

# Reporting Flock Patterns

Marc Benkert<sup>1\*</sup>   Joachim Gudmundsson<sup>2</sup>   Florian Huebner<sup>1</sup>   Thomas Wolle<sup>2</sup>

<sup>1</sup> Department of Computer Science, Karlsruhe University, Germany. {mbenkert,florianh}@ira.uka.de

<sup>2</sup> National ICT Australia,† Sydney, 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 analyzed 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 [14, 16] is to analyze such data sets for interesting patterns. For example, a group of 25 moose in Sweden was equipped with GPS-GSM collars. The GPS collar acquire a position every half hour and then sends the information to a GSM-modem where the positions are extracted and stored. Analyzing this data gives insight into entity behavior, in particular, migration patterns. There are many other examples where spatio-temporal data is collected [1, 15]. The analysis of moving objects also has applications in sports (e.g., soccer players [10]), in socio-economic geography [7] and in defence and surveillance areas.

The input is a set  $P$  of  $n$  moving point objects  $p_1, \dots, p_n$  whose locations are known at  $\tau$  consecutive time steps  $t_1, \dots, t_\tau$  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 velocity of an entity along a line segment of the trajectory is constant.

There is some research on data mining of moving objects (e.g., [11, 17, 18, 21]) in particular, on the discovery of similar directions or clusters. Verhein and Chawla [21] used associated data mining to detect patterns in spatio-temporal sets. They partitioned spaced into cells and then defined a cell to be a *source*, *sink* or a *thoroughfare* depending on the number of objects entering, exiting or passing through the cell.

Laube and Imfeld [12] proposed a different approach in 2002 - the REMO framework (Relative MOTion) which defines similar behavior 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. [13] 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 [13] 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 [4] to the problem gives a  $(1 + \varepsilon)$ -approximation with expected running

---

\*Supported by grant WO 758/4-2 of the German Science Foundation (DFG).

†NICTA is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

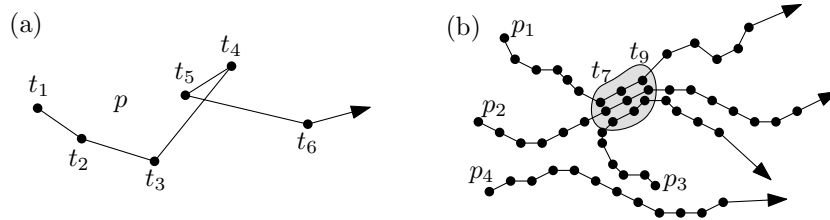


Figure 1: (a) A polygonal line describing the movement of an entity  $p$  in the time interval  $[t_1, t_6]$ . (b) A flock for  $p_1, p_2, p_3$  in the time interval  $[t_7, t_9]$ .

time  $\mathcal{O}(\tau n / \varepsilon^2 \log^2 n)$ . Gudmundsson et al. [9] 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 it is defined as a flock. Therefore we propose the following definition of a flock:

**Definition 1** ( $(m, k, r)$ -flock<sub>A</sub>) - Given a set of  $n$  trajectories where each trajectory consists of  $\tau$  line segments, a flock in a time interval  $I = [t_i, t_j]$ , where  $j - i + 1 \geq k$ , 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.

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 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  $\Delta r$  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  $\tau n$ ) 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.

This paper is organized 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. Then, in Section 3, we give three approximation algorithms all derived from a 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 experimental results.

## 1.1 The skip quadtree and 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. A small modification to the skip-quadtree results in the following lemma (a more detailed discussion can be found in the appendix):

**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 [5, 6] in computational geometry using quadtrees or octrees is that certain operations can be done in constant time. In arithmetic terms, the computations needed to perform point location, range queries or nearest neighbor 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 normalization.

## 2 Approximate flocks

The input is a set  $P$  of  $n$  trajectories  $p_1, \dots, p_n$ , where each trajectory  $p_i$  is a sequence of  $\tau$  coordinates in the plane  $(x_1^i, y_1^i), (x_2^i, y_2^i), \dots, (x_\tau^i, y_\tau^i)$ , 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<sub>B</sub>) - Given a set of  $n$  trajectories where each trajectory consists of  $\tau$  line segments a flock in a time interval  $[t_i, t_j]$ , where  $j - i + 1 \geq k$  consists of at least  $m$  entities such that for every discrete time step  $t_\ell$ ,  $i \leq \ell \leq j$ , there is a disk of radius  $r$  that contains all the  $m$  entities.

Note that the center 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.

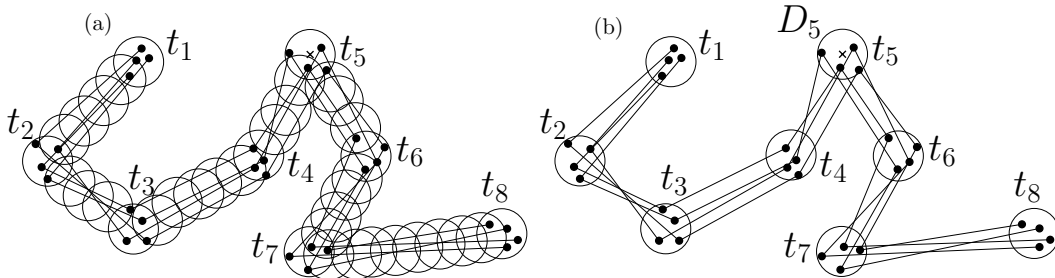


Figure 2: A flock of four entities in the time interval  $[t_1, t_8]$ , according to the definitions of (a) flock<sub>A</sub> and (b) flock<sub>B</sub>.

**Lemma 2** If the entities move with constant velocity along the straight line segment between two consecutive time steps then flock<sub>A</sub> and flock<sub>B</sub> 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  $D_\ell$  and  $D_{\ell+1}$  be disks of radius  $r$  that include the entities of  $f_B$  at time  $t_\ell$  and  $t_{\ell+1}$  respectively, see Fig. 3a. It is enough to consider every two discrete time steps  $t_\ell$  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  $\{p_1, \dots, p_m\}$  in  $f_B$ . Let  $c_\ell$  and  $c_{\ell+1}$  be the centers of  $D_\ell$  and  $D_{\ell+1}$  respectively, and let  $h$  be the straight-line segment with endpoints at  $c_\ell$  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  $c_\gamma$ . Next we show that the disk  $\mathcal{D}_\gamma$  with center at  $c_\gamma$  and radius  $r$  contains all the entities of  $f_B$  at time  $\gamma$ . Let  $p_i$  be an arbitrary entity of  $f_B$ . The movement of  $q$  and the movement of  $p_i$  during  $I'$  follows a straight-line and both move with constant velocity thus 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  $p_i$  lies within  $D_\ell$  and  $D_{\ell+1}$  at time  $t_\ell$  and  $t_{\ell+1}$ , 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.  $\square$

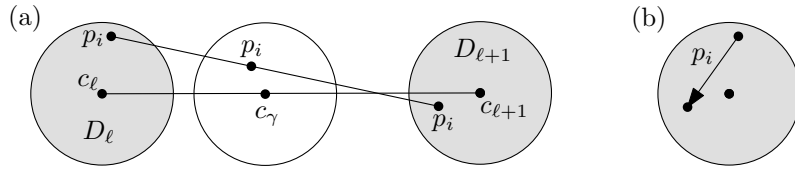


Figure 3: Illustration of the proof of Lemma 2.

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  $F = \{p_1, \dots, p_m\}$  that can be covered by a disk of radius  $r$  at each discrete time step in  $[t_i, t_{i+k-1}]$ . Next 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 bounds 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 modeled as a point in  $2d$  dimensions. The trajectory of an entity  $p$  in the time interval  $[t_i, t_j]$  is described by the polygonal line  $p(i, j) = \langle (x_i, y_i), (x_{i+1}, y_{i+1}), \dots, (x_j, y_j) \rangle$ , which corresponds to a point  $p'(i, j) = (x_i, y_i, x_{i+1}, y_{i+1}, \dots, x_j, y_j)$  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 characterization 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:

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

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

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

To see that this equivalence holds we observe the following: for each time step  $t_i \in I$  the disk with radius  $r$  and center  $(x_i, y_i)$  contains the entity positions  $p_1^i, \dots, p_m^i$ .

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  $\Delta$ -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  $\Delta r$  will be reported.

#### 3.1 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  $1 \leq i \leq \tau - k + 1$ , 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  $P' = \{p'_1, \dots, p'_n\}$ . 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 of side length  $4r$  and center at  $p'$ . That is, we approximate the  $2k$ -dimensional cube which is itself an approximation for the query region. Every counting query 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.

**Lemma 3** *Method 1 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  $D_{t_i}$  with radius  $r$  that contains the entities in  $f$  for each discrete time step  $t_i$  in  $I$ . The algorithm performs a counting query for each entity corresponding point in  $\mathbb{R}^{2k}$  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 center at  $p'_f$ , where  $p'_f$  is the point in  $2k$ -dimensions corresponding to  $p_f$ . For a discrete time  $t_\ell$ , the query range corresponds to a square in two dimensions with center at  $p_f$  and side length  $4r$ . 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  $f \subseteq g$ .

For 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  $I = [t_i, t_{i+k-1}]$ . For every time step  $t_\ell$  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  $\delta$ . If we choose  $\delta$  to be  $\varepsilon/\sqrt{8}$ , the square of side length  $4(1 + \delta)r$  is contained in the disk with radius  $(\sqrt{8} + \varepsilon)r$  centered at  $p'_f$ , as illustrated in Fig. 4a. This completes the proof of the lemma.  $\square$

**Lemma 4** *Method 1 runs in  $\mathcal{O}(\tau nk^2(\log n + 1/\varepsilon^{2k-1}))$  time and requires  $\mathcal{O}(\tau n)$  space.*

**Proof.** 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  $(1 + \delta)$  counting query is performed for each of the  $n$  entities; each query requires  $\mathcal{O}(k^2(\log n + 1/\varepsilon^{2k-1}))$  time as  $\delta$  is in  $\mathcal{O}(\varepsilon)$ . Hence, the total time needed to perform all the  $n(\tau - k')$  queries is bounded by  $\mathcal{O}(\tau k^2 n(\log n + 1/\varepsilon^{2k-1}))$  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.  $\square$

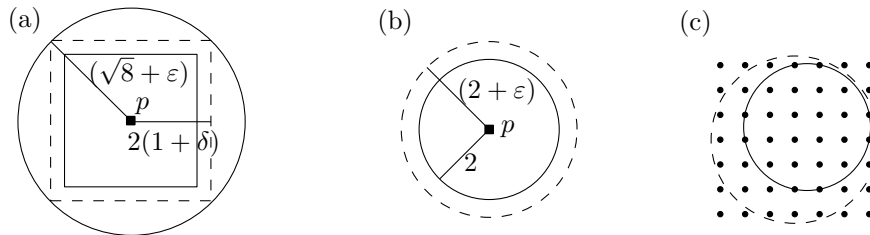


Figure 4: Illustration of the approximative range of methods 1,2 and 3 for  $r = 1$ . The dashed region is the query region.

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

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  $1 \leq i \leq \tau - k + 1$ , 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  $P' = \{p'_1, \dots, p'_n\}$ . Then, for each point  $p' \in P'$  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  $t_i$ .

The definition of fatness we use was introduced by Van der Stappen [19]. For convex objects it is basically equivalent to other definitions [2, 3, 20].

**Definition 3** [19] Let  $\alpha > 1$  be a real value. An object  $s$  is  $\alpha$ -fat if for any  $d$ -dimensional ball  $D$  whose center lies in  $s$  and whose boundary intersects  $s$ , we have  $\text{volume}(D) \leq \alpha \cdot \text{volume}(s \cap D)$ .

**Lemma 5** The intersection of  $d$  pipes  $(x_i, y_i, 2i - 1, 2r)$ ,  $1 \leq i \leq k$ , in  $2d$ -dimensional space is a bounded convex  $4^d$ -fat region whose boundary consists of  $\mathcal{O}(d)$  surfaces of quadratic complexity.

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

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

**Lemma 7** Method 2 runs in  $\mathcal{O}(\tau nk^2(\log n + 1/\varepsilon^{2k-1}))$  time and requires  $\mathcal{O}(\tau n)$  space.

**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, in practice we chose to implement the second method using a compressed quadtree. The reason for this is that the skip-quadtree computes the volume 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 1 and 2 use a different query data structure.

### 3.3 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  $1 \leq i \leq \tau - k + 1$ , 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  $P' = \{p'_1, \dots, p'_n\}$ . Let  $\Gamma$  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  $2r$  centered at  $p'_i$ . Clearly, this gives rise to  $\mathcal{O}(1/\varepsilon^{2k})$  sample points for each entity  $p$ .

Next, we perform a  $(1 + \frac{\varepsilon}{2+\varepsilon})$ -approximate range counting query in  $T$  for each sample point  $(x_1, y_1, \dots, x_k, y_k)$  where the query range is the intersection of the  $k$  pipes  $(x_i, y_i, 2i-1, (1+\varepsilon/2)r)$ ,  $1 \leq i \leq k$ . 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  $2r$  centered 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** *Method 3 is a  $(1 + \varepsilon)$ -approximation algorithm.*

**Lemma 9** *Method 3 runs in  $\mathcal{O}(\frac{\tau nk^2}{m\varepsilon^{2k}}(\log n + 1/\varepsilon^{2k-1}))$  time and requires  $\mathcal{O}(\tau n)$  space.*

## 4 Prune the set of flocks reported

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. 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. This gets even worse if a flock is found whose number of entities exceeds  $m$ . Below we briefly discuss two approaches how reporting this redundant information could be avoided.

**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 a bounded number of flocks.** The above approach minimizes 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 query point within the query region will be marked, so that no query will be performed with those points as centers. 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.

## 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 algorithms and we present and discuss the running times of them with respect to different parameters of the input.

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 running times are reported as seconds. Our point sets used in the experiments were created artificially. Each point coordinate of an input point is taken from the interval  $[0, \dots, 2^{13}]$  or  $[0, \dots, 2^{16}]$ . 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). 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. In all point sets, 10% of the points were placed in such a way that they form (randomly positioned) flocks of 50 entities in a circle of radius 50 (hence the number of artificially inserted flocks is 0.002 times the number of points). Note that each generated data instance contains the coordinates of points for a certain number of time steps  $\tau$ , and in the experiments on that instance, we always looked for flocks of at least 50 entities in a circle with radius 50 and of length  $k$  with  $k = \tau$ .

### 5.1 Methods

We compare the results of four methods called ‘box’, ‘pipe’, ‘no-tree’ and ‘pruning’. The box and pipe method are named after their query region and 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 center and the inner one (running again over all input points) computing the distance between a point and the potential flock center. If there are enough points within a ball (around the potential flock center) 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  $k$ . It works as follows: As a first step we compute flocks of length 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 of the input.

### 5.2 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  $[0, \dots, 2^{16}]$ . The columns below ‘input’ specify the number of points and the number of timesteps, and the columns below ‘uniformly’ and ‘clustered’ show the number of flocks found and the running times needed when performing the box-, pipes- and notree-algorithm on the corresponding input. We also performed the same experiments on point-sets where the coordinates were chosen from  $[0, \dots, 2^{13}]$ . 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. All tables show the results only for  $\varepsilon = 0.05$ , because no big influence of different values of  $\varepsilon$  could be observed. The number of flocks found

| input |     | uniformly |      |            |      |        |      | clustered |      |        |      |        |      |
|-------|-----|-----------|------|------------|------|--------|------|-----------|------|--------|------|--------|------|
| size  | $k$ | box       |      | pipes      |      | notree |      | box       |      | pipes  |      | notree |      |
|       |     | 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    | <i>41</i>  | 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    | <i>163</i> | 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    | <i>321</i> | 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  $[0, \dots, 2^{16}]$ .

by the algorithms is indicated in *italics* in case it deviates from the number of artificially inserted flocks.

### 5.3 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 dimensions). Recall that an internal node of an octree has  $2^d$  children where  $d$  is the number of dimensions. Using 16 timesteps means 32 dimensions which translates to more than 4,000 million 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, which results in high running times.

**Error value  $\varepsilon$ .** When performing a range query,  $\varepsilon$  influences the approximate region to be queried. One could expect that a larger value of  $\varepsilon$  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 behavior could not be observed in our experiments. Our point sets and therefore our trees in the experiments are rather sparse. 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  $\varepsilon$  and only used  $\varepsilon = 0.05$ .

**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 timesteps, 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 timesteps. In one case more than 1300 flocks were found (where only 320 were artificially inserted) which indicates that for that instance the distribution of the points and clusters (in combination with a high number of points and a small coordinate space) reached a

| input |     | uniformly |      |        |      |        |      | clustered |      |        |      |        |      |
|-------|-----|-----------|------|--------|------|--------|------|-----------|------|--------|------|--------|------|
| size  | $k$ | box       |      | pipes  |      | notree |      | box       |      | pipes  |      | notree |      |
|       |     | 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  $[0, \dots, 2^{13}]$ .

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 artificially 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  $[0, \dots, 2^{13}]$  vs.  $[0, \dots, 2^{16}]$ .** Somewhat surprisingly, the experiments with point-sets with coordinates in  $[0, \dots, 2^{16}]$  were much faster than those with point-sets with coordinates in  $[0, \dots, 2^{13}]$  (all other parameters were the same). One explanation is that in a bigger underlying space (i.e. where the coordinates are in  $[0, \dots, 2^{16}]$ ) 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 always perform better on the clustered data. This behavior could be expected because, as we have seen from the experiments in general, uniformly distributed points result in octrees 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 depth 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. 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 timesteps, as expected. On the other hand the box and pipe algorithms are strongly influenced by the number of timesteps 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 behavior to become similar (although with a big overhead) to the no-tree method.

| input |     | coordinates from $[0, \dots, 2^{13}]$ |      |           |      | coordinates from $[0, \dots, 2^{16}]$ |      |           |      |
|-------|-----|---------------------------------------|------|-----------|------|---------------------------------------|------|-----------|------|
|       |     | uniformly                             |      | clustered |      | uniformly                             |      | clustered |      |
| size  | $k$ | pruning                               |      | pruning   |      | pruning                               |      | pruning   |      |
|       |     | 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,  $\varepsilon = 0.05$ .

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-quad-tree, while the pipe method incorporates a compressed quadtree.

**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 delt with in a couple of minutes. Furthermore, we observed that the number of time steps only has an influence on the running times of the clustered point sets with coordinates in  $[0, \dots, 2^{13}]$  (because of space restrictions, we only give the numbers for 16 time steps). Also the point distribution (uniformly or clustered) has the same effect. 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. 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  $[0, \dots, 2^{13}]$ , however, the probability of random flocks is much higher. 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.

- one more observation: the box method is often faster than the pipe method for the b instances!

## 6 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.

## References

- [1] Wildlife tracking projects with GPS GSM collars, 2006.  
[www.environmental-studies.de/projects/projects.html](http://www.environmental-studies.de/projects/projects.html).
- [2] P. Agarwal, M. Katz, and M. Sharir. Computing depth orders for fat objects and related problems. *Computational Geometry – Theory & Applications*, 5:187–206, 1995.
- [3] H. Alt, R. Fleischer, M. Kaufmann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Approximate motion planning and the complexity of the boundary of the union of simple geometric figures. *Algorithmica*, 8(5 & 6):391–406, 1992.

- [4] B. Aronov and S. Har-Peled. On approximating the depth and related problems. In *Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 886–894, 2005.
- [5] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. *Int. Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.
- [6] 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*, 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 spatio-temporal data. Manuscript, April 2006.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] P. Laube and S. Imfeld. Analyzing relative motion within groups of trackable moving point objects. In *GIScience 2002*, number 2478 in Lecture Notes in Computer Science, pages 132–144. Springer, Berlin, 2002.
- [13] 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*, pages 201–214, 2004.
- [14] H.J. Miller and J. Han, editors. *Geographic Data Mining and Knowledge Discovery*. Taylor & Francis, London, 2001.
- [15] Porcupine caribou herd satellite collar project. <http://www.taiga.net/satellite/>.
- [16] J. Roddick, K. Hornsby, and M. Spiliopoulou. An updated bibliography of temporal, spatial, and spatio-temporal data mining research. In *TSDM 2000*, number 2007 in Lecture Notes in Artificial Intelligence, pages 147–163. Springer, Berlin, 2001.
- [17] 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.
- [18] N. Sumpter and A. J. Bulpitt. Learning spatio-temporal patterns for predicting object behaviour. *Image Vision and Computing*, 18(9):697–704, 2000.
- [19] A. F. van der Stappen. *Motion Planning amidst Fat Obstacles*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1994.
- [20] M. van Kreveld. On fat partitioning, fat covering, and the union size of polygons. In *Proc. 3rd Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 452–463. Springer-Verlag, 1993.
- [21] F. Verhein and S. Chawla. Mining spatio-temporal association rules, sources, sinks, stationary regions and thoroughfares in object mobility databases. In *Proceedings of the Workshop on Temporal Data Mining: Algorithms, Theory and Applications*, 2005.

## A Appendix

### A.1 Skip Quadtree

The skip quadtree uses the compressed quadtree the bottom-level structure. The standard compressed quadtree 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 with an  $\mathcal{O}(\log d)$ -factor.

The description is taken from [6]. Consider the standard quadtree  $T$  of the input set  $S$ . We may assume that the center 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  $p$  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  $2^d$  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 definition of a compressed quadtree implies that the size of the tree is  $\mathcal{O}(2^d \cdot n)$ . Instead of storing information about which children that exist and which children that are empty, we modify the tree such that it contains a list containing 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  $Q_\delta$ . The extended query range  $Q_\delta$  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  $\mathbb{R}^d \setminus Q_\delta$ .

Together with the results of [6], the above discussion can be summarized by 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.

### A.2 Proofs from Section 3

**Lemma 6** Method 2 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  $(x_i, y_i, 2i - 1, 2r)$ ,  $1 \leq i \leq k$  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 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  $2r$  centered at  $(x_i, y_i)$ . Thus, when querying  $p$ , the algorithm reports an  $(m, k, (2 + \varepsilon)r)$ -flock  $g$  with  $f \subseteq g$ .  $\square$

**Lemma 7** Method 2 runs in  $\mathcal{O}(\tau n k^2 (\log n + 1/\varepsilon^{2k-1}))$  time and requires  $\mathcal{O}(\tau n)$  space.

**Proof.** At each of the  $(\tau - k')$  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 point in  $\mathcal{P}'$ ; each query requires  $\mathcal{O}(k^2 (\log n + 1/\varepsilon^{2k-1}))$  time, thus the

total time needed to perform all the  $n$  queries is bounded by  $\mathcal{O}(k^2n(\log n + 1/\varepsilon^{2k-1}))$  time 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.  $\square$

**Lemma 5** The intersection of  $d$  pipes  $(x_i, y_i, 2i - 1, 2r)$ ,  $1 \leq i \leq k$ , in  $2d$ -dimensional space is a bounded convex  $4^d$ -fat region whose boundary consists of  $\mathcal{O}(d)$  surfaces of quadratic complexity.

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

$$\begin{aligned} x_1^2 + x_2^2 &\leq r^2 \\ x_3^2 + x_4^2 &\leq r^2 \\ &\dots \\ x_{d-1}^2 + x_d^2 &\leq r^2. \end{aligned}$$

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. Thus it remains to prove that  $\mathcal{I}$  is  $4^d$ -fat. The definition of  $\alpha$ -fat says that a  $d$ -dimensional region  $R$  is  $\alpha$ -fat if for every  $d$ -dimensional ball  $B$  with center within  $R$  and whose boundary intersects  $R$  the volume of  $R$  within  $B$  is at least  $1/\alpha$  of the total volume of  $B$ . We place an arbitrary ball whose center lies in  $\mathcal{I}$  and consider the intersection of the ball and  $\mathcal{I}$  projected onto two dimensions, say dimension  $2i - 1$  and  $2i$ . The projection of the ball is a disk whose center lies within the projection of  $\mathcal{I}$  which is also a disk. Thus, it is obvious that at least  $1/4$  of the projected ball lies into the projected region of  $\mathcal{I}$ . Taking all dimensions into account this yields that  $\mathcal{I}$  is  $4^d$ -fat.  $\square$

**Lemma 8** Method 3 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  $(x_1, y_1, \dots, x_k, y_k) \in \mathbb{R}^{2k}$  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  $(\dots, x_i^q, y_i^q, \dots) \in \Gamma$  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  $(1 + \varepsilon/2)r$  centered at  $q$  completely contains the disk with radius  $r$  centered at  $(x_i, y_i)$ . Thus, when checking the sample points  $(\dots, x_i^q, y_i^q, \dots)$  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  $f \subseteq g$ .  $\square$

**Lemma 9** Method 3 runs in  $\mathcal{O}(\frac{\tau nk^2}{m\varepsilon^{2k}}(\log n + 1/\varepsilon^{2k-1}))$  time and requires  $\mathcal{O}(\tau n)$  space.

**Proof.** At each of the  $(\tau - k')$  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 of the  $\mathcal{O}(n/(m\varepsilon^{2k}))$  candidate sample points in  $\Gamma$ ; each query requires  $\mathcal{O}(k^2(\log n + 1/\varepsilon^{2k-1}))$  time, thus the total time needed to perform all  $n(\tau - k')$  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.  $\square$