On Discovering Moving Clusters in Spatio-temporal Data

Panos Kalnis[1], Nikos Mamoulis[2], and Spiridon Bakiras[3]

[1] Department of Computer Science, National University of Singapore
kalnis@comp.nus.edu.sg
[2] Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong
nikos@cs.hku.hk
[3] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
sbakiras@cs.ust.hk

Abstract. A moving cluster is defined by a set of objects that move close to each other for a long time interval. Real-life examples are a group of migrating animals, a convoy of cars moving in a city, etc. We study the discovery of moving clusters in a database of object trajectories. The difference of this problem compared to clustering trajectories and mining movement patterns is that the identity of a moving cluster remains unchanged while its location and content may change over time. For example, while a group of animals are migrating, some animals may leave the group or new animals may enter it. We provide a formal definition for moving clusters and describe three algorithms for their automatic discovery: (i) a straight-forward method based on the definition, (ii) a more efficient method which avoids redundant checks and (iii) an approximate algorithm which trades accuracy for speed by borrowing ideas from the MPEG-2 video encoding. The experimental results demonstrate the efficiency of our techniques and their applicability to large spatio-temporal datasets.

1 Introduction

With the advances of telecommunication technologies we are able to record the movements of objects over a long history. Data analysts are often interested in the automatic discovery of trends or patterns from large amounts of recorded movements. An interesting problem is to find dense clusters of objects which move similarly for a long period. For example, migrating animals usually move in groups (clusters). Another example could be a convoy of cars that follow the same route in a city.

We observe that in many cases such moving clusters do not retain the same set of objects in their lifetime, but objects may enter or leave, while the clusters are moving. In the migrating animals example, during the movement of the cluster, some new animals may enter the group (e.g., those passing nearby the cluster's trajectory), while some animals may leave the group (e.g., those attacked and eaten by lions). Nevertheless the cluster itself retains its density during its whole lifetime, no matter whether it ends up with a totally different set of objects compared to its initial formation.

The automatic discovery of such *moving* clusters is arguably an important problem with several applications. For example, ecologists may want to study the evolution of

C. Bauzer Medeiros et al. (Eds.): SSTD 2005, LNCS 3633, pp. 364–381, 2005.

moving groups of animals. Military applications may monitor troops that move in parallel and merge/evolve over time. The identification of moving dense areas of traffic is useful to traffic surveillance systems. Finally, intelligence and counterterrorism services may want to identify suspicious activity of individuals moving similarly.

The contribution of this paper is the formal definition of moving clusters and the proposal of methods that automatically discover them from a long history of recorded trajectories. Intuitively, a moving cluster is a sequence of spatial clusters that appear in consecutive snapshots of the object movements, such that two consecutive spatial clusters share a large number of common objects. Here, we propose three methods to identify moving clusters in spatio-temporal datasets. Based on the problem definition, our first algorithm, MC1, performs spatial clustering at each snapshot and combines the results into a set of moving clusters. We prove that we can speed-up this process by pruning object combinations that cannot belong to the same cluster, without affecting the correctness of the solution; the resulting algorithm is called MC2. Then, we observe that many clusters remain relatively stable in consecutive snapshots. The challenge is to identify them without having to perform clustering for the entire set of objects. We propose an approximate algorithm, called MC3, which uses information from the past to predict the set of clusters at the current snapshot. In order to minimize the approximation error, we borrow from MPEG-2 video encoding the idea of interleaving approximate with exact cluster sets. We minimize the number of the expensive computations of exact cluster sets by employing a method inspired by the TCP/IP protocol. Our experiments show that MC3 reduces considerably the execution time and produces high quality results.

Previous work has focused mainly on the identification of static dense areas over time [1], or on the clustering of object trajectories for sets that contain the same objects during a time interval [2]. Our problem is different, since both the location and the set of objects of a moving cluster change over time. Related to our work is the incremental maintenance of clusters [3]. Such methods are efficient only if a small percentage of objects is updated. This is not true in our case since potentially all object may be updated (i.e., move) in consecutive snapshots. Recent methods which use approximation to improve the efficiency of incremental clustering [4] are also not applicable in our problem since they do not maintain the continuity of clusters in the time dimension. To the best of our knowledge, this is the first work which deals with the identification of moving clusters.

The rest of the paper is organized as follows: First, we present a formal definition of our problem in Section 2, followed by a survey of the related work in Section 3. Our methods are presented in Section 4, while Section 5 contains the results of our experiments. Finally, Section 6 summarizes the paper and presents some directions for future work.

2 Problem Formulation

Let $H = \{t_1, t_2, \ldots, t_n\}$ be a long, timestamped history. Let $\mathcal{S} = \{o_1, o_2, \ldots, o_m\}$ be a collection of objects that have moved during H. An object o_i not necessarily existed throughout the whole history, but during a contiguous subsequence $o_i.T$ of H. Without

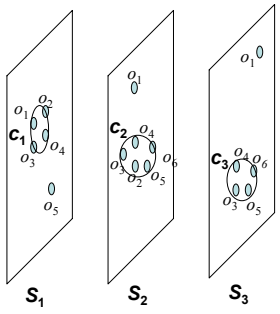


Fig. 1. Example of a moving cluster

loss of generality, we assume that the locations of each object were sampled at every timestamp during $o_i.T$. We refer to $o_i.T$, as the *lifetime* of o_i.

A *snapshot* S_i of H is the set of objects and their locations at time t_i. S_i is a subset of $\mathcal{S}$, since not all objects in $\mathcal{S}$ necessarily existed at t_i. Formally, $S_i = \{o_j \in \mathcal{S} : t_i \in o_j.T\}$. Given a snapshot S_i, we can employ a standard spatial clustering algorithm, like DBSCAN [5] to identify dense groups of objects in S_i which are close to each other and the density of the group meets the density constraints ($MinPts$ and ϵ) of the clustering algorithm.

Let c_i and c_{i+1} be two such *snapshot clusters* for S_i and S_{i+1}, respectively. We say that $c_i c_{i+1}$ is a *moving* cluster if $\frac{|c_i \cap c_{i+1}|}{|c_i \cup c_{i+1}|} \geq \theta$, where θ $(0 < \theta \leq 1)$ is an integrity threshold for the contents of the two clusters. Intuitively, if two spatial clusters at two consecutive snapshots have a large percentage of common objects then we consider them as a *moving* cluster that moved between these two timestamps. The definition of a spatio-temporal cluster can be generalized as follows:

Definition 1. *Let $g = c_1, c_2, \ldots, c_k$ be a sequence of snapshot clusters such that for each $i (1 \leq i < k)$, the timestamp of c_i is exactly before the timestamp of c_{i+1}. Then g is a* moving *cluster, with respect to an integrity threshold θ $(0 < \theta \leq 1)$, if $\frac{|c_i \cap c_{i+1}|}{|c_i \cup c_{i+1}|} \geq \theta$, $\forall i : 1 \leq i < k$.*

Figure 1 shows an example of a moving cluster. S_1, S_2, and S_2 are three snapshots. In each of them there is a timeslice cluster (c_1, c_2, and c_3). Let $\theta = 0.5$. $c_1 c_2 c_3$ is a moving cluster, since $\frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} = \frac{3}{6}$ and $\frac{|c_2 \cap c_3|}{|c_2 \cup c_3|} = \frac{4}{5}$ are both at least θ. Note that objects may enter or leave the moving cluster during its lifetime.

3 Related Work

3.1 Clustering Static Spatial Data

Clustering static spatial data (i.e., static points) is a well-studied subject. Different clustering paradigms have been proposed with different definitions and evaluation criteria,

based on the clustering objective. *Partitioning* methods, like k-medoids [6,7], divide the objects into k groups and iteratively exchange objects between them until the quality of the clusters does not further improve. First, k medoids are chosen randomly from the dataset. Each object is assigned to the cluster corresponding to their nearest medoid and the quality of the clusters is defined by summing the distances of all points to their nearest medoid. Then, a medoid is replaced by a random object and the change is committed only if it results to clusters of better quality. A local optimum is reached after a large sequence of unsuccessful replacements. This process is repeated for a number of initial random medoid-sets and the clusters are finalized according to the best local optimum found.

Another class of (agglomerative) *hierarchical* clustering techniques define the clusters in a bottom-up fashion, by first assuming that all objects are individual clusters and gradually merging the closest pair of clusters until a desired number of clusters remain. Algorithms like BIRCH [8] and CURE [9] were proposed to improve the scalability of agglomerative clustering and the quality of the discovered partitions. C2P [10] is another hierarchical algorithm similar to CURE, which employs closest pairs algorithms and uses a spatial index to improve scalability.

Density-based methods discover dense regions in space, where objects are close to each other and separate them from regions of low density. DBSCAN [5] is the most representative method in this class. First, DBSCAN selects a point p from the dataset. A range query, with center p and radius ϵ is applied to verify if the neighborhood of p contains at least a number $MinPts$ of points (i.e., it is dense). If so, these points are put in the same cluster as p and this process is iteratively applied again for the new points of the cluster. DBSCAN continues until the cluster cannot be further expanded; the whole dense region where p falls is discovered. The process is repeated for unvisited points until all clusters and outlier points have been discovered. OPTICS [11] is another density based method. It works similarly to DBSCAN but it does not compute the set of clusters. Instead, it outputs an ordering of the points in the dataset which is used in a second step to identify the clusters for various values of ϵ.

Although these methods can be used to discover snapshot clusters at a given timeslice of the history, they cannot be applied directly for the identification of moving clusters.

3.2 Clustering Spatio-temporal Data

Previous methods on clustering spatio-temporal data have focused on grouping trajectories of similar shape. The one-dimensional version of this problem is equivalent to clustering time-series that exhibit similar movements. Ref. [12] formalized a LCSS (Least Common Subsequence) distance, which assists the application of traditional clustering algorithms (e.g., partitioning, hierarchical, etc.) on object trajectories. In Ref. [2], regression models are used for clustering similar trajectories. Finally, Ref. [13,14] use traditional clustering algorithms on features of segmented time series. The problem of clustering similar trajectories or time-series is essentially different to that of finding moving clusters. The key difference is that a trajectory cluster has a constant set of objects throughout its lifetime, while the contents of a moving cluster may change over time. Another difference is that the input to a moving cluster discovery problem does

not necessarily include trajectories that span the same lifetime. Finally, we require the segments of trajectories that participate in a moving cluster to move similarly *and* to be close to each other in space.

A similar problem to the discovery of moving clusters is the identification of areas that remain dense in a long period of time. Ref. [1] proposed methods for discovering such regions in the future, given the locations and velocities of currently moving objects. This problem is different to moving clusters discovery in several aspects. First, it deals with the identification of static, as opposed to moving, dense regions. Second, a sequence of such static dense regions at consecutive timestamps does not necessarily correspond to a moving cluster, since there is no guarantee that there are common objects between regions in the sequence. Third, the problem refers to predicting dense regions in the future, as opposed to discovering them in a history of trajectories.

Our work is also related to the incremental maintenance of clusters in data warehouses. Many researchers have studied the incremental updating of association rules for data mining. Closer to our problem are the incremental implementations of DBSCAN [3] and OPTICS [15]. The intuition of both methods is that, due to the density-based nature of the clustering, the insertion or deletion of a new object o_j affects only the objects in the neighborhood of o_j. Updates are applied in batch and it is assumed that the updated objects are only a small percentage of the dataset. This is not true for our problem due to the movement of objects. Potentially, the entire dataset can be updated at each timestamp rendering the incremental maintenance of clusters prohibitively expensive. Another method, proposed by Nassar et. al. [4], minimizes the updating cost by employing *data bubbles* [16] which are approximate representations of clusters. The method attempts to redistribute the updated objects inside the existing data bubbles. It is not suitable for our problem, since it does not maintain the continuity of clusters in the time dimension.

4 Retrieval of Moving Clusters

In this section we describe three algorithms for the retrieval of moving clusters. The first one, *MC1*, is a straight forward implementation of the problem definition. The next algorithm, *MC2*, improves the efficiency by avoiding redundant checks. Finally, *MC3* is an approximate algorithm which trades accuracy for speed.

4.1 MC1: The Straight-Forward Approach

A direct method for retrieving moving clusters is to follow the problem definition. Starting from S_1, density-based clustering is applied at each timeslice and consecutive timeslice clusters are merged to moving clusters. A pseudocode for the algorithm is given in Figure 2.

The algorithm scans through the timeslices, maintaining at each step a set $\mathcal{G}$ of *current* moving clusters. When examining timeslice S_i, $\mathcal{G}$ includes the moving clusters containing a timeslice cluster in S_{i-1}. After discovering the timeslice clusters in S_i, every pair (g, c), $g \in \mathcal{G}$, $c \in S_i$ is checked to verify whether $g \circ c$ (i.e., g extended by c in S_i) forms a valid moving cluster. Clusters in $\mathcal{G}$ that were not extended at S_i,

Algorithm. MC1(Spatio-temporal history H, real ϵ, int $MinPts$, real θ)

```
1.  G := ∅; // set of current clusters
2.  for i:=1 to n // for each timestamp
3.      for each current moving cluster g ∈ G
4.          g.extended := false
5.      G_next := ∅; // next set of current clusters
6.      // retrieve timeslice clusters at S_i
7.      L := DBSCAN(S_i, e, MinPts);
8.      for each timeslice cluster c ∈ L
9.          assigned := false;
10.         for each current moving cluster g ∈ G
11.             if g ∘ c is a valid moving cluster then
12.                 g.extended := true;
13.                 G_next := G_next ∪ g ∘ c;
14.                 assigned := true;
15.         if (not assigned) then
16.             G_next := G_next ∪ c;
17.     for each current moving cluster g ∈ G
18.         if (not g.extended) then
19.             output g;
20.     G := G_next;
```

Fig. 2. The MC1 algorithm for computing moving clusters

are output. The $\mathcal{G}_{next}$ set of moving clusters to be used at the next iteration (i.e., for S_{i+1}) consists of (i) clusters in $\mathcal{G}$ that were extended in S_i and (ii) timeslice clusters $c \in S_i$ that were not used as extensions to some $g \in \mathcal{G}$. In this way, MC1 does not miss any cluster and does not output any redundant clusters, whose extensions are also valid clusters.

We assume that all points of the *current* timeslice S_i fit in memory. In practice this means that a relatively low-end computer with 1GB of RAM supports more than 10M points per timeslice. Notice that there is no restriction on the number of timeslices. Having the entire S_i in the main memory eliminates the need to build a spatial index in order to speed-up the clustering algorithm. Instead, we developed a main-memory version of DBSCAN. We divide the 2-D space into a grid where the cell size is $\frac{\epsilon}{\sqrt{2}} \times \frac{\epsilon}{\sqrt{2}}$ and hash each point of S_i to its corresponding cell. Observe that the distance between any two points in the same cell is at most ϵ. Therefore, any cell containing more than $MinPts$ points is part of a cluster; such cells are called *dense*. The main-memory DBSCAN proceeds by merging neighboring dense cells and finally it handles the remaining points which belong to sparse cells.

For each iteration of Line 3, we only need to keep one cluster g of $\mathcal{G}$ in the memory. Therefore, the memory required by MC1 is $O(|S_i| + |g| + \frac{\epsilon^2}{2})$. The if-statement in Line 11 is executed $|G| \cdot |L|$ times and calculates the similarity criterion of Definition 1 which involves an intersection and a union operation. We employ a hash table to implement these operations efficiently. The cost is $O(|g|+|c|)$, where g and c are clusters belonging to $\mathcal{G}$ and L, respectively.

4.2 MC2: Minimizing Redundant Checks

MC1 contains two time-consuming operations: the call to DBSCAN (line 7) and the computation of intersection/union of clusters (line 11). Especially the later exhibits significant redundancy since we expect each cluster $g \in \mathcal{G}$ to share objects only with a few clusters $c \in L$. In this section we present an improved algorithm, called *MC2* which minimizes the redundant combinations of (g, c).

The idea is simple: We select a random object $o_j \in g_i$ and search for it in all clusters of L. Let $c_i \in L$ be the cluster which contains o_j. Then we calculate the intersection and union only for the pair (g_i, c_i). If they satisfy the similarity criterion, we insert $g_i \circ c_i$ in the result. Else we must select another object $o_k \in g_i$ and repeat the process. Notice that some objects are pruned: o_k is selected from $g_i - c_i$ since the common objects of (g_i, c_i) cannot belong to any other cluster of L. The interesting observation is that we never need to test more than $(1-\theta)|g_i|$ points. The following lemma has the details:

Lemma 1. *Let c_1 and c_2 be clusters. c_1c_2 is* not *a moving cluster if $|c_1 - c_2| > (1-\theta)|c_1|$.*

Proof. For any set c_1, c_2, it holds that $|c_1 \cap c_2| \leq |c_1|$. We know that there are more than $(1-\theta)|c_1|$ points in c_1 which do not exist in c_2. Therefore, $|c_1 \cap c_2| < |c_1| - (1-\theta)|c_1|$. By using this value in the formula of Definition 1, we have:

$$\frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} < \frac{|c_1| - (1-\theta)|c_1|}{|c_1 \cup c_2|} \leq \frac{|c_1| - (1-\theta)|c_1|}{|c_1|} = \theta$$

Since $\dfrac{|c_1 \cap c_2|}{|c_1 \cup c_2|} < \theta$, c_1c_2 is not a moving cluster.

Figure 3 presents the pseudocode of MC2. The algorithm is similar to MC1, except that the expensive union/intersection between clusters (Line 15) is executed at most $(1-\theta)|g_i|$ times for every $g_i \in \mathcal{G}$. Notice that another potentially expensive operation is the search for object o_j in the clusters of L (Line 12). To implement this efficiently, at each timeslice S_i we generate a hash table which contains all objects of S_i. The cost is $O(|S_i|)$ on average. Then we can find an object in constant time, on average. The tradeoff is that we need $O(|S_i|)$ additional memory to store the hash table. If memory size is a concern, we can use an in-place algorithm (e.g., quicksort) to sort the objects of S_i and locate objects by binary search.

Note that if $\theta \leq \dfrac{1}{2}$ ties may appear during the generation of moving clusters; in this case, MC2 does not necessarily produce the same output as MC1. This is illustrated in the example of Figure 4, where $\theta = \dfrac{1}{3}$. The original cluster c_0 splits into two smaller clusters c_1 and c_2 at timeslice S_{i+1}. Both of them satisfy the criterion of Definition 1 since $\dfrac{|c_0 \cap c_1|}{|c_0 \cup c_1|} = \dfrac{|c_0 \cap c_2|}{|c_0 \cup c_2|} = \dfrac{1}{3}$. Therefore, both $\{c_0c_1, c_2\}$ and $\{c_0c_2, c_1\}$ are legal sets of moving clusters. The same behavior is also observed in the symmetric case, where two small clusters are merged into a larger one. Since MC1 and MC2 break such ties arbitrarily, the outputs may differ; nevertheless, the result of MC2 is *not* approximate.

Algorithm. MC2(Spatio-temporal history H, real ϵ, int $MinPts$, real θ)

```
1.  G := ∅; // set of current clusters
2.  for i:=1 to n // for each timestamp
3.      G_next := ∅; // next set of current clusters
4.      L := DBSCAN(S_i, e, MinPts); // retrieve timeslice clusters at S_i
5.      for each timeslice cluster c ∈ L
6.          c.assigned := false;
7.      for each current moving cluster g ∈ G
8.          g.extended := false;
9.          k := (1 − θ)|g|;
10.         while (k > 0)
11.             o_j is a random object of g;
12.             c := find(o_j inside L); // c ∈ L contains o_j
13.             if (o_j not found) then k := k − 1;
14.             else
15.                 if g ∘ c is a valid moving cluster then
16.                     g.extended := true;
17.                     G_next := G_next ∪ g ∘ c;
18.                     c.assigned := true;
19.                 k = k − |g − c|;
20.         if (not g.extended) then output g;
21.     for each cluster c ∈ L
22.         if (not c.assigned) then G_next := G_next ∪ c;
23.     G := G_next;
```

Fig. 3. The MC2 algorithm for computing moving clusters

4.3 MC3: Approximate Moving Clusters

Although MC2 avoids checking redundant combinations of clusters, it still needs to perform an expensive call to DBSCAN for each timeslice. In this section we present an alternative algorithm, called *MC3*, which decreases the execution time by minimizing the set of objects processed by DBSCAN. MC3 is an approximate algorithm which trades speed for accuracy. A cluster c may be approximate because: (i) there exists an object $o_j \in c$ which is a core object but has less than $MinPts$ objects in its ϵ-neighborhood (see [5] for details), or (ii) the distance of o_j from any other object in c is larger than ϵ.

MC3 works as follows: given a current moving cluster set $\mathcal{G}$ and a timeslice S_i, it maps all objects $o_j \in S_i$ to a set of clusters $\mathcal{G}'$. This mapping is done by assuming that all objects which are common in S_{i-1} and S_i remain in the same clusters, irrespectively of their new position in S_i. Any new objects appearing in S_i are assigned to the closest existing cluster within distance $\sqrt{2}\epsilon$, or they are considered as noise if no such cluster exists. Let L_1 be the subset of $\mathcal{G}'$ containing the clusters that do not overlap with each other, and $S_i' \subseteq S_i$ be a set containing these objects that do not belong to any cluster of L_1. The algorithm assumes that L_1 contains valid clusters and does not process them further. On the other hand, the objects in S_i' probably define some new clusters. Therefore, MC3 applies DBSCAN only on S_i'. The output of DBSCAN together with

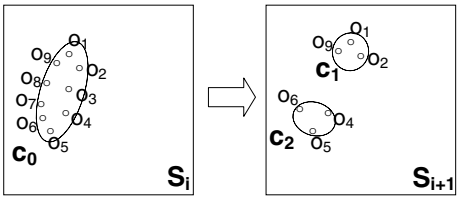


Fig. 4. Cluster split ($\theta = \frac{1}{3}$)

L_1 is the set L of clusters in timeslice S_i. Finally, the next set of moving clusters $\mathcal{G}_{next}$ is computed from $\mathcal{G}$ and L by employing the fast intersection technique of MC2. The details of MC3 are presented in Figure 5.

Algorithm. MC3(Spatio-temporal history H, real ϵ, int $MinPts$, real θ)

```
1.  G := ∅; // set of current clusters
2.  timer := 0;
3.  period := 1;
4.  for i:=1 to n // for each timestamp
5.      G_next := ∅; // next set of current clusters
6.      if (timer < period) then // Approximate clustering
7.          G' := Use G to assign all objects of S_i to clusters;
8.          L_1 := {g | g ∈ G' ∧ g is disjoint};
9.          S'_i := S_i − {all objects belonging to a cluster of L_1};
10.         L_2 := DBSCAN(S'_i, e, MinPts); // retrieve timeslice clusters at S'_i
11.         L := L_1 ∪ L_2;
12.         timer := timer + 1;
13.     else
14.         L := DBSCAN(S_i, e, MinPts); // retrieve timeslice clusters at S_i
15.         if ((deleted + inserted clusters in MC2) > α|G|) then
16.             period := min(1, period/2);
17.         else period := period + 1;
18.         timer := 0;
19.     Use the fast intersection method of MC2 to compute G_next;
20.     G := G_next;
```

Fig. 5. The MC3 algorithm for computing moving clusters

The intuition behind the algorithm is that several clusters will continue to exist in consecutive timeslices. Therefore, by employing the incremental approach of MC3, $|S'_i|$ (i.e., the input to the expensive DBSCAN procedure) is expected to be much smaller than $|S_i|$. Notice that S'_i can be computed with linear time complexity. First we create a hash table which maps objects to moving clusters of $\mathcal{G}$. We use this hash table to generate $\mathcal{G}'$; the entire process costs $O(|S_{i-1}| + |S_i|)$ on average. Next, we divide the space into a regular grid with cell size equal to $\epsilon \times \epsilon$ and we assign each object to its corresponding cell; the cost is $O(|S_i|)$. During this step we can identify if two clusters

intersect with each other. Then we check that (i) every cluster c is connected (i.e, all cells belonging to c *meet* each other) and (ii) no pair of clusters *meet* each other. The complexity of this step is $O(9\epsilon^2)$, where ϵ is constant. Figure 6 presents an example with 3 clusters $c_{1...3}$. c_1 is connected and does not meet or intersect with any other cluster; therefore it is placed in L_1. On the other hand, the objects of c_2 and c_3 must be added to S_i' because the two clusters meet.

Observe that in order to minimize the computation cost, we determine the relationships among clusters by considering only the corresponding grid cells. This introduces inaccuracies to the cluster identification process. For instance, the distance of an object $o_j \in c_1$ from any other object in c_1 may be up to $2\sqrt{2}\epsilon$, therefore o_j may need to be removed from c_1. Also, there may be some noise object o_k in the space between c_1 and c_2. By considering these two clusters together with o_k, c_1 and c_2 may need to be merged. When dealing with moving clusters which span several timeslices, an error at the initial assignment propagates to the following timeslice. Therefore, errors tend to accumulate and the quality of the result degrades fast.

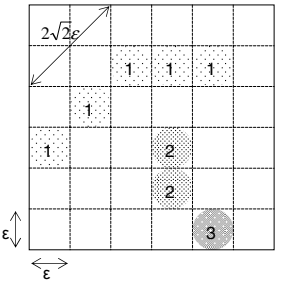
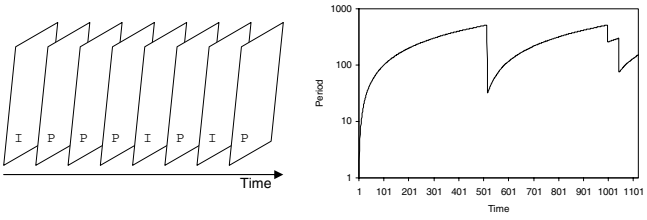


Fig. 6. Checking cluster intersection on an $\epsilon \times \epsilon$ grid

In order to minimize the approximation error, we use a method from video compression. The MPEG-2 encoder achieves high compression rates at high quality by using two types of video frames[1]: I-frames which are static JPEG images and P-frames which represent the difference between $frame_t$ and $frame_{t-1}$; in general I-frames are larger that P-frames. First an I-frame is sent followed by a stream of P-frames (Figure 7.a). When the encoding error exceeds some threshold (e.g., a different scene in the movie), a new I-frame is sent to increase the quality (i.e., eliminate the error). We employ a similar technique in our algorithm. Observe that the set L of clusters generated for each timeslice by MC3, is analogous to P-frames. To decrease the approximation error, we need to introduce periodically new reference cluster sets (i.e., similar to I-frames). We achieve this by interleaving the approximate clustering algorithm with exact clustering. This is shown in Line 14 of MC3, where DBSCAN is executed on the entire S_i. Notice that, contrary to MPEG-2, even after performing exact clustering the error is not necessarily eliminated. This is due to the fact that moving clusters span several timeslices, while exact clustering assigns correctly only the *static* clusters of the current timeslice.

[1] There is also a B-frame, which is not relevant to our problem.



(a) Example MPEG-2 frame sequence (b) Example period function.

Fig. 7. Example of MPEG-2 and TCP/IP-based period adjustment

Therefore, in order to recover from errors, we may need to execute exact clustering several times within a short interval. On the other hand, if exact clustering is performed too frequently, MC3 degenerates to MC2.

Obviously, the $period$ between calls to exact clustering cannot be constant, since the data distribution (and consequently the error), vary among timeslices. In order to adjust the $period$ adaptively, we borrowed an idea from the networks area and specifically from the TCP/IP protocol. When a node starts transmitting, TCP/IP assigns a medium transmission rate. If the percentage of packet loss is low, it means that there is spare capacity in the network; therefore the transmission rate is slowly increased. However, if the percentage of packet loss increases considerably, the transmission rate decreases quickly in order to resolve fast the congestion problem in the network. We employ a similar method: initially the $period$ for executing exact clustering is set to 1. If the error is low, the $period$ is increased linearly at each timestamp. In this way, as long as the error does not increase too much, the expensive clustering is performed infrequently. On the other hand, if the error exceeds some threshold, the $period$ is decreased exponentially (Figure 7.b). Therefore, the incorrect cluster assignments are not allowed to propagate through many timeslices.

The reader should note that we cannot perform exact clustering only when there is an error, because we cannot estimate the error unless we compute the exact clustering. The above mentioned method minimizes the unnecessary computations, but some redundancy is unavoidable. We estimate the error in the following way: when an exact clustering is performed, we count the number of moving clusters which are created or removed at the current timeslice. If this number is greater that $\alpha|\mathcal{G}|$, $0 < \alpha \leq 1$, we assume that the error is high. The intuition is that if many moving clusters are changing this may be due to the fact that there were many incorrect assignments at the previous timeslices. Other methods for error estimation are also possible. For instance, we could execute approximate and exact clustering at the same timeslice and compare the results. However, this would increase the execution time, while our heuristic poses minimal overhead and works well in practice.

5 Experimental Evaluation

In this section we present the experimental evaluation of our methods. Due to the unavailability of real datasets, we developed a generator which generates synthetic

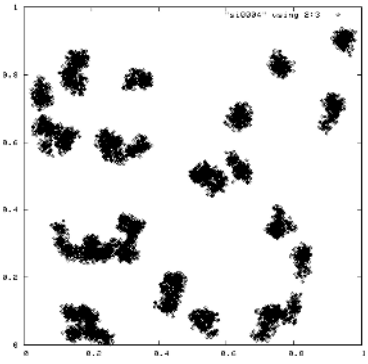


Fig. 8. Output from the generator: a sample timeslice S_i with 9000 objects

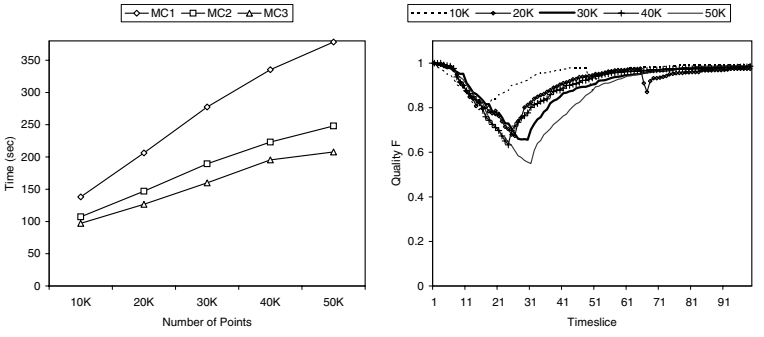
datasets with various distributions. The generator accepts several parameters including the number of clusters per timeslice, the average number of objects per cluster, the neighborhood radius ϵ and the density $MinPts$. The average velocity of the clusters and the change probability P_c are also given as input. The output of the generator is a series of timeslices. At each timeslice each cluster may move from the previous position; the velocity vector changes with probability P_c. With the same probability, a cluster may rotate around its center. Also, objects are inserted or deleted with probability P_c. Figure 8 shows a sample of the distribution of objects at a given timeslice. We generated several datasets by varying the number of objects per timeslice from 10K to 50K and the number of timeslices from 50 to 100. Therefore, the size of each dataset was between 500K and 5M objects. We implemented our algorithms in C++ and run all experiments on a relatively low-end Linux system with 1.2GB RAM and 1.3GHz CPU.

In order to evaluate the accuracy of MC3's approximation, we compared at each timeslice the current set of moving clusters produced by MC3 against the set generated by MC2. The quality of the solution is defined as:

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

This metric is commonly used in data mining [17]. Obviously, F is always 1 for MC1 and MC2, since their solution is exact.

In the first set of experiments we test the scalability of our methods to the size of the dataset. We generated datasets with 100 timeslices. Each timeslice contained 10K to 50K objects and 800 clusters on average, resulting to a database size of 1M to 5M objects. We set $\theta = 0.9$ and $\alpha = 0.1$ (recall that α is used in Line 15 of MC3). The results are shown in Figure 9. As expected, the execution time of all algorithms increases with the dataset size. MC2 is much faster than MC1 and the difference increases for larger datasets. This demonstrates clearly that MC2 manages to prune a large number of redundant cluster combinations. MC3 is faster than MC2 but there is some error introduced to the solution. In Figure 9.b we draw the quality F of MC3's solution for each timeslice. Notice that the quality is reduced at first, because the error is not high enough to trigger the execution of exact clustering. However, when the error exceeds the threshold value, the algorithm adjusts fast to high quality solutions.



(a) Execution time vs. database size (b) Quality F of MC3 at each timeslice

Fig. 9. Varying the number of objects. 800 clusters, $\theta = 0.9$, $\alpha = 0.1$

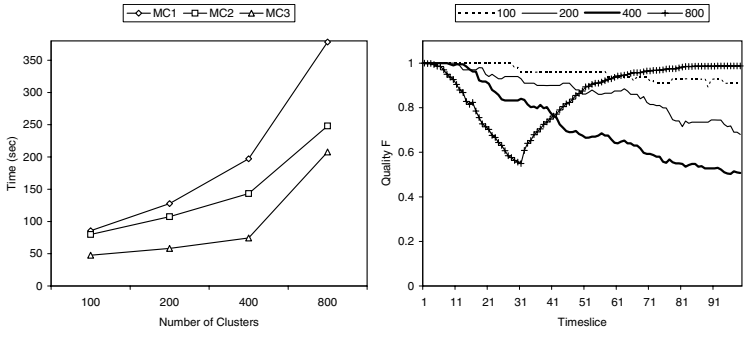
Table 1. Average quality $\overline{F}$ of MC3. 800 clusters, $\theta = 0.9$, $\alpha = 0.1$

	10K objects	20K objects	30K objects	40K objects	50K objects
$\overline{F}$	94.7%	91.1%	90.0%	90.8%	87.0%

The average quality of MC3 for the entire database is shown in Table 1. Given that the number of clusters remains the same for all datasets, when the database size increases, so does the average number of objects per clusters. When this happens, the extends of clusters tend to grow and therefore more errors are introduced due to incorrect assignment of cluster splits or noise objects (see Figure 6 for details). Nevertheless, the average quality remained at acceptable levels (i.e, at least 87%) in all cases.

The second set of experiments test the scalability of the algorithms to the number of clusters. All datasets contain 100 timeslices with 50K objects each (i.e., database size is 5M objects). The average number of clusters per timeslice is varied from 100 to 800. Again, we set $\theta = 0.9$ and $\alpha = 0.1$. The results are presented in Figure 10. The trend is similar to the previous experiment with the exception that the relative performance of MC3 has improved. To explain this, observe the graph of the quality F. For the case of 100 clusters, the quality remains high during the entire lifespan of the dataset. This happens because there are fewer clusters in the space, therefore there is a smaller probability of interaction among them which decreases the probability of errors in MC3. Consequently, the expensive exact clustering in Line 14 of MC3 is called very infrequently and the total execution time is improved.

Table 2 shows the average quality of MC3 for the entire dataset. Notice the strange trend: the quality first decreases and then increases again when there are more clusters. To understand this behavior, refer again to Figure 10.b. We already explained why quality is high for the 100 clusters dataset. On the other hand, when there are 800 clusters, there are a lot of interactions among them which introduce a large margin for error. Therefore, MC3 reaches the error threshold fast and starts performing exact clustering in order to recover. Now look at the 200 and 400 clusters datasets. There is a large number of clusters, therefore quality drops. Nevertheless, the error does not exceed



(a) Execution time vs. number of clus- (b) Quality F of MC3 at each timeslice
ters

Fig. 10. Varying the number of clusters. 50K objects, $\theta = 0.9$, $\alpha = 0.1$

Table 2. Average quality $\overline{F}$ of MC3. 50K objects, $\theta = 0.9$, $\alpha = 0.1$

	100 clusters	200 clusters	400 clusters	800 clusters
$\overline{F}$	95.7%	86.9%	72.7%	87.0%

the threshold and MC3 does not attempt to recover. The cause of this problem is the inappropriate value for parameter α.

In order to investigate further how parameter α affects the result, we used the 800 clusters, 50K objects dataset from the previous experiment and varied α from 0.05 to 0.2. The quality of MC3 for each timeslice, is shown in Figure 11. When α is low, MC3 reaches the error threshold fast. When this happens, it starts using the exact clustering function in order to recover fast. When α is larger, MC3 needs more time before initiating the recovery process. A point to note here is that for large values of α, the algorithm does not recover completely. For instance, if $\alpha = 0.2$, the algorithm starts oscillating around $F = 0.4$ after some time.

The execution time of MC3 also depends on α. This is expected, because the algorithm trades accuracy for speed. To test this, we generated two more datasets with the same characteristics as before, but with different object agility. Assuming that the previous dataset has *Medium* agility, one of the new datasets contains objects with *High* agility and the other contains objects with *Low* agility. The results are shown in Figure 12. As expected, MC3 is faster for larger values of α. This is due to the fact that when we accept higher error, MC3 uses most of the time the fast approximate function for clustering. Observe that MC3 is faster for low agility datasets. Such datasets contain fewer interactions among clusters; therefore the approximation error is low resulting to very few calls to the expensive exact clustering function.

Table 3 compares the execution time and the solution quality of MC3 against MC2 (recall that $F = 1$ always for MC2). In contrast to MC3, MC2 does not exhibit sensitivity to the agility of the dataset. Therefore, for the low agility dataset, the speedup of MC3 is very high. However, the average quality drops. To compare the various cases,

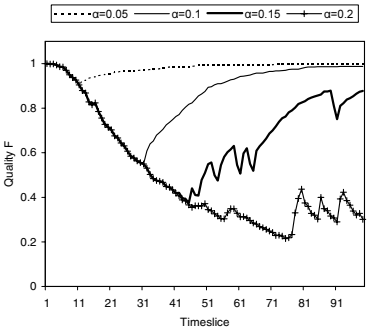


Fig. 11. Quality F of MC3 at each timeslice. 800 clusters, 50K objects, $\theta = 0.9$

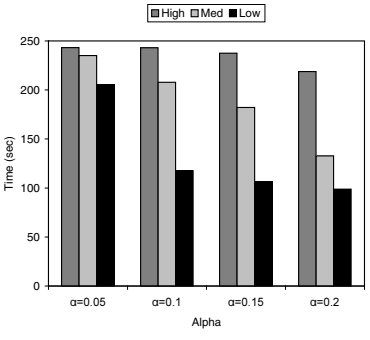


Fig. 12. Execution time of MC3 for varying α. 800 clusters, 50K objects, $\theta = 0.9$

we use the relative quality per time unit, defined as:

$$F(MC3|MC2) = \frac{\frac{F_{MC3}}{t_{MC3}} - \frac{1}{t_{MC2}}}{\frac{1}{t_{MC2}}}$$

Observe that $F(MC3|MC2)$ is much higher for the low agility dataset, meaning that MC3 traded a small percentage of accuracy in order to achieve a much lower execution time.

The last set of experiments investigates the effect of θ, which is the integrity threshold of Definition 1. We used the 800 clusters, 50K objects, medium agility dataset from the previous experiment and set $\alpha = 0.1$. We varied θ from 0.7 to 0.95. Figure 13 shows the execution time for the three algorithms. There is a very small increase of the execution time when θ increases. This is more obvious for MC1. The reason is that when θ is large, there is a smaller probability to find matching moving clusters in consecutive timeslices. To conclude that there is no match, MC1 must check all cluster pairs, whereas, the search would stop as soon as one match was found.

Figure 14 and Table 4 demonstrate the effect of theta on the quality of MC3. Interestingly, the quality improves when θ increases. This happens because, even if the

Table 3. Average quality of MC3 for datasets with varying agility

	MC2: Time (sec)	MC3: Time (sec)	MC3: Quality (F)	$F(MC3\|MC2)$
High	252	246	98.6%	1.2%
Med	248	208	87.0%	3.9%
Low	256	118	73.0%	59.0%

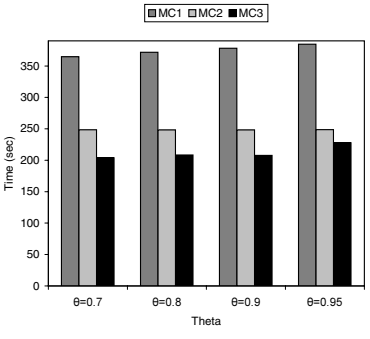


Fig. 13. Execution time vs. θ. 800 clusters, 50K objects, $\alpha = 0.1$

approximate clustering function generates some incorrect clusters, there is a high possibility that the corresponding moving clusters will not satisfy the θ criterion and will be eliminated. Therefore, there is a smaller probability of errors.

6 Conclusions

In this paper we investigated the problem of identifying moving clusters in large spatio-temporal datasets. The availability of accurate location information from embedded GPS devices, will enable in the near future numerous novel applications requiring the extraction of moving clusters. Consider, for example, a traffic control data mining system which diagnoses the causes of traffic congestion by identifying convoys of similarly moving vehicles. Military applications can also benefit by monitoring, for instance, the movement of groups of soldiers.

We defined formally the problem and proposed exact and approximate algorithms to identify moving clusters. MC2 is an efficient algorithm which can be used if 100% accuracy is essential. MC3, on the other hand, generates faster an approximate solution. In order to minimize the approximation error without sacrificing the efficiency, we borrowed methods from MPEG-2 and TCP/IP. Our experimental results demonstrate the applicability of our methods to large datasets with varying object distribution and agility. To the best of our knowledge, this is the first work to focus on the automatic extraction of moving clusters.

The efficiency and accuracy of MC3 depends on the appropriate selection of parameter α and the accurate estimation of error. Currently we are working on a self-tuning

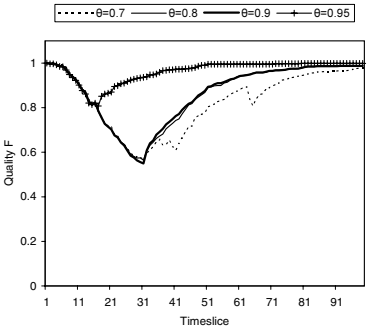


Fig. 14. Quality F of MC3 vs. θ. 800 clusters, 50K objects, $\alpha = 0.1$

Table 4. Average quality of MC3 for varying θ

	$\theta = 0.7$	$\theta = 0.8$	$\theta = 0.9$	$\theta = 0.95$
MC3: Quality ($\overline{F}$)	83.0%	86.8%	87.0%	96.5%
$F(MC3\|MC2)$	1.1%	3.4%	3.9%	5.3%

method for parameter selection. We also plan to explore sophisticated methods for error estimation.

References

1. Hadjieleftheriou, M., Kollios, G., Gunopulos, D., Tsotras, V.J.: On-line discovery of dense areas in spatio-temporal databases. In: Proc. of SSTD. (2003)
2. Gaffney, S., Smyth, P.: Trajectory clustering with mixtures of regression models. In: Proc. of ICDM. (1999) 63–72
3. Ester, M., Kriegel, H.P., Sander, J., Wimmer, M., Xu, X.: Incremental clustering for mining in a data warehousing environment. In: Proc. of VLDB. (1998) 323–333
4. Nassar, S., Sander, J., Cheng, C.: Incremental and effective data summarization for dynamic hierarchical clustering. In: Proc. of ACM SIGMOD. (2004) 467–478
5. Martin, E., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. of KDD. (1996)
6. Kaufman, L., Rousueeuw, P.: Finding Groups in Data: an Introduction to Cluster Analysis. John Wiley and Sons (1990)
7. Ng, R.T., Han, J.: Efficient and effective clustering methods for spatial data mining. In: Proc. of VLDB. (1994)
8. Zhang, T., Ramakrishnan, R., Livny, M.: BIRCH: An efficient data clustering method for very large databases. In: Proc. of ACM SIGMOD. (1996)
9. Guha, S., Rastogi, R., Shim, K.: CURE: An efficient clustering algorithm for large databases. In: Proc. of ACM SIGMOD. (1998)
10. Nanopoulos, A., Theodoridis, Y., Manolopoulos, Y.: C2P: Clustering based on closest pairs. In: Proc. of VLDB. (2001)

11. Ankerst, M., Breunig, M., Kriegel, H.P., Sander, J.: OPTICS: Ordering points to identify the clustering structure. In: Proc. of ACM SIGMOD. (1999) 49–60
12. Vlachos, M., Kollios, G., Gunopulos, D.: Discovering similar multidimensional trajectories. In: Proc. of ICDE. (2002) 673–684
13. Das, G., Lin, K.I., Mannila, H., Renganathan, G., Smyth, P.: Rule discovery from time series. In: Proc. of KDD. (1998) 16–22
14. Li, C.S., Yu, P.S., Castelli, V.: Malm: A framework for mining sequence database at multiple abstraction levels. In: Proc. of CIKM. (1998) 267–272
15. Kriegel, H.P., Kröoger, P., Gotlibovich, I.: Incremental OPTICS: Efficient computation of updates in a hierarchical cluster ordering. In: Proc. of DaWaK. (2003) 224–233
16. Breunig, M.M., Kriegel, H.P., Kröger, P., Sander, J.: Data bubbles: Quality preserving performance boosting for hierarchical clustering. In: Proc. of ACM SIGMOD. (2001)
17. Larsen, B., Aone, C.: Fast and effective text mining using linear-time document clustering. In: Proc. of KDD. (1999) 16–22