corresponding to a critical node ν is said to be a *critical square.*

In Theorem 10 in [6], Eppstein et al. argue that $(1+\delta)$-approximate range queries can be answered in time proportional to the time it requires to search for the critical nodes. The critical nodes are partitioned into two sets, big and small. The big critical squares have diameter greater than w and only $O(2^d)$ of these can intersect R. It takes $O(d \log n)$ time to search for each of them. For the small critical squares of side length less than w, the searching time for all of them will not exceed the total number of stabbing squares of side length less than w [6]. Since the query region is convex, using (almost exactly) the same analysis as Arya and Mount in the proof of Lemma 3 in [2], we can bound the number of critical squares by $O(d^2/\delta^{d-1})$.

Adding up the time and multiplying with the time it requires to search one node we get $\mathcal{O}(d(2^d \log n + d^2/\delta^{d-1}))$, as stated in the lemma. This is similar to the bound obtained by Arya and Mount in [2]. □

Note that these bounds are expected bounds as well as bounds with high probability for the randomized version and worst-case bounds for the deterministic version of the skip-quadtree.

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 $(x_1^i, y_1^i), (x_2^i, y_2^i), \ldots, (x_\tau^i, y_\tau^i)$, where (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

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

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

Note that the centre of a disk does not have to coincide with one of the positions of the entities, see for example the disk D_5 in Fig. 2.

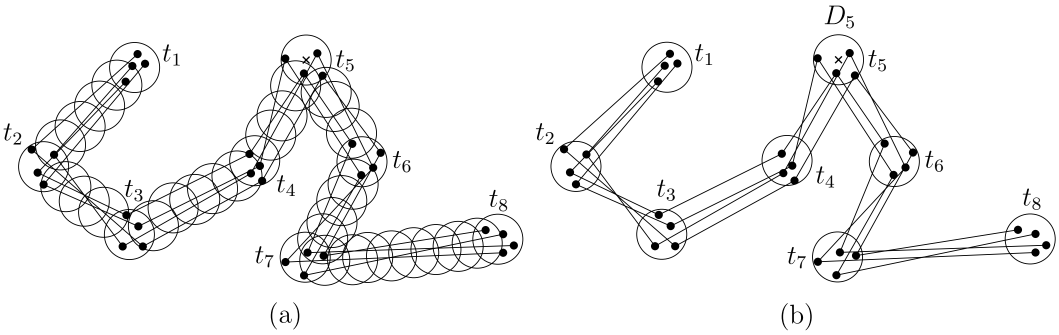

Fig. 2. A flock of four entities in the time interval $[t_1, t_8]$, according to the definitions of (a) flock$_A$ and (b) flock$_B$.

Lemma 2. *If the entities move with constant velocity along straight line segments between consecutive positions, then flock$_A$ and flock$_B$ are equivalent.*

Proof. Consider a given time interval $I = [t_1, t_k]$ and assume that F_A and F_B are the set of all flocks in I according to Definition 1 and 2 respectively. Obviously every flock $f_A \in F_A$ is also a flock in F_B, thus $F_A \subseteq F_B$.

It remains to prove that $F_B \subseteq F_A$. Let f_B be an arbitrary flock in F_B, and let D_ℓ and $D_{\ell+1}$ be disks of radius r that include the entities of f_B at time t_ℓ and $t_{\ell+1}$ respectively, see Fig. 3a. It is enough to consider every two discrete time-steps t_ℓ and $t_{\ell+1}$ in I separately.

Next we prove that at every point in time $\gamma \in I' = [t_\ell, t_{\ell+1}]$ there is a disk $\mathcal{D}_\gamma$ that contains all the entities $\{p_1, \ldots, p_m\}$ in f_B. Let c_ℓ and $c_{\ell+1}$ be the centres of D_ℓ and $D_{\ell+1}$ respectively, and let h be the straight-line segment with endpoints at c_ℓ and $c_{\ell+1}$, as illustrated in Fig. 3a. An entity q that moves with constant velocity on h has a well-defined position at time $\gamma \in I'$, we denote this position by c_γ. Next we show that the disk $\mathcal{D}_\gamma$ with centre at c_γ and radius r contains all the entities of f_B. Let p_i be an arbitrary entity of f_B. Since both, the movement of q and the movement of p_i during I' follows a straight-line and since both move with constant velocity the relative trajectory of p_i in relation to q is a straight-line as shown in Fig. 3b. Since a disk is convex and since $p_i(t_\ell)$ and $p_i(t_{\ell+1})$ are points within D_ℓ and $D_{\ell+1}$, respectively, it holds that $p_i(\gamma)$ must lie within $\mathcal{D}_\gamma$. Consequently, $f_B \in F_A$ and therefore $F_B \subseteq F_A$ which completes the proof of the lemma. □

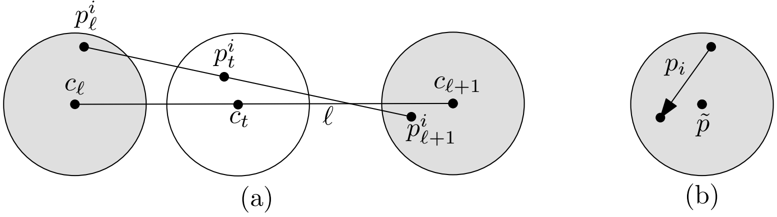

Fig. 3. Illustration for the proof of Lemma 2.

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 $[t_i, t_{i-1+k}]$. Next we will show how this observation allows 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. We then build a tree structure for every possible start time $1, ..., \tau$ and flock length k from scratch, and we then perform counting queries in this tree structure. This might seem like overkill, but both the theoretical bounds and the experimental bounds support this approach, as long as k is fairly small. 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 entities.

The basic idea is to model a 2-dimensional polygonal line with d vertices as a single point in $2d$ dimensions. Formally, 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 we map to the single point $p'(i, j)$ in $2(j - i + 1)$-dimensional space:

$$p'(i, j) = (x_i, y_i, x_{i+1}, y_{i+1}, \ldots, x_j, y_j).$$

Let p_1 and p_2 be two entities. Now, a key characterisation that will help to find flocks is that two entities p_1 and p_2 are close to each other during the time interval $[t_i, t_{i+k-1}]$ if and only if the two points $p'_1(i, i + k - 1)$ and $p'_1(i, i + k - 1)$ are close to each other in $\mathbb{R}^{2k}$. Therefore, the first step when checking whether there is a flock in the time interval $[t_i, t_{i+k-1}]$ is to map the corresponding polygonal lines of all entities to $\mathbb{R}^{2k}$. We now 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:

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

Two entities p_1 and p_2 have distance at most r at time-step $j \in \{i, i + k - 1\}$ if and only if there exists a pipe with radius r for the dimensions corresponding to j such that the $2k$-dimensional points $p'_1(i, i + k - 1)$ and $p'_2(i, i + k - 1)$ are contained in that pipe. Hence, two entities p_1 and p_2 have distance at most r at time-steps $i, ..., i + k - 1$ if and only if there exist k pipes with radius r for the dimensions corresponding to $i, ..., i + k - 1$ such that the $2k$-dimensional points $p'_1(i, i+k-1)$ and $p'_2(i, i+k-1)$ are contained in all those pipes. This leads to a characterisation of the flock-pattern, which is formalised by the following equivalence.

Equivalence 1 *Let $F = \{p_1, \ldots, p_m\}$ be a set of entities and let $I = [t_1, t_k]$ be a time interval. Let $p'_1, \ldots, p'_m$ be the mappings of the trajectories of the entities in F to $\mathbb{R}^{2k}$ w.r.t. I. It holds that:*

$$F \text{ is a } (m, k, r)\text{-flock} \iff \exists x_1, y_1, \ldots, x_k, y_k : \ \forall p \in F : \ p' \in \bigcap_{i=1}^{k} (x_i, y_i, 2i - 1, r)\text{-pipe}$$

3 Approximation algorithms

We will now show how approximation algorithms can be obtained by performing a set of range counting queries in higher dimensional space. These algorithms approximate the flock radius r. Here, a Δ-approximation (with $\Delta > 1$) means that every (m, k, r)-flock will be reported, an $(m, k, \Delta r)$-flock may or may not be reported, while no $(m, k, \hat{r})$-flock will be reported, where $\hat{r} > \Delta r$.

3.1 Method 'box': A $(\sqrt{8}+\varepsilon)$-approximation algorithm

By Equivalence 1 it is fairly straight-forward to 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 each entity p let p' denote the mapping of the trajectory of p to $\mathbb{R}^{2k}$ with respect to I. We 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 $\delta>0$ we perform a $(1+\delta)$-approximate range counting query in T, where the query range $Q(p')$ is a $2k$-dimensional cube of side length $4r$ and centre at p'. That is, we approximate the $2k$-dimensional cube which is itself an approximation for the query region. Every such query region containing at least m entities corresponds to an $(m,k,\sqrt{8}+\varepsilon)$-flock as Lemma 3 will show. Note that the same flock may be reported several times. We call the method 'box', since the query region is a box.

Lemma 3. *The algorithm is a $(\sqrt{8}+\varepsilon)$-approximation algorithm.*

Proof. First we show that each (m,k,r)-flock f is reported by the algorithm. Let p_f be an arbitrary entity of f and assume that f is a flock in the time interval $I=[t_i,t_{i+k-1}]$. We will prove that the approximation algorithm returns an $(m,k,(\sqrt{8}+\varepsilon)r)$-flock g such that $f\subseteq g$.

According to Definition 2 there exists a disk D_l with radius r that contains the entities in f for each discrete time-step t_l in I. The algorithm performs a counting query for each point in P' w.r.t. $[t_i,t_{i+k-1}]$, in particular for p'_f. The query range $Q(p'_f)$ is a $2k$-dimensional cube of side length $4r$ and centre at p'_f, where p'_f is the point in $2k$-dimensions corresponding to p_f. For a discrete time l, the query range corresponds to a square Q' in two dimensions with centre at p and side length $4r$, where the dimensions mark the $x-$ and $y-$positions of the entities at time l. As every entity of f has distance at most $2r$ to p_f this implies that every entity in f lies within $Q(p'_f)$. Thus, when p_f is queried, the algorithm reports an $(m,k,(\sqrt{8}+\varepsilon)r)$-flock g such that $f\subseteq g$.

To establish the approximation bound we still have to show that no (m,k,r')-flock g where r' exceeds $(\sqrt{8}+\varepsilon)r$ is reported. Let g be a reported flock w.r.t. the time interval $I=[t_i,t_{i+k-1}]$. We have to show that for every time-step t_l in I there exists a disk of radius $(\sqrt{8}+\varepsilon)r$ that contains the entities in g. This follows trivially by the choice of δ. If we choose δ to be $\varepsilon/\sqrt{8}$, the square of side length $4(1+\delta)r$ is contained in the disk with radius $(\sqrt{8}+\varepsilon)r$ centred at p'_f, as illustrated in Fig. 4a. This completes the proof of the lemma. □

Lemma 4. *The algorithm reports at most τn $(m,k,(\sqrt{8}+\varepsilon)r)$-flocks. It runs in $\mathcal{O}(\tau nk(2^k\log n+k^2/\varepsilon^{2k-1}))$ time and requires $\mathcal{O}(\tau n)$ space.*

Proof. The number of reported flocks is trivially bounded by n, the number of entities, times τ, the number of time-steps. At each of the $(\tau-k+1)$ time intervals the algorithm builds a skip-quadtree of the n elements from scratch. In total this requires $\mathcal{O}(\tau kn\log n)$ time, according to Lemma 1. Then a $(1+\delta)$ counting query is performed for each of the n entities; each query requires $\mathcal{O}(k(2^k\log n+k^2/\varepsilon^{2k-1}))$ time as $\delta=\frac{\varepsilon}{\sqrt{8}}$. Hence, the total time needed to perform all the $n(\tau-k')$ queries is bounded by $\mathcal{O}(\tau nk(2^k\log n+k^2/\varepsilon^{2k-1}))$ dominates the running time as stated in the lemma.

The space needed to build the skip-quadtree for each time interval is $\mathcal{O}(kn)$, and since we only maintain one tree at a time the bound follows. □

3.2 Method 'pipe': A $(2+\varepsilon)$-approximation algorithm

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 the trajectory of p to $\mathbb{R}^{2k}$ with respect to I. We construct a skip-quadtree T for the point set $P'=\{p'_1,\ldots,p'_n\}$. Then, for each point $p'\in P'$ we perform a $(1+\varepsilon)$-approximate range counting query in T, where 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. We call the method 'pipe', since the query region is the intersection of pipes.

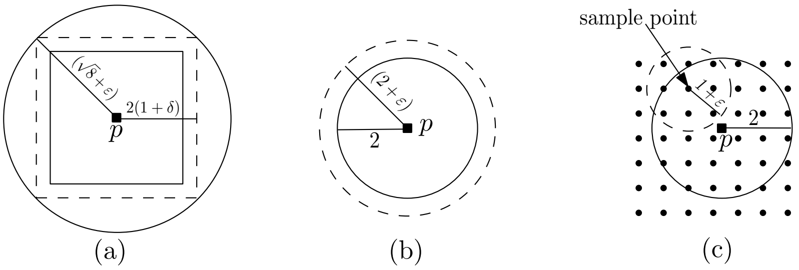

Fig. 4. Illustration of the query regions of methods box (a), pipe (b) and sample-points (c) for $r = 1$. The approximative query ranges are marked by dashed lines.

Lemma 5. *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 region whose boundary consists of $\mathcal{O}(d)$ surfaces of quadratic complexity.*

Proof. W.l.o.g. we assume that the centre of the intersection $\mathcal{I}$ of the d pipes is the origin, then $\mathcal{I}$ can be described by the following d inequalities:

$$\begin{gathered} x_1^2 + x_2^2 \leqslant r^2 \\ x_3^2 + x_4^2 \leqslant r^2 \\ \cdots \\ x_{d-1}^2 + x_d^2 \leqslant r^2. \end{gathered}$$

The set of inequalities together with the fact that the inequalities are pairwise independent immediately gives that $\mathcal{I}$ is bounded, convex and its boundary consists of $\mathcal{O}(d)$ surfaces of quadratic complexity. □

Recall that since the query range is convex we can use the result stated in Lemma 1.

Lemma 6. *The algorithm is a $(2+\varepsilon)$-approximation algorithm.*

Proof. The proof follows from the same arguments as used in the proof of Lemma 3. When approximately evaluating the query range $Q(p')$ which is the intersection of the k pipes $(x_i, y_i, 2i-1, 2r)$, $1 \leqslant i \leqslant k$ where (x_i, y_i) is the position of entity p at time-step t_i, we test whether there is an $(m, k, (2+\varepsilon)r)$-flock which p is part of. If p is part of an (m, k, r)-flock f in the time interval I, the disk with radius r containing all the entities in f at time-step $t_i \in I$ is contained in the disk with radius $2r$ centred at (x_i, y_i). Thus, when querying p, the algorithm reports an $(m, k, (2+\varepsilon)r)$-flock g with $f \subseteq g$. □

Lemma 7. *The algorithm reports at most τn $(m, k, (2+\varepsilon)r)$-flocks. It runs in $\mathcal{O}(kn(2^k \log n + k^2/\varepsilon^{2k-1}))$ time and requires $\mathcal{O}(\tau n)$ space.*

Proof. The number of reported flocks is trivially bounded by n, the number of entities, times τ, the number of time-steps. At each of the $(\tau-k+1)$ time intervals the algorithm builds a skip-quadtree of the n elements from scratch. In total this requires $\mathcal{O}(\tau kn \log n)$ time, according to Lemma 1. Next a counting query is performed for each point in $\mathcal{P}'$; each query requires $\mathcal{O}(k(2^k \log n + k^2/\varepsilon^{2k-1}))$ time, thus the total time needed to perform all the n queries is bounded by $\mathcal{O}(kn(2^k \log n + k^2/\varepsilon^{2k-1}))$ and thus dominates the running time as stated in the lemma.

The space needed to build the skip-quadtree for each time interval is $\mathcal{O}(kn)$, and since we only maintain one tree at a time the bound follows. □

Remark 1. A quick comparison between Lemmas 4 and 7 reveals that even though the approximation factor of the second method is smaller the running time is identical. However, this is a theoretical bound, for our experiments we chose to implement the second method using a compressed quadtree. The reason for this is that the skip-quadtree computes the volume of the intersection between a d-dimensional cell (orthogonal box) and $Q(p')$, where $Q(p')$ is the intersection of the k pipes, which is possible in theory but hard in practice. The query data structure of a compressed quadtree only checks whether the intersection is non-empty which is much easier to implement. Consequently, the experiments performed with methods 'pipe' and 'box' use a different query data structure.

3.3 Method 'sample-points': 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 $I = [t_i, t_{i+k'}]$, where $1 \leqslant i \leqslant \tau - k + 1$ and $k' = k - 1$, we will do the following computations.

For each entity p let p' denote the mapping of the trajectory 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 $\mathit{\Gamma}$ be the intersection points of a regular grid in $\mathbb{R}^{2k}$ of spacing $\varepsilon \cdot r/2$. Each input point p'_i generates the sample set $\mathit{\Gamma} \cap D(p'_i)$, 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 p. We call the method 'sample-points', since the query region is based on sample points.

Now, we perform a $1+\varepsilon/(2+\varepsilon)$-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 8. *The algorithm is a* $(1+\varepsilon)$*-approximation algorithm.*

Proof. The $1+\varepsilon/(2+\varepsilon)$-approximation of the range query ensures that no (m,k,r')-flock with $r' > (1+\varepsilon)r$ is reported: as we query pipes of radius $(1+\varepsilon/2)r$, the maximum distance from a grid query point to a counted entity could be $(1+\varepsilon/2) \cdot (1+\varepsilon/(2+\varepsilon))r = (1+\varepsilon)r$.

Next, we show that each (m,k,r)-flock is reported by the algorithm. Assume that f is an (m,k,r)-flock in the time interval I. We prove that the approximation algorithm returns an $(m,k,(1+\varepsilon)r)$-flock g such that $f \subseteq g$.

Let $(x_1, y_1, \ldots, x_k, y_k) \in \mathbb{R}^{2k}$ be a point that induces an (m,k,r)-flock f with respect to I. We look only at one time-step $t_i \in I$. By the cell spacing it is obvious that there are sample points $(\ldots, x_i^q, y_i^q, \ldots) \in \mathit{\Gamma}$ such that the Euclidean distance from (x_i^q, y_i^q) to (x_i, y_i) is less than $\varepsilon r/2$. This means that the disk (in $\mathbb{R}^2$) with radius $(1+\varepsilon/2)r$ centred at q completely contains the disk with radius r centred at (x_i, y_i). Thus, when checking the sample points $(\ldots, x_i^q, y_i^q, \ldots)$ all entities of f are in range for time-step t_i. As this holds analogously for all other time-steps the algorithm reports an $(m,k,(1+\varepsilon)r)$-flock g such that $f \subseteq g$. □

Lemma 9. *The algorithm reports at most* τn $(m,k,(1+\varepsilon)r)$*-flocks. It runs in* $\mathcal{O}(\frac{\tau n k}{m\varepsilon^{2k}}(2^{2k}\log n + k^2/\varepsilon^{2k-1}))$ *time and requires* $\mathcal{O}(\tau n)$ *space.*

Proof. The number of reported flocks is trivially bounded by n, the number of entities, times τ, the number of time-steps. At each of the $(\tau-k+1)$ time intervals the algorithm builds a skip-quadtree of the n elements from scratch. In total this requires $\mathcal{O}(\tau k n \log n)$ time, according to Lemma 1. Next a counting query is performed for each of the $\mathcal{O}(n/(m\varepsilon^{2k}))$ candidate sample points in $\mathit{\Gamma}$; each query requires $\mathcal{O}(k(2^{2k}\log n + k^2/\delta^{2k-1}))$ time, thus the total time needed to perform all $n(\tau-k+1)$ queries is as stated in the lemma.

The space needed to build the skip-quadtree for each time interval is $\mathcal{O}(kn)$, and since we only maintain one tree at a time the bound follows. □

4 Minimise the number of reported flocks

The general (theoretical) 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. This may get even worse if a flock is found whose number of entities exceeds m. Below we briefly discuss three approaches how reporting this redundant information could be avoided. The main idea for all of them is to prune the number of reported flocks; the last approach abandons the restriction that a flock defining region always has to be a disk.

Each entity is part of at most one flock. In theory one object can be part of many flocks at the same time which, in practice, seems unreasonable. Thus, the first method we propose guarantees that an object belongs to at most one flock at a time.

The strategy for this approach is very simple. If a counting query reports a flock then the entities involved in the flock are marked and the skip-quadtree is updated so that the marked entities will not be counted again. 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 trivially bounded by $\tau n/m$.

Each entity is part of at most a constant number of flocks. The above approach minimises the number of reported flocks; however, it also overlooks a lot of flocks. Therefore we chose to use a different approach in the experiments which guarantees a higher level of correctness while bounding the number of flocks that an entity may belong to simultaneously.

The idea is that when a flock is found every input point within the query region will be marked, so that no query will be performed with those 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.

Extending flocks that have been found. In such an approach we also assume that each entity can only be part of at most one flock. Once a flock is found, we first check whether we can extend it, which means we may manipulate the disk as flock-defining region if it seems reasonable to join objects that are close. There are many ways to do this that work in practice, however, guaranteed theoretical bounds are hard to prove.

5 Experiments

In this section, we report on the performed experiments. We describe the experimental setup, i.e. the hard- and software used for the experiments, we briefly explain the methods and we present and discuss the running times of them with respect to different parameters of the input.

5.1 Setup

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 in C++ and compiled with the Gnu C++ compiler. All our trajectories used in the experiments were created artificially. Each trajectory of length k was generated as a single point in $2k$ dimensions. Hence, we focus from now on on the characteristics of these higher dimensional point sets that were generated. 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; one algorithm was run with data with 1000 time-steps), in the size of the underlying universe (coordinates from $\left[0, ..., 2^{13}\right)$ or $\left[0, ..., 2^{16}\right)$) and also in the distribution of the points (uniformly random or clustered).

To ensure that our algorithms indeed find flocks, we arranged 10% of the points in each point-set in such a way that they form randomly positioned flocks. Each of the flocks has $m = 50$ entities in a circle of radius $r = 50$ (hence the number of artificially inserted flocks is 0.002 times the total number of points). As it is unlikely to have flocks that were generated by accident, we inserted those artificial flocks to make sure that the methods correctly find them.

The remaining 90% of the points were randomly distributed, either uniformly or in clusters. Although it is improbable it is possible for these points to interfere or extend our artificial flocks. The purpose of the clustered point sets is that they are more likely to resemble real data, and hence it is interesting to compare the impact of different distributed point sets on the running times of our methods. The number of clusters and the number of points per cluster were chosen to be roughly equal. Each cluster was generated by choosing uniformly at random a cluster centre. We then distributed the points for that cluster around this cluster centre by choosing uniformly at random an angle around the cluster centre and a distance from the cluster centre. This distance was computed with a Gaussian distribution (with mean 0.0 and standard deviation 1.0) multiplied by 0.02 times the size of the universe. However, the number, distribution, radius and density of the clusters was chosen that it is unlikely (although it can happen) to create flocks by accidents; i.e. a cluster is in general not dense enough to form a flock because its radius is much larger than the flock radius. Choosing the clusters in this way makes a comparison between the results for clustered and uniformly randomly distributed point sets easier, as the differently distributed points can have a strong effect on the height and width of the created tree structures.

Each point coordinate of an input point is an integer taken from the interval $\big[0, ..., 2^{13}\big)$ or $\big[0, ..., 2^{16}\big)$, respectively. We chose these two different sizes of the underlying universe to be able to compare the results for data sets with different overall densities.

Note that each generated data set contains the coordinates of points for a certain number of time-steps τ, and in the experiments on that instance, we always looked for flocks of at least $m = 50$ entities in a circle with radius $r = 50$ and of length k with $k = \tau$.

When performing a range query, ε influences the approximate region to be queried. One could expect that a larger value of ε can lead to shorter running times and more flocks that are found, because the descent in the tree can be stopped earlier and the query region can become larger. However, apart from very marginal fluctuations, this behaviour could not be observed in initial experiments. Our point sets and therefore 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 flock radius r and also to $(1 + \varepsilon)r$. 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 use $\varepsilon = 0.05$.

5.2 Methods

We compare the results of four methods called 'box', 'pipe', 'no-tree' and 'pruning'. All of them mark points that were found to belong to a flock, and in the further course of the algorithm those marked points are not used as a potential flock centre, see Section 4 for a discussion. (The output of the algorithms are the number and centres of the found flocks, as well as the running times and number of performed queries.) The box and pipe method are named after their query region and are explained in Sections 3.1 and 3.2, respectively.

The no-tree method (which was implemented for the sake of comparison) does not use a tree as underlying structure. It contains two nested loops, the outer one (running over all input points) specifying a potential flock centre and the inner one (running again over all input points) 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 (see the proof of Lemma 3 for an explanation why the radius is doubled) then we found a flock. Hence, the no-tree method is a 2-approximation.

The pruning method takes advantage of the fact that each flock of a certain length k is also a flock of length $k^* < k$. Therefore all points not involved in flocks of length k^* cannot be involved in flocks of length k. The method works as follows. As a first step we compute flocks of length

$k^* = 2$ time-steps 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 apply the box method on the new tree for the entire length k.

5.3 Results

We run the experiments with a couple of generated point-sets for each combination of point-set characteristics, such as number of points, number of time-steps and point distribution. 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 are depicted in Table 1, where the coordinates of the points are chosen from the interval $\big[0, ..., 2^{16}\big)$. 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 and the running times in seconds needed when performing the box-, pipe- and no-tree-algorithm on the corresponding input. We also performed the same experiments on point-sets where the coordinates where chosen from $\big[0, ..., 2^{13}\big)$. 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. Table 4 shows the results of the no-tree method for a large number of time-steps and a small number of entities. All tables show the results, i.e. the number of flocks found and the running time in seconds, only for $\varepsilon = 0.05$, because no big influence of different values of ε could be observed. From our point of view the running times are much more important than the number of flocks found. Hence, the number of flocks are shown here only for the sake of completeness. These numbers are indicated in *italics* in case they deviate from the number of artificially inserted flocks. (In most cases the methods found exactly as many flocks as were artificially inserted.)

5.4 Discussion

Flat trees in high dimensions. One general observation is that the running times of our algorithms are increasing with the number of time-steps (i.e. with the number of dimensions). Recall that an internal node of an octree has 2^d children where d is the number of dimensions. 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 we only store non-empty children in a list, which reduces storage space but increases time complexity). In an experiment with 160,000 points in 32 dimensions it is very unlikely that many of the randomly distributed points (not in flocks) fall into the same quadrant. Therefore the tree is very flat, i.e. has only a very small depth, which results in high running times.

Number of flocks. Most of the times the algorithms found exactly as many flocks as were artificially put into the point-sets. A few times more flocks were found but only in instances with a small number of time-steps, which is reasonable since if the points that are not belonging to an artificially inserted flock, form a flock at all, then it is more likely that this happened for only a small number of time-steps.

One case is remarkable (see Table 2, on clustered points, $n = 160K$, $k = 4$), where the box method found 913 flocks while the pipe method found only 320 flocks. This difference can be explained by recalling that the volume of the box query region is strictly larger than the volume of the pipe query region, and the ratio between these two volumes increases exponentially with the number of dimensions. The large number of flocks found by the box method indicates that for that particular instance the distribution of the points and clusters (in combination with a high number of points and a small coordinate space) is dense enough to form random flocks of radius at most $(\sqrt{8} + \varepsilon)r$. This could be confirmed in experiments, where we used the same data, but removed all points that make up the artificial flocks (for the above mentioned data set, 593 flocks were found, all of which must have been accidentally generated).

input		uniformly						clustered					
		box		pipe		no-tree		box		pipe		no-tree	
n	k	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	0	20	0	20	5	20	1	20	0	20	5
10K	8	20	1	20	1	20	5	20	0	20	0	20	5
10K	16	20	3	20	2	20	6	20	1	20	1	20	5
20K	4	40	0	40	0	40	21	40	1	40	1	40	20
20K	8	40	8	40	5	40	21	40	1	40	0	40	22
20K	16	40	13	40	9	40	24	40	3	40	3	40	24
40K	4	80	1	80	1	80	83	80	1	80	1	80	82
40K	8	80	34	80	22	80	86	80	3	80	1	80	86
40K	16	80	61	80	39	80	98	80	13	80	12	80	98
80K	4	160	2	160	3	160	327	160	3	160	2	160	327
80K	8	160	127	160	86	160	345	160	8	160	5	160	345
80K	16	160	235	160	172	160	391	160	47	160	39	160	391
160K	4	320	8	320	9	320	1306	320	7	320	5	320	1306
160K	8	320	438	320	313	320	1372	320	26	320	21	320	1377
160K	16	320	981	320	705	320	1558	320	148	320	122	320	1552

Table 1. Results for $\varepsilon = 0.05$ and a large coordinate space, where trajectories are sparsely distributed, i.e. the position-coordinates are in $[0, ..., 2^{16})$. The number of flocks is reported and the running time (in seconds).

input		uniformly						clustered					
		box		pipe		no-tree		box		pipe		no-tree	
n	k	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	1	20	1	20	4	20	0	20	0	20	5
10K	8	20	8	20	5	20	5	20	3	20	2	20	5
10K	16	20	14	20	10	20	6	20	6	20	11	20	6
20K	4	40	2	40	3	40	20	40	2	40	1	40	20
20K	8	40	52	40	35	40	21	40	6	40	6	40	22
20K	16	40	79	40	56	40	25	40	19	40	44	40	25
40K	4	80	3	80	14	80	82	80	6	80	3	80	82
40K	8	80	232	80	164	80	87	80	18	80	24	80	87
40K	16	80	341	80	249	80	98	80	78	80	194	80	98
80K	4	160	10	160	57	160	327	*183*	15	160	9	160	329
80K	8	160	942	160	698	160	345	160	52	160	88	160	345
80K	16	160	1389	160	976	160	391	160	268	160	649	160	391
160K	4	320	30	320	198	320	1335	*913*	39	320	26	320	1311
160K	8	320	3122	320	2618	320	1387	320	191	320	348	320	1380
160K	16	320	5730	320	4089	320	1580	320	958	320	2540	320	1565

Table 2. Results for $\varepsilon = 0.05$ and a smaller coordinate space, where trajectories are densely distributed, i.e. the position-coordinates are in $[0, ..., 2^{13})$. The number of flocks is reported and the running time (in seconds).

Coordinate space $\left[\mathbf{0,...,2^{13}}\right)$ vs. $\left[\mathbf{0,...,2^{16}}\right)$. Somewhat surprising might be that the experiments with point-sets with coordinates in $\left[0,...,2^{16}\right)$ were much faster than those with point-sets with coordinates in $\left[0,...,2^{13}\right)$ (all other parameters were the same). An explanation is that in a bigger underlying space (i.e. where the coordinates are in $\left[0,...,2^{16}\right)$) it is more likely that the query region falls into a single square corresponding to a quadtree node. Due to the sparseness of the point-sets the algorithms are likely to find just a single point in that square. On the other hand in a smaller underlying space the query region might intersect more squares, which results in more subsequent queries, which in turn takes more time.

Uniformly vs. clustered. When comparing the results of the uniformly distributed point-sets with the clustered point-sets it becomes evident that our tree-based algorithms almost always perform better on the clustered data (an exception is the pruning algorithm for a small underlying space; explained below). This behaviour could be expected because, as we have seen from the experiments in general, uniformly distributed points result in trees that are rather flat (especially for higher dimensions). Before exploring what this means, recall that an internal node of an original (compressed) quadtree has 2^d nodes as children, where d is the dimension. In our modified quadtree a node will not have pointers to all these children, but will have a list associated that only contains the non-empty children. Now consider e.g. a quadtree for 16 time-steps. Then the root node R of that tree has 2^{32} children. If we store $n = 160,000$ uniformly distributed points (in 32 dimensions) in that quadtree, we will most likely have that all children nodes of R correspond to quadrants of the tree that are very sparsely filled. A quadrant that contains only one point will not create another level in the tree structure. Hence, for high dimensions it is unlikely that we have trees with large height. Querying such a flat tree, however, is expensive as this involves checking all 2^d children or, as in our case, traversing the entire list of non-empty children, which is likely to have $O(n)$ length in high dimensions. 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 deeper on some branches or subtrees, and therefore the algorithm will descent on those subtrees cutting off everything not contained in them. As expected, 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. We observe that 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 pipe algorithms are strongly influenced by the number of time-steps and the number of points. As discussed above for high dimensions the box and pipe methods operate on an underlying tree that is very flat. 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 pipe method is caused by the different data structure they use. The box method uses the more complex skip-quadtree, while the pipe method incorporates a compressed quadtree.

Pruning. The impressive impact of the pruning method is illustrated in Table 3, which shows the running times of this method for $k = 16$ and $\varepsilon = 0.05$. Depending on the density and distribution, even some point-sets with more than 1 million points can be dealt with by the pruning method within a couple of minutes. Furthermore, we observed that the number of time-steps hardly has an influence on the running times. Also the point distribution (uniformly or clustered) does not affect the running times of the point-sets with coordinates in $\left[0,...,2^{16}\right)$.

However, for the point-sets with coordinates in $\left[0,...,2^{13}\right)$, we observe much longer running times for the clustered point-sets. When looking at the number of queries (also output by the algorithm) to the tree structure that were performed during the second phase of the pruning, we see that e.g. for $n = 1280K$ around 3,000 queries were performed for the uniform data and around 520,000 queries were performed for the clustered data. Hence, in the uniform data sets many more points were pruned, while in the clustered data sets many random flocks were found for the first two time-steps and therefore their points were not pruned. For the clustered data, this results in many more expensive queries in high dimensional space that need to be carried out.

input		coordinates from $\left[0,...,2^{13}\right)$				coordinates from $\left[0,...,2^{16}\right)$			
		uniformly		clustered		uniformly		clustered	
n	k	flocks	time	flocks	time	flocks	time	flocks	time
10K	16	20	0	20	2	20	1	20	0
20K	16	40	2	40	3	40	1	40	1
40K	16	80	4	80	7	80	1	80	1
80K	16	160	11	160	16	160	4	160	4
160K	16	320	31	320	47	320	9	320	9
320K	16	640	80	640	328	640	25	640	25
640K	16	1280	196	1280	2271	1280	74	1280	74
1280K	16	2560	525	2560	12456	2560	242	2560	242

Table 3. Results for the pruning method for $\varepsilon = 0.05$. The number of flocks is reported and the running time (in seconds).

6 Concluding remarks

In this paper we have presented different algorithms for finding flock patterns and analysed them theoretically as well as experimentally. From the experiments we have seen that our tree-based algorithms can perform very well. Especially for a small number of time-steps the resulting running times are often very small, however, they depend very much on the characteristics of the input point-sets, which motivates more research and experiments, preferably on real-world data.

For a larger number of time-steps the no-tree method can be used. This method's running time is mainly influenced by the number of entities and not by the number of dimensions. Table 4 shows the performance of this algorithm for up to $40K$ entities and up to 1000 time-steps. As we have seen from Tables 1 and 2, the characteristics (such as distribution and coordinate space) of the point sets has no influence on the running time of the no-tree method and therefore, Table 4 only shows the results for uniformly distributed points with coordinates in $\left[0,...,2^{16}\right)$. We see that also point sets with 1000 time-steps can be searched for flocks of length 1000 within a couple of minutes.

input	uniformly distributed, coordinates from $\left[0,...,2^{16}\right)$											
	$k = 32$		$k = 64$		$k = 125$		$k = 250$		$k = 500$		$k = 1000$	
n	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	20	14	20	15	20	14	20	14	20	15	20	17
20K	40	57	40	64	40	67	40	71	40	79	40	90
40K	80	229	80	257	80	263	80	276	80	306	80	331
80K	160	914	160	1026	160	1052	160	1099	160	1211	160	1285

Table 4. Results of the no-tree method, $\varepsilon = 0.05$. The number of flocks is reported and the running time (in seconds).

Hence, for a small number n of entities and many time-steps, we can use the no-tree method, which has a running time quadratic in n. For many entities and few time-steps k our tree based methods perform very well, which have a running time exponential in the number of dimensions of the tree, i.e. exponential in k. Thus, we are faced with a trade-off. One approach to tackle the case of many entities and many time-steps has recently been developed by Al-Naymat et al. [1], where the data is preprocessed. In this preprocessing step the number of dimensions (i.e. time-steps) is reduced by using random projection. However, to be able to use this technique the definition of a flock has to be relaxed. Instead of using a maximum radius of r in each time step they bound the sum of the distances. Intuitively this means that two entities that are very close to each other in all but one time step may still belong to the same flock. In experiments it was shown [1] that the tree-based methods perform very well on the data with reduced dimensionality. As a conclusion we see that the idea of projecting trajectories into points in higher dimensional space is very viable for finding flocks in spatio-temporal data.

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.

Acknowledgements

We would like to thank the anonymous reviewers of an earlier version of this article for their very useful comments and suggestions, and also thanks to Damian Merrick, Bojan Djordjevic and Adel Ahmed for their help regarding the implementations.

References

1. G. Al-Naymat, S. Chawla, and J. Gudmundsson. Dimensionality reduction for long duration and complex spatio-temporal queries. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, pages 393–397. ACM, 2007.
2. S. Arya and D. M. Mount. Approximate range searching. *Computational Geometry—Theory and Applications*, 17(3–4):135–152, 2000.
3. G Balme and L. Hunter. Mortality in a protected leopard population, phinda private game reserve, south africa: A population in decline? *Ecological Journal*, 6, 2004.
4. M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computaional Geometry and Applications*, 9(6):517–532, 1999.
5. H. Dettki, G. Ericsson, and L. Edenius. Real-time moose tracking: an internet based mapping application using GPS/GSM-collars in Sweden. *Alces*, 40:13–21, 2004.
6. D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proceedings of the 21st ACM Symposium on Computational Geometry*, pages 296–305, 2005.
7. A. U. Frank, J. F. Raper, and J.-P. Cheylan, editors. *Life and motion of spatial socio-economic units.* Taylor & Francis, London, 2001.
8. J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of the 14th ACM Symposium on Advances in GIS*, pages 35–42, 2006.
9. J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal sets. *GeoInformatica*, 11(2):195–215, 2007.
10. M. Heurich, P. Löttker, A. Stache, F. Baierl, and H. Kiener. Der Luchs im Bergwaldökosystem. *AFZ - Der Wald*, 10:530–531, 2007.
11. S. Iwase and H. Saito. Tracking soccer player using multiple views. In *Proceedings of the IAPR Workshop on Machine Vision Applications (MVA02)*, pages 102–105, 2002.
12. 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.
13. P. Laube and S. Imfeld. Analyzing relative motion within groups of trackable moving point objects. In M. J. Egenhofer and D. M. Mark, editors, *Geographic Information Science 2002*, volume 2478 of *Lecture Notes in Computer Science*, pages 132–144, Berlin, 2002. Springer.
14. P. Laube, M. van Kreveld, and S. Imfeld. Finding REMO – detecting relative motion patterns in geospatial lifelines. In P. F. Fisher, editor, *Developments in Spatial Data Handling: Proceedings of the 11th International Symposium on Spatial Data Handling*, pages 201–214, Berlin, 2004. Springer.

15. D. P. Metha and S. Sahni, editors. *Handbook Of Data Structures And Applications (Chapman & Hall/CRC Computer and Information Science Series.).* Chapman & Hall/CRC, 2005.
16. H. J. Miller and J. Han, editors. *Geographic Data Mining and Knowledge Discovery.* Taylor & Francis, London, 2001.
17. J. F. Roddick, K. Hornsby, and M. Spiliopoulou. An Updated Bibliography of Temporal, Spatial, and Spatio-temporal Data Mining Research. In J. F. Roddick and K. Hornsby, editors, *Temporal, spatial and spatio-temporal data mining, TSDM 2000*, volume 2007 of *Lecture Notes in Artificial Intelligence*, pages 147–163, Berlin, 2001. Springer.
18. C.-B. Shim and J.-W.Chang. A new similar trajectory retrieval scheme using k-warping distance algorithm for moving objects. In *Proceedings of the 4th International Conference on Advances in Web-Age Information Management, (WAIM 2003)*, number 2762 in Lecture Notes in Computer Science, pages 433–444. Springer, Berlin, 2003.
19. N. Sumpter and A. J. Bulpitt. Learning spatio-temporal patterns for predicting object behaviour. *Image Vision and Computing*, 18(9):697–704, 2000.
20. F. Verhein and S. Chawla. Mining spatio-temporal association rules, sources, sinks, stationary regions and thoroughfares in object mobility databases. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA)*, volume 3882 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2006.