International Journal of Information Technology & Decision Making


# ALL IN ONE: MINING MULTIPLE MOVEMENT PATTERNS

NHATHAI PHAN
*Information Systems Department, New Jersey Institute of Technology*
*University Heights, Newark, NJ 07102-1982, USA*

PASCAL PONCELET
*Lirmm Laboratory, University Montpellier 2*
*161 Rue Ada 34095, Montpellier, Cedex 5, France*

MAGUELONNE TEISSEIRE
*Tetis Laboratory, Irstea Montpellier*
*500 Rue Jean-Francois Breton 34093, Montpellier, Cedex 5, France*

Recent improvements in positioning technology have led to a much wider availability of massive moving object data. A crucial task is to find the moving objects that travel together. In common, these object sets are called object movement patterns. Due to the emergence of many different kinds of object movement patterns in recent years, different approaches have been proposed to extract them. However, each approach only focuses on mining a specific kind of patterns. It is costly and time consuming to mine and manage various number of patterns, since we have to execute a large number of different pattern mining algorithms. Moreover, we have to execute these algorithms again whenever new data are added to the existing database. To address these issues, we first redefine movement patterns in the itemset context. Secondly, we propose a unifying approach, named *GeT_Move*, which uses a frequent closed itemset-based object movement pattern-mining algorithm to mine and manage different patterns. GeT_Move is developed in two versions which are GeT_Move and Incremental GeT_Move. To optimize the efficiency and to free the parameters setting, we further propose a Parameter Free Incremental GeT_Move algorithm. Comprehensive experiments are performed on real and large synthetic datasets to demonstrate the effectiveness and efficiency of our approaches.

*Keywords*: Object movement pattern; frequent closed itemset; unifying approach; trajectories.

## 1. Introduction

Nowadays, many electronic devices are used for real world applications. Telemetry attached on wildlife, GPS installed in cars, sensor networks, and mobile phones have enabled the tracking of almost any kind of data and has led to an increasingly large amount of data that contains moving objects and numerical data. Therefore, analysis on such data to find interesting patterns is attracting increasing attention for applications such as movement pattern analysis, animal behavior study, route

planning and vehicle control, location prediction, location-based services.

Early approaches designed to recover information from spatio-temporal datasets included ad-hoc queries aimed as answering queries concerning a single predicate range or nearest neighbour. For instance, *"finding all the moving objects inside area A between 10:00 am and 2:00 pm"* or *"how many cars were driven between Main Square and the Airport on Friday"*[29]. Spatial query extensions in GIS applications are able to run this type of query. However, these techniques are used to find the best solution by exploring each spatial object at a specific time according to some metric distance measurement (usually Euclidean). As results, it is difficult to capture collective behaviour and correlations among the involved entities using this type of queries.

Recently, there has been growing interest in the querying of patterns which capture *'group'* or *'common'* behaviour among moving entities. This is particularly true for the development of approaches to identify groups of moving objects for which a strong relationship and interaction exist within a defined spatial region during a given time duration. Some examples of these patterns are flocks[1,2,14,15], moving clusters[4,7,18], convoy queries[3,16], stars and k-stars[17], closed swarms[6,13], group patterns[21], periodic patterns[25], co-location patterns[22], TraLus[23], and so on.

To extract these kinds of patterns, different algorithms have been proposed. Naturally, the computation is costly and time consuming because we need to execute different algorithms consecutively. However, if we had an algorithm which could extract different kinds of patterns, the computation cost will be significantly reduced and the process would be much less time consuming. Since the results are comparative, it is beneficial for analysts to better exploit and understand the object movement behavior.

In some applications (e.g., cars), object locations are continuously reported by using Global Positioning System (GPS). Therefore, new data is always available. If we do not have an incremental algorithm, we need to execute again and again the algorithms on the whole database including existing data and new data to extract patterns. This is of course, cost-prohibitive and time consuming as well. An incremental algorithm can indeed improve the process by combining the results extracted from the existing data and the new data to obtain the final results.

With the above issues in mind, we propose *GeT_Move* a unifying incremental object movement pattern-mining approach. Part of this approach is based on an existing state-of-the-art algorithm which is extended to take advantage of well-known frequent closed itemset (FCI) mining algorithms. In order to apply it, we first redefine object movement patterns in an itemset context. Secondly, we propose a spatio-temporal matrix to describe original data and then an incremental FCI-based object movement pattern-mining algorithm to extract patterns.

In fact, obtaining the optimal parameters is a difficult task for most of algorithms which require parameter setting. Even if we are able to obtain the optimal parameters after doing many executions and result evaluations on a dataset, the optimal values of parameters will be different on other datasets. To address this

issue, we propose a parameter free incremental GeT_Move. The basic idea is to rearrange the input data based on nested concept[31] so that incremental GeT_Move can automatically extract patterns without parameter setting.

This work is extended from the state-of-the-art works[32,33]. The main contributions of this paper are summarized below.

- We re-define the object movement patterns in the itemset context which enables us to effectively extract different kinds of movement patterns.
- We present incremental approaches, named *GeT_Move* and *Incremental GeT_Move*, which efficiently extract frequent closed itemsets from which object movement patterns are retrieved.
- We propose a parameter free Incremental GeT_Move. The advantages of this approach is that it does not require the parameter setting and automatically extract patterns efficiently.
- To deal with the arriving of new data, we propose an explicit combination of pairs of FCIs-based pattern mining algorithm. This approach combines the results in the existing database with the arriving data to obtain the final results.
- Conducted experimental results on real and large synthetic datasets demonstrate that our techniques enable us to effectively extract different kinds of patterns. Furthermore, our approaches are more efficient compared to state-of-the-art algorithms in most of cases.
- The system is attached into a state-of-the-art research which is available to the public[33].

The remaining sections of the paper are organized as follows. Section 2 discusses preliminary definitions of the object movement patterns as well as the related work. The patterns such as swarms, closed swarms, convoys, and group patterns are redefined in an itemset context in Section 3. We present our approaches in Section 4. Experiments testing effectiveness and efficiency are shown in Section 5. Finally, we draw our conclusions in Section 6.

## 2. Object Movement Patterns

In this section, we briefly propose an overview of the main object movement patterns. Then we discuss the related work.

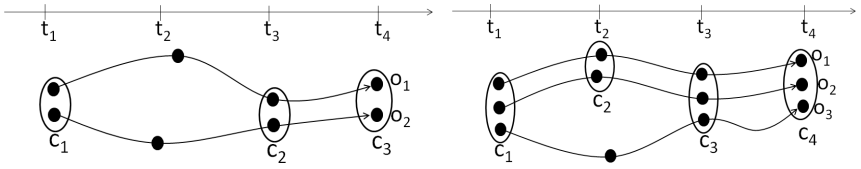
### 2.1. *Preliminary Definitions*

Object movement pattern mining has been extensively addressed over the last years. Basically, an object movement pattern is designed to group similar trajectories or objects which tend to move together during a time interval. In recent years, many different pattern definitions have been defined such as ocks[1,2,14,15], convoys[3,16], swarms and closed swarms[6,13], moving clusters[4,7,18], and periodic patterns[25].

In this paper, we focus on proposing a unifying approach to extract all these

Table 1. An example of a moving object database.

| Objects $O_{DB}$ | Timesets $T_{DB}$ | $x$ | $y$ |
|---|---|---|---|
| $o_1$ | $t_1$ | 2.3 | 1.2 |
| $o_2$ | $t_1$ | 2.1 | 1 |
| $o_1$ | $t_2$ | 10.3 | 28.1 |
| $o_2$ | $t_2$ | 0.3 | 1.2 |




(a) Swarm (b) Convoy

Fig. 1. An example of swarm and convoy where $c_1, c_2, c_3, c_4$ are clusters gathering closed objects together at specific timestamps.

different kinds of patterns. Let us assume that we have a group of moving objects $O_{DB} = \{o_1, o_2, \ldots, o_z\}$, a set of timestamps $T_{DB} = \{t_1, t_2, \ldots, t_n\}$ and at each timestamp $t_i \in T_{DB}$, spatial information[a] $x, y$ for each object. For example, Table 1 illustrates an example of a moving object database. In object movement pattern mining, we are interested in extracting a group of objects which stay together during a period. Therefore, from now, $O = \{o_{i_1}, o_{i_2}, \ldots, o_{i_p}\}(O \subseteq O_{DB})$ is used to indicate a group of objects, $T = \{t_{a_1}, t_{a_2}, \ldots, t_{a_m}\}(T \subseteq T_{DB})$ is the set of timestamps within which the objects stay together. Let $\varepsilon$ be the user-defined threshold standing for a minimum number of objects and $min_t$ the minimum number of timestamps. Thus $|O|$ (resp. $|T|$) must be greater than or equal to $\varepsilon$ (resp. $min_t$). In the following, we formally define all the different kinds of movement patterns.

Informally, a *swarm* is a group of moving objects $O$ containing at least $\varepsilon$ individuals which are closed each other for at least $min_t$ timestamps. In a swarm pattern, the $min_t$ timestamps could be non-consecutive entirely. A swarm can be formally defined as follows:

**Definition 2.1.** *Swarm*[6]. A pair $(O, T)$ is a swarm if:

$$\begin{cases} (1) : \forall t_{a_i} \in T, \exists c \text{ s.t. } O \subseteq c, \ c \text{ is a cluster} \\ (2) : |O| \geq \varepsilon \\ (3) : |T| \geq min_t \end{cases} \tag{2.1}$$

Note that the meaning of the above conditions are: (1) there is at least one cluster which contains all the objects in $O$ at each timestamp in $T$, (2) there must be at least $\varepsilon$ objects, (3) there must be at least $min_t$ timestamps in $T$.

[a]Spatial information can be for instance GPS location.

For example, as shown in Figure 1a, if we set $\varepsilon = 2$ and $min_t = 2$, we can find the following swarms $(\{o_1, o_2\}, \{t_1, t_3\})$, $(\{o_1, o_2\}, \{t_1, t_4\})$, $(\{o_1, o_2\}, \{t_3, t_4\})$, $(\{o_1, o_2\}, \{t_1, t_3, t_4\})$. We can note that these swarms are redundant since they can be grouped together in the following swarm $(\{o_1, o_2\}, \{t_1, t_3, t_4\})$.

To avoid this redundancy, Zhenhui Li et al.[6] propose the notion of *closed swarm* for grouping together both objects and timstamps. A swarm $(O, T)$ is *object-closed* if, when fixing $T$, $O$ cannot be enlarged. Similarly, a swarm $(O, T)$ is *time-closed* if, when fixing $O$, $T$ cannot be enlarged. Finally, a swarm $(O, T)$ is a closed swarm if it is both object-closed and time-closed.

**Definition 2.2.** *Closed Swarm*[6]. A pair $(O, T)$ is a closed swarm if:

$$\begin{cases} (1) : (O, T) \textit{ is a swarm} \\ (2) : \nexists O' \textit{ s.t. } (O', T) \textit{ is a swarm and } O \subset O' \\ (3) : \nexists T' \textit{ s.t. } (O, T') \textit{ is a swarm and } T \subset T' \end{cases} \tag{2.2}$$

For instance, in the previous example, $(\{o_1, o_2\}, \{t_1, t_3, t_4\})$ is a closed swarm.

A *convoy* is a group of objects such that these objects are close each other during at least $min_t$ time points. The main difference between convoys and swarms (or closed swarms) is that a convoy lifetimes must be consecutive. In essence, by adding the consecutiveness condition to swarms, we can define convoys as follows:

**Definition 2.3.** *Convoy*[3]. A pair $(O, T)$, is a convoy if:

$$\begin{cases} (1) : (O, T) \textit{ is a swarm} \\ (2) : \forall i, 1 \leq i < |T|,\ t_{a_{i+1}} = t_{a_i} + 1 \end{cases} \tag{2.3}$$

For instance, in Figure 1b, with $\varepsilon = 2, min_t = 2$ we have two convoys which are $(\{o_1, o_2\}, \{t_1, t_2, t_3, t_4\})$ and $(\{o_1, o_2, o_3\}, \{t_3, t_4\})$. In this paper, we not only consider *maximal convoys*[3] but also *valid convoys*[34]. Similar to swarm and closed swarm, a convoy becomes a valid convoy if it cannot be enlarged in terms of objects and timestamps.

Until now, we have considered that we have a group of objects that move close to each other for a long time interval. For instance, as shown in Ref. 28, moving clusters and different kinds of flocks virtually share essentially the same definition. Basically, the main difference is the use of clustering techniques. Flocks, for instance, usually consider a rigid definition of the radius while moving clusters and convoys apply a density-based clustering algorithm (e.g., DBScan[5]). Moving clusters can be seen as special cases of convoys with the additional condition that they need to share some objects between two consecutive timestamps[28]. Therefore, in the following, for brevity and clarity sake, we will mainly focus on convoy mining and density-based clustering algorithm.

According to the previous definitions, the main difference between convoys and swarms is about the consecutiveness and non-consecutiveness of clusters during a

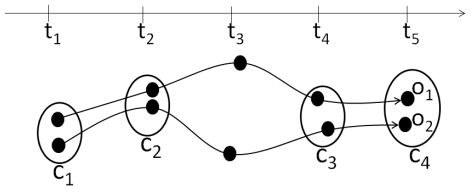



Fig. 2. A group pattern example.

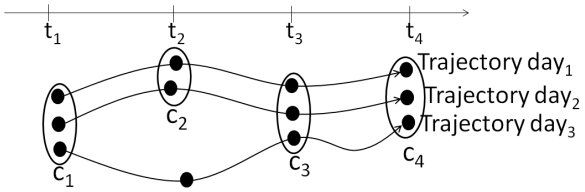



Fig. 3. A periodic pattern example.

time interval. In Ref. 21, Hwang et al. propose a general pattern, called a *group pattern*, which essentially is a combination of both convoys and closed swarms. In basic, a group pattern is a set of disjointed convoys which are generated by the same group of objects in different time intervals. By considering a convoy as a timepoint, a group pattern can be seen as a swarm of disjointed convoys. In addition, a group pattern cannot be enlarged in terms of objects and number of convoys. Therefore, a group pattern essentially is a closed swarm of disjointed convoys. In formal, group patterns can be defined as follows:

**Definition 2.4.** *Group Pattern*[21]. Given a set of objects $O$, a minimum weight threshold $min_{wei}$, a set of disjointed convoys $T_{\mathcal{S}} = \{s_1, s_2, \ldots, s_n\}$, a minimum number of convoys $min_c$. $(O, T_{\mathcal{S}})$ is a group pattern if:

$$\begin{cases} (1) : (O, T_{\mathcal{S}}) \text{ is a closed swarm } min_c \text{ w.r.t } \varepsilon \\ (2) : \frac{\sum_{i=1}^{|T_{\mathcal{S}}|} |s_i|}{|T_{DB}|} \geq min_{wei} \end{cases} \tag{2.4}$$

Note that $min_c$ is only applied for $T_{\mathcal{S}}$ (i.e., $|T_{\mathcal{S}}| \geq min_c$)

For instance, see Figure 2, with $min_t = 2$ and $\varepsilon = 2$, we have a set of convoys $T_{\mathcal{S}} = \{(\{o_1, o_2\}, \{t_1, t_2\}), (\{o_1, o_2\}, \{t_4, t_5\})\}$. With $min_c = 1$, we have $(\{o_1, o_2\}, T_{\mathcal{S}})$ is a closed swarm of convoys because $|T_{\mathcal{S}}| = 2 \geq min_c$, $|O| \geq \varepsilon$ and $(O, T_{\mathcal{S}})$ cannot be enlarged. Furthermore, with $min_{wei} = 0.5$, $(O, T_{\mathcal{S}})$ is a group pattern since $\frac{|[t_1,t_2]|+|[t_4,t_5]|}{|T_{DB}|} = \frac{4}{5} \geq min_{wei}$.

We have overviewed patterns in which a group of objects move together during some time intervals. Furthermore, mining patterns from individual object movement is also interesting. In Ref. 25, N. Mamoulis et al. propose the notion of periodic pattern in which an object follows the same routes (approximately) over regular

time intervals. For example, people wake up at the same time and generally follow the same route to their work everyday. Informally, given an object's trajectory including $\mathcal{N}$ time-points, $\mathcal{T}_\mathcal{P}$ which is the number of timestamps that a pattern may re-appear. The object's trajectory is decomposed into $\lfloor \frac{\mathcal{N}}{\mathcal{T}_\mathcal{P}} \rfloor$ sub-trajectories. $\mathcal{T}_\mathcal{P}$ is data-dependent and has no definite value. For instance, $\mathcal{T}_\mathcal{P}$ can be set to 'a day' in traffic control applications since many vehicles have daily patterns. Annual animal migration patterns can be discovered by $\mathcal{T}_\mathcal{P}$ = 'a year'. In Figure 3, an object's trajectory is decomposed into daily sub-trajectories.

In essence, a periodic pattern is a closed swarm discovered from $\lfloor \frac{\mathcal{N}}{\mathcal{T}_\mathcal{P}} \rfloor$ sub-trajectories. For instance, in Figure 3, we have 3 daily sub-trajectories and from them we extract the two following periodic patterns: $\{c_1, c_2, c_3, c_4\}$ and $\{c_1, c_3, c_4\}$. The main difference in periodic pattern mining is the data preprocessing step. While the definition is similar to that of closed swarms. As we have provided the definition of closed swarms, we will mainly focus on closed swarm mining below.

### 2.2. *Related Work*

As mentioned before, many approaches have been proposed to extract patterns. The interested readers may refer to Ref. 20, 28 where short descriptions of the most efficient or interesting patterns and approaches are proposed. For instance, Gudmundsson and van Kreveld[1], Vieira et al.[2] define a flock pattern, in which the same set of objects stay together in a circular region with a predefined radius, Kalnis et al.[4] propose the notion of *moving cluster*, while Jeung et al.[3] define a convoy pattern.

Jeung et al.[3] adopt the DBScan algorithm[5] to find candidate convoy patterns. The authors propose three algorithms which incorporate trajectory simplification techniques in the first step. The distance measurements are performed on trajectory segments of as opposed to point based distance measurements. Another problem is related to the trajectory representation. Some trajectories may have missing timestamps or are measured at different time intervals. Therefore, the density measurements cannot be applied between trajectories with different timestamps. To address the problem of missing timestamps, the authors proposed to interpolate the trajectories either by creating virtual time points or by applying density measurements on trajectory segments. In addition, the convoy is defined as a maximal pattern when it has at least $k$ clusters during $k$ consecutive timestamps. To accurate the discovery of convoys, H. Yoon et al. propose the notion of *valid convoy*[34] which cannot be enlarged in terms of timestamps and objects. In this paper, we focus on valid convoy mining since it is more general.

Recently, Zhenhui Li et al.[6] propose the concept of swarm and closed swarm. An algorithm, named *ObjectGrowth*, has been designed to extract closed swarms. The ObjectGrowth method is a depth-first-search framework based on the objectset search space (i.e., the collection of all subsets of $O_{DB}$). For the search space of $O_{DB}$, they perform a depth-first search of all subsets of $O_{DB}$ through a pre-order

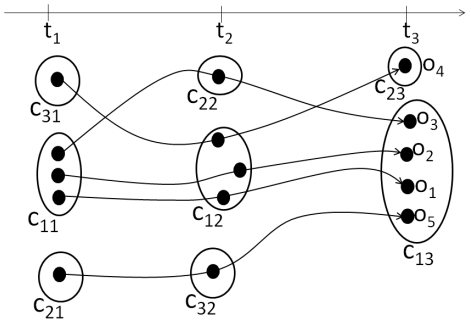



Fig. 4. An illustrative example. Note that clusters can be overlapped in the same timestamp.

Table 2. A Cluster Matrix. Note that clusters can be overlapped in the same timestamp.

| $T_{DB}$ | | $t_1$ | | | $t_2$ | | | $t_3$ | |
|---|---|---|---|---|---|---|---|---|---|
| Clusters $C_{DB}$ | | $c_{11}$ | $c_{21}$ | $c_{31}$ | $c_{12}$ | $c_{22}$ | $c_{32}$ | $c_{13}$ | $c_{23}$ |
| $O_{DB}$ | $o_1$ | 1 | | | 1 | | | 1 | |
| | $o_2$ | 1 | | | 1 | | | 1 | |
| | $o_3$ | 1 | | | | 1 | | 1 | |
| | $o_4$ | | | 1 | 1 | | | | 1 |
| | $o_5$ | | 1 | | | | 1 | 1 | |

tree traversal. Even though, the search space remains still huge for enumerating the objectsets in $O$ $(2^{|O_{DB}|})$. To speed up the search process, they propose two pruning rules. The first pruning rule, called *Apriori Pruning*, is used to stop traversal the subtree when we find further traversal that cannot satisfy mint. The second pruning rule, called Backward Pruning, makes use of the closure property. It checks whether there is a superset of the current objectset, which has the same maximal corresponding timeset as that of the current one. If so, the traversal of the subtree under the current objectset is meaningless. After pruning the invalid candidates, the remaining ones may or may not be closed swarms. Then a *Forward Closure Checking* is used to determine whether a pattern is a closed swarm or not.

In Ref. 21, Hwang et al. propose two algorithms to mine group patterns, known as the *Apriori-like Group Pattern Mining* algorithm and *Valid Group-Growth* algorithm. The former explores the Apriori property of valid group patterns and extends the Apriori algorithm[11] to mine valid group patterns. The latter is based on idea similar to the FP-growth algorithm[27]. Recently in Ref. 29, A. Calmeron proposes a frequent itemset-based approach for flock identification purposes.

Even if these approaches are very efficient they suffer the problem that they only extract a specific kind of patterns. When considering a dataset, it is difficult to know in advance which kind of patterns embedded in the data. Therefore proposing an approach able to automatically extract all these different kinds of patterns can be very useful since the results are comparative. This is the challenging issue we

address in this paper. Our algorithms will extract multiple movement patterns based on frequent itemset mining techniques. One of related techniques is biclustering. Many biclustering algorithms have been successfully applied to gene expression data to discover local patterns, in which a subset of genes exhibit similar expression levels over a subset of conditions. Even biclustering algorithms aim at discover local patterns, they can be adopt to extract frequent closed itemsets given two-dimensional binary matrices. The concept of biclustering was first introduced by Hartigan[38], and applied to gene expression data by Cheng and Church[39]. Many other such algorithms have been published, and popular algorithms are Cheng and Church[39], Plaid[40], OPSM[41], ISA[42], BiMax[43], and FCPMiner[44]. *Cheng and Church* and *OPSM* are deterministic greedy algorithms that seek to find the biclusters either with low variance, as defined by the mean squared residue, or with ordered rows. *BiMax* is a divide and conquer algorithm that seeks the rectangles of 1's in a binary matrix. *FCPMiner* algorithm recursively builds up the frequent closed patterns. This is done by taking the consecutive rows one-by-one and recording only those column indices that show the same changing tendency (same or exactly the opposite). Then the closeness of the candidate pattern is checked before the method is calling itself with the updated parameters.

## 3. Object Movement Patterns in Itemset Context

Extracting different kinds of patterns requires the use of several algorithms. To address this issue, we propose a unifying approach to extract and manage the patterns.

**Database of clusters.** A database of clusters, $C_{DB} = \{C_{t_1}, \ldots, C_{t_m}\}$, is a collection of snapshots of the moving object clusters at timestamps $\{t_1, \ldots, t_m\}$. Note that an object could belong to several clusters at one timestamp (i.e., cluster overlapping). Given a cluster $c \in C_{DB}$ and $c \subseteq O_{DB}$, $|c|$ and $t(c)$ are respectively used to denote the number of objects belonging to cluster $c$ and the timestamp that $c$ is involved in. To make our framework more general, we take clustering as a preprocessing step. The clustering methods could be different based on various scenarios. We leave the details of this step in the Appendix *A. Obtaining Clusters.*

In basic, patterns are evolution of clusters over time. Therefore, to manage the evolution of clusters, we need to analyse the correlations between them. Furthermore, if clusters share some characteristics (e.g. share some objects), they could be a pattern. Consequently, if a cluster is considered as an item we can have a set of items (called itemset). The key issue essentially is to efficiently combine items (clusters) to find itemsets (a set of clusters) which share some characteristics or satisfy some properties to be considered as a movement pattern. To describe the cluster evolution, moving object data is presented as a cluster matrix from which patterns can be extracted.

**Definition 3.1.** *Cluster Matrix.* Assume that we have a set of clusters $C_{DB}$. A cluster matrix is thus a matrix of size $|O_{DB}| \times |C_{DB}|$ such that each row represents an object and each column represents a cluster. The value of the cluster matrix cell,

$(o_i, c_j)$ is 1 (resp. empty) if $o_i$ is in (resp. not in) cluster $c_j$. An item $c_j$ is a cluster formed by applying clustering techniques.

For instance, the data from our illustrative example (Figure 4) is presented in a cluster matrix in Table 2. Object $o_1$ belongs to the cluster $c_{11}$ at timestamp $t_1$. For clarity reasons, in the following, $c_{ij}$ represents the cluster $c_i$ at time $t_j$. Therefore, the matrix cell ($o_1$-$c_{11}$) is 1, meanwhile the matrix cell ($o_4$-$c_{11}$) is empty because object $o_4$ does not belong to cluster $c_{11}$.

By presenting data in a cluster matrix, each object acts as a transaction while each cluster $c_j$ stands for an item. In addition, an itemset can be formed as $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$ with life time $T_\Upsilon = \{t_{a_1}, t_{a_2}, \ldots, t_{a_p}\}$ where $t_{a_1} < t_{a_2} < \ldots < t_{a_p}$, $\forall a_i : t_{a_i} \in T_{DB}, c_{t_{a_i}} \in C_{a_i}$. The support of the itemset $\Upsilon$, denoted $\sigma(\Upsilon)$, is the number of common objects in every items belonging to $\Upsilon$, $O(\Upsilon) = \bigcap_{i=1}^{p} c_{t_{a_i}}$. Additionally, the length of $\Upsilon$, denoted $|\Upsilon|$, is the number of items or timestamps $(= |T_\Upsilon|)$.

For instance, in Table 2, for a support value of 2 we have: $\Upsilon = \{c_{11}, c_{12}\}$ veryfying $\sigma(\Upsilon) = 2$. Every items (resp. clusters) of $\Upsilon, c_{11}$ and $c_{12}$, are in the transactions (resp. objects) $o_1, o_2$. The length of $|\Upsilon|$ is the number of items $(= 2)$.

In fact, the number of clusters can be large. However, the maximum length of itemsets is $T_{DB}$. Since, we are working on movement pattern context and thus clusters at the same timestamp cannot be in the same itemsets. Now, we will define some useful properties to extract the patterns which have been presented in Section 2 from frequent itemsets as follows:

Property 3.1. *Swarm.* Given a frequent itemset $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$. $(O(\Upsilon), T_\Upsilon)$ is a swarm if, and only if:

$$\begin{cases} (1) : \sigma(\Upsilon) \geq \varepsilon \\ (2) : |\Upsilon| \geq min_t \end{cases} \tag{3.5}$$

**Proof.** After construction, we have $\sigma(\Upsilon) \geq \varepsilon$ and $\sigma(\Upsilon) = |O(\Upsilon)|$ then $|O(\Upsilon)| \geq \varepsilon$. In addition, as $|\Upsilon| \geq min_t$ and $|\Upsilon| = |T_\Upsilon|$ then $|T_\Upsilon| \geq min_t$. Furthermore, $\forall t_{a_j} \in T_\Upsilon, O(\Upsilon) \subseteq c_{t_{a_j}}$, means that at any timestamps, we have a cluster containing all objects in $O(\Upsilon)$. Consequently, $(O(\Upsilon), T_\Upsilon)$ is a swarm because it satisfies all the requirements of the *Definition 2.1.* □

For instance, in Figure 5, for the frequent itemset $\Upsilon = \{c_{11}, c_{13}\}$ we have $(O(\Upsilon) = \{o_1, o_2, o_3\}, T_\Upsilon = \{t_1, t_3\})$ which is a swarm with support threshold $\varepsilon = 2$ and $min_t = 2$. We can notice that $\sigma(\Upsilon) = 3 > \varepsilon$ and $|\Upsilon| = 2 \geq min_t$.

In essence, a closed swarm is a swarm which satisfies the *object-closed* and *time-closed* conditions therefore closed-swarm property is as follows:

Property 3.2. *Closed Swarm.* Given a frequent itemset $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$.

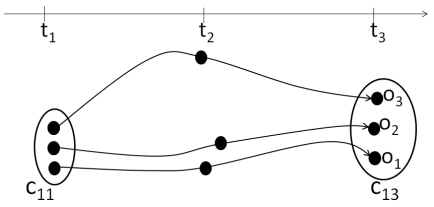



Fig. 5. A swarm from our example.

$(O(\Upsilon), T_\Upsilon)$ is a closed swarm if and only if:

$$\begin{cases} (1) : (O(\Upsilon), T_\Upsilon) \textit{ is a swarm} \\ (2) : \nexists \Upsilon' \textit{ s.t } O(\Upsilon) \subset O(\Upsilon'), T_{\Upsilon'} = T_\Upsilon \textit{ and } (O(\Upsilon'), T_\Upsilon) \textit{ is a swarm} \\ (3) : \nexists \Upsilon' \textit{ s.t. } O(\Upsilon') = O(\Upsilon), T_\Upsilon \subset T_{\Upsilon'} \textit{ and } (O(\Upsilon), T_{\Upsilon'}) \textit{ is a swarm} \end{cases} \tag{3.6}$$

**Proof.** After construction, we obtain $(O(\Upsilon), T_\Upsilon)$ which is a swarm. In addition, if $\nexists \Upsilon'$ s.t $O(\Upsilon) \subset O(\Upsilon'), T_{\Upsilon'} = T_\Upsilon$ and $(O(\Upsilon'), T_\Upsilon)$ is a swarm then $(O(\Upsilon), T_\Upsilon)$ cannot be enlarged in terms of objects. Therefore, it satisfies the *object-closed* condition. Furthermore, if $\nexists \Upsilon'$ s.t. $O(\Upsilon') = O(\Upsilon), T_\Upsilon \subset T_{\Upsilon'}$ and $(O(\Upsilon), T_{\Upsilon'})$ is a swarm then $(O(\Upsilon), T_\Upsilon)$ cannot be enlarged in terms of lifetime. Therefore, it satisfies the *time-closed* condition. Consequently, $(O(\Upsilon), T_\Upsilon)$ is a swarm and it satisfies *object-closed* and *time-closed* conditions and therefore $(O(\Upsilon), T_\Upsilon)$ is a closed swarm according to the *Definition 3.2*. □

According to the *Definition 2.3*, a convoy is a swarm which satisfies the lifetime consecutiveness condition. Therefore, for an itemset, we can extract a convoy if the following property holds:

Property 3.3. *Convoy.* Given a frequent itemset $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$. $(O(\Upsilon), T_\Upsilon)$ is a convoy if and only if:

$$\begin{cases} (1) : (O(\Upsilon), T_\Upsilon) \textit{ is a swarm} \\ (2) : \forall j, 1 \leq j < p : t_{a_{j+1}} = t_{a_j} + 1 \end{cases} \tag{3.7}$$

**Proof.** After construction, we obtain $(O(\Upsilon), T_\Upsilon)$ which is a swarm. In addition, if $\Upsilon$ satisfies the condition (2) then the $\Upsilon$'s lifetime is consecutive. Consequently, $(O(\Upsilon), T_\Upsilon)$ is a convoy according to the *Definition 2.3*. □

For instance, see Table 2 and Figure 6, for the frequent itemset $\Upsilon = \{c_{11}, c_{12}, c_{13}\}$ we have $(O(\Upsilon) = \{o_1, o_2\}, T_\Upsilon = \{t_1, t_2, t_3\})$ is a convoy with support threshold $\varepsilon = 2$ and $min_t = 2$. Note that $o_3$ is not in the convoy.

Please remind that a group pattern is a set of disjointed convoys which share the same objects, but in different time intervals. Therefore, the group pattern property is as follows:

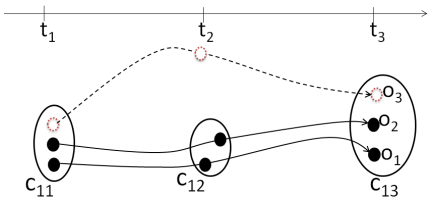



Fig. 6. A convoy from our example.

Property 3.4. *Group Pattern.* Given a frequent itemset $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$, a mininum weight $min_{wei}$, a minimum number of convoys $min_c$, and a set of consecutive time segments $T_{\mathcal{S}} = \{s_1, s_2, \ldots, s_n\}$. $(O(\Upsilon), T_{\mathcal{S}})$ is a group pattern if and only if:

$$\begin{cases} (1) : |T_{\mathcal{S}}| \geq min_c \\ (2) : \forall s_i, s_i \subseteq T_\Upsilon, |s_i| \geq min_t \\ (3) : \bigcap_{i=1}^{n} s_i = \emptyset, \bigcap_{i=1}^{n} O(s_i) = O(\Upsilon) \\ (4) : \forall s \notin T_{\mathcal{S}}, s \textit{ is a convoy, } O(\Upsilon) \not\subseteq O(s) \\ (5) : \frac{\sum_{i=1}^{n} |s_i|}{|T|} \geq min_{wei} \end{cases} \tag{3.8}$$

**Proof.** If $|T_{\mathcal{S}}| \geq min_c$ then we know that there are at least $min_c$ consecutive time intervals $s_i$ in $T_{\mathcal{S}}$. Furthermore, if $\forall s_i, s_i \subseteq T_\Upsilon$ then we have $O(\Upsilon) \subseteq O(s_i)$. In addition, if $|s_i| \geq min_t$ then $(O(\Upsilon), s_i)$ is a convoy (*Definition 2.3*). Now, $T_{\mathcal{S}}$ actually is a set of convoys of $O(\Upsilon)$ and if $\bigcap_{i=1}^{n} s_i = \emptyset$ then $T_{\mathcal{S}}$ is a set of disjointed convoys. A little bit further, if $\forall s \notin T_{\mathcal{S}}, s$ is a convoy and $O(\Upsilon) \not\subseteq O(s)$ then $\nexists T_{\mathcal{S}'}$ s.t. $T_{\mathcal{S}} \subset T_{\mathcal{S}'}$ and $\bigcap_{i=1}^{|T_{\mathcal{S}'}|} O(s_i) = O(\Upsilon)$. Therefore, $(O(\Upsilon), T_{\mathcal{S}})$ cannot be enlarged in terms of *number of convoys*. Similarly, if $\bigcap_{i=1}^{n} O(s_i) = O(\Upsilon)$ then $(O(\Upsilon), T_{\mathcal{S}})$ cannot be enlarged in terms of *objects*. Consequently, $(O(\Upsilon), T_{\mathcal{S}})$ is a closed swarm of disjointed convoys because $|O(\Upsilon)| \geq \varepsilon, |T_{\mathcal{S}}| \geq min_c$ and $(O(\Upsilon), T_{\mathcal{S}})$ cannot be enlarged (*Definition 3.2*). Finally, if $(O(\Upsilon), T_{\mathcal{S}})$ satisfies condition (5) then it is a valid group pattern due to *Definition 2.4*. □

As mentioned before, the main difference in periodic pattern mining is the input data while the definition is similar to that of closed swarm. The cluster matrix which is used for periodic mining can be defined as follows:

**Definition 3.2.** *Periodic Cluster Matrix (PCM).* Periodic cluster matrix is a cluster matrix with some differences as follows: 1) Object $o$ is a sub-trajectory which is denoted as $st$, and 2) $ST_{DB}$ is a set of all sub-trajectories in dataset.

For instance, see Table 3, a trajectory of objects is decomposed into 3 sub-trajectories and from them a periodic cluster matrix can be generated by applying clustering techniques. Assume that we can extract a frequent itemset $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$ from periodic cluster matrix, periodic patterns can be defined as follows:

Table 3. Periodic Cluster Matrix.

| $T_{DB}$ | | $t_1$ | $t_2$ | | $t_3$ |
|---|---|---|---|---|---|
| Clusters $C_{DB}$ | | $c_{11}$ | $c_{12}$ | $c_{22}$ | $c_{13}$ |
| $ST_{DB}$ | $st_1$ | 1 | 1 | | 1 |
| | $st_2$ | 1 | 1 | | 1 |
| | $st_3$ | 1 | | 1 | 1 |

Property 3.5. *Periodic Pattern.* Given a mininum weight $min_{wei}$, a frequent itemset $\Upsilon = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$ which is extracted from periodic cluster matrix. $(ST(\Upsilon), (T)_\Upsilon)$ is a periodic pattern if and only if $(ST(\Upsilon), (T)_\Upsilon)$ is a closed swarm. Note that $ST(\Upsilon) = \bigcap_{i=1}^{p} c_{t_{a_i}}$.

Above, we presented some useful properties to extract object movement patterns from itemsets. Now we will focus on the fact that from an itemset mining algorithm we are able to extract the set of all movement patterns. We thus start the proof process by analyzing the swarm extracting issue. This first lemma shows that from a set of frequent itemsets we are able to extract all the swarms embedded in the database.

**Lemma 3.1.** *Let $FI = \{\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_l\}$ be the frequent itemsets being mined from the cluster matrix with $minsup = \varepsilon$. All swarms $(O, T)$ can be extracted from $FI$.*

**Proof.** Let us assume that $(O, T)$ is a swarm, $T = \{t_{a_1}, t_{a_2}, \ldots, t_{a_m}\}$. According to the *Definition 2.1* we know that $|O| \geq \varepsilon$. If $(O, T)$ is a swarm then $\forall t_{a_i} \in T, \exists c_{t_{a_i}}$ s.t. $O \subseteq c_{t_{a_i}}$ therefore $\bigcap_{i=1}^{m} c_{t_{a_i}} = O$. In addition, we know that $\forall c_{t_{a_i}}, c_{t_{a_i}}$ is an item so $\exists \Upsilon = \bigcup_{i=1}^{m} c_{t_{a_i}}$ is an itemset and $O(\Upsilon) = \bigcap_{i=1}^{m} c_{t_{a_i}} = O, T_\Upsilon = \bigcup_{i=1}^{m} t_{a_i} = T$. Therefore, $(O(\Upsilon), T_\Upsilon)$ is a swarm. So, $(O, T)$ is extracted from $\Upsilon$. Furthermore, $\sigma(\Upsilon) = |O(\Upsilon)| = |O| \geq \varepsilon$ then $\Upsilon$ is a frequent itemset and $\Upsilon \in FI$. Finally, $\forall (O, T)$ s.t. if $(O, T)$ is a swarm then $\exists \Upsilon$ s.t. $\Upsilon \in FI$ and $(O, T)$ can be extracted from $\Upsilon$, we can conclude for all swarms $(O, T)$, they can be mined from $FI$. □

We can consider that by adding constraints such as "consecutive lifetime", "time-closed", "object-closed", and "integrity proportion" to swarms, we can retrieve convoys, closed swarms, and moving clusters. Therefore, if $Swarm, CSwarm, Convoy, MCluster$ respectively contain all swarms, closed-swarms, convoys, and moving clusters then we have: $CSwarm \subseteq Swarm$, $Convoy \subseteq Swarm$, and $MCluster \subseteq Swarm$. By applying *Lemma 3.1*, we retrieve all swarms from frequent itemsets. Since, a set of closed swarms, a set of convoys, and a set of moving clusters are subsets of swarms. Therefore they can be completely extracted from frequent itemsets. In addition, all periodic patterns also can be extracted because they are similar to closed swarms. Now, we will consider group patterns and we show that they can be extracted from the set of all frequent itemsets.

**Lemma 3.2.** *Given $FI = \{\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_l\}$ which contains all frequent itemsets*

*mined from cluster matrix with minsup = $\varepsilon$. All group patterns $(O, T_{\mathcal{S}})$ can be extracted from $FI$.*

**Proof.** $\forall(O, T_{\mathcal{S}})$ is a valid group pattern, we have $\exists T_{\mathcal{S}} = \{s_1, s_2, \ldots, s_n\}$ and $T_{\mathcal{S}}$ is a set of disjointed convoys of $O$. Therefore, $(O, T_{s_i})$ is a convoy and $\forall s_i \in T_{\mathcal{S}}, \forall t \in T_{s_i}, \exists c_t$ s.t. $O \subseteq c_t$. Let us assume $C_{s_i}$ is a set of clusters corresponding to $s_i$, we know that $\exists \Upsilon$, $\Upsilon$ is an itemset, $\Upsilon = \bigcup_{i=1}^{n} C_{s_i}$ and $O(\Upsilon) = \bigcap_{i=1}^{n} O(C_{s_i}) = O$. In addition, $(O, T_{\mathcal{S}})$ is a valid group pattern; therefore, $|O| \geq \varepsilon$ so $|O(\Upsilon)| \geq \varepsilon$. So, $\Upsilon$ is a frequent itemset and $\Upsilon \in FI$ because $\Upsilon$ is an itemset and $\sigma(\Upsilon) = |O(\Upsilon)| \geq \varepsilon$. Consequently, $\forall(O, T_{\mathcal{S}}), \exists \Upsilon \in FI$ s.t. $(O, T_{\mathcal{S}})$ can be extracted from $\Upsilon$ and therefore all group patterns can be extracted from $FI$. □

As we have shown that patterns such as swarms, closed swarms, convoys, group patterns can be similarly mapped into frequent itemset context. However, mining all frequent itemsets is cost prohibitive in some cases. Fortunately, the set of frequent closed itemsets has been proved to be a condensed collection of frequent itemsets, i.e., both a concise and lossless representation of a collection of frequent itemsets (Ref. 8, 9, 10, 24, 26). They are concise since a collection of frequent closed itemsets is orders of magnitude smaller than the corresponding collection of frequents. This allows the use of very low minimum support thresholds. Moreover, they are lossless, because it is possible to derive the support of every frequent itemsets in the collection from them. Therefore, we only need to extract frequent closed itemsets (FCIs) and then to scan them with properties to obtain the corresponding object movement patterns instead of having to mine all frequent itemsets (FIs).

## 4. Frequent Closed Itemset-based Object Movement Pattern Mining Algorithm

In previous, object movement patterns have been redefined in the itemset context. In this section, we propose two approaches i.e., *GeT_Move* and *Incremental GeT_Move*, to extract the patterns. The global process is illustrated in Figure 7.

In the first step, a clustering approach (Figure 7-(1)) is applied at each timestamp to group objects into different clusters. For each timestamp $t_a$, we have a set of clusters $C_a = \{c_{1t_a}, c_{2t_a}, \ldots, c_{mt_a}\}$, with $1 \leq k \leq m, c_{kt_a} \subseteq O_{DB}$. Moving object data can thus be converted to a cluster matrix $CM$ (Table 2).

### 4.1. *GeT_Move*

After generating the cluster matrix $CM$, a FCI mining algorithm is applied on $CM$ to extract all the FCIs. By scanning FCIs and checking properties, we can obtain the patterns.

In this paper, LCM algorithm[26] is applied to extract FCIs as it is known to be a very efficient algorithm. The key feature of the LCM algorithm is that after discovering a FCI $X$, it generates a new generator $X[i]$ by extending $X$ with a

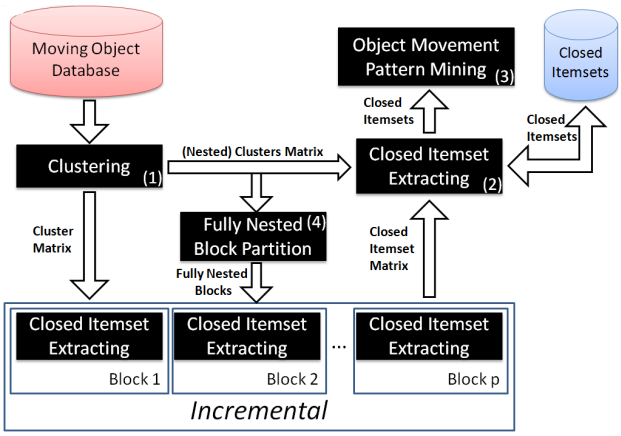



Fig. 7. The main process.

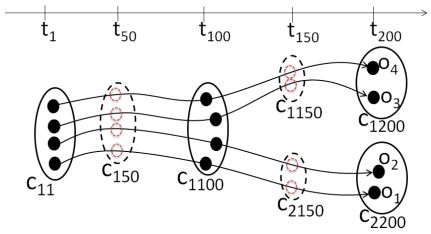
frequent item $i, i \notin X$. Using a total order relation on frequent items, LCM verifies if $X[i]$ violates this order by performing tests using only the *tidset*[b] of $X$, called $\mathcal{T}(X)$, and those of the frequent items $i$. If $X[i]$ is not discarded, then $X[i]$ is an order preserving generator of a new FCI. Then, its closure is computed using the previously mentioned tidsets.

In this process, we discard some useless itemset candidates. In object movement patterns, items (resp. clusters) must belong to different timestamps and therefore items (resp. clusters) which form a FCI must be in different timestamps. In contrast, to extract potential movement patterns by combining a set of items, these items cannot be in the same timestamp. Consequently, FCIs which include more than 1 item in the same timestamp will be discarded.

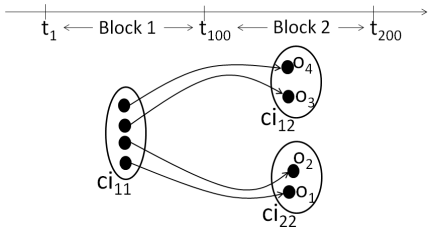
Thanks to the above characteristic, we now have the maximum length of the FCIs which is the number of timestamps $|T_{DB}|$. In addition, the LCM search space only depends on the number of objects (transactions) $|O_{DB}|$ and the maximum length of itemsets $|T_{DB}|$. Consequently, by using LCM and by applying the property, *GeT_Move* is not affected by the number of clusters and therefore the computation efficiency can be improved.

The pseudo code of *GeT_Move* is described in *Algorithm 1*. The core of *GeT_Move* algorithm is based on the LCM algorithm which has been slightly modified by adding the pruning rule and by extracting patterns from FCIs. The initial value of FCI $X$ is empty and then we start by putting item $i$ into $X$ (lines 2-3). By adding $i$ into $X$, we have $X[i]$ and if $X[i]$ is a FCI then $X[i]$ is used as a generator of a new FCI, call LCM_Iter$(X, \mathcal{T}(X), i(X))$ (lines 4-5). In LCM_Iter, we first check properties presented in Section 3 (line 8) for FCI $X$. Next, for each transaction $t \in \mathcal{T}(X)$, we add all items $j$, which are larger than $i(X)$ and satisfy the pruning

[b]Called *tidlists* in Ref. 24 and *denotations* in Ref. 26.




(a) The entire dataset.




(b) Data after applying frequent closed itemsets mining on Blocks.

Fig. 8. A case study example. (b)-$ci_{11}$ ($ci_{12}, ci_{22}$) is a FCI extracted from block 1 (block 2).

rule, into the occurrence set $\mathcal{J}[j]$ (lines 9-11). Next, for each $j \in \mathcal{J}[j]$, we check to see if $\mathcal{J}[j]$ is a FCI, and if so, then we recall LCM_Iter with the new generator (lines 12-14). After terminating the call for $\mathcal{J}[j]$, the memory for $\mathcal{J}[j]$ is released for the future use in $\mathcal{J}[k]$ for $k < j$ (lines 15).

Regarding to the PatternMining sub-function (lines 16-37), the algorithm basically checks properties of the itemset $X$ to extract patterns. If $X$ satisfies the $min_t$ condition then $X$ is a closed swarm (lines 18-19). After that, we check the consecutive time constraint for convoy and moving cluster (lines 21-22) and then if the convoy satisfies $min_t$ condition and correctness in terms of object containing (line 31), outputs convoy (line 32). Next, we put the convoy into a group pattern $gPattern$ (line 33) and then output $gPattern$ if it satisfies the $min_c$ condition and $min_{wei}$ condition at the end of scanning $X$ (line 37). Regarding to the moving cluster $mc$, we check the integrity at each pair of consecutive timestamps (line 24). If $mc$ satisfies the condition then the previous item $x_k$ will be merged into $mc$ (line 25). If not, we check the $min_t$ condition for $mc \cup x_k$ and if it is satisfied then we output $mc \cup x_k$ as a moving cluster.

### 4.2. *Incremental GeT_Move*

Usually, the transaction length can be large. FCI mining approaches can be slowed down when working with long transactions. Thus, if we apply the *GeT_Move* on the whole dataset, the extraction of the itemsets can be very time consuming.

To address this issue, we propose an *Incremental GeT_Move* algorithm. The basic idea is to shorten the transactions by splitting the trajectories (resp. cluster

**Algorithm 1: GeT_Move**

**Input** : Occurrence sets $\mathcal{J}$, int $\varepsilon$, int $min_t$, set of items $C_{DB}$, double $\theta$, int $min_c$, double $min_{wei}$

**begin**
  $X := I(\mathcal{T}(\emptyset))$; //The root
  **for** $i := 1$ *to* $|C_{DB}|$ **do**
    **if** $|\mathcal{T}(X[i])| \geq \varepsilon$ *and* $|X[i]|$ *is closed* **then**
      LCM_Iter($X[i], \mathcal{T}(X[i]), i$);
**LCM_Iter(**$X, \mathcal{T}(X), i(X)$**)**
**begin**
  **PatternMining**($X, min_t$); /*$X$ is a pattern?*/
  **foreach** *transaction* $t \in \mathcal{T}(X)$ **do**
    **foreach** $j \in t, j > i(X), j.time \notin time(X)$ **do**
      insert $j$ to $\mathcal{J}[j]$;
  **foreach** $j \in \mathcal{J}[j]$ *in the decreasing order* **do**
    **if** $|\mathcal{T}(\mathcal{J}[j])| \geq \varepsilon$ *and* $\mathcal{J}[j]$ *is closed* **then**
      LCM_Iter($\mathcal{J}[j], \mathcal{T}(\mathcal{J}[j]), j$);
    **Delete** $\mathcal{J}[j]$;
**PatternMining(**$X, min_t$**)**
**begin**
  **if** $|X| \geq min_t$ **then**
    **output** $X$; /*Closed Swarm*/
    $gPattern := \emptyset; convoy := \emptyset; mc := \emptyset$;
    **for** $k := 1$ *to* $|X - 1|$ **do**
      **if** $x_k.time = x_{(k+1)}.time - 1$ **then**
        $convoy := convoy \cup x_k$;
        **if** $\frac{|\mathcal{T}(x_k) \cap \mathcal{T}(x_{k+1})|}{|\mathcal{T}(x_k) \cup \mathcal{T}(x_{k+1})|} \geq \theta$ **then**
          $mc := mc \cup x_k$;
        **else**
          **if** $|mc \cup x_k| \geq min_t$ **then**
            **output** $mc \cup x_k$; /*MovingCluster*/
          $mc := \emptyset$;
      **else**
        **if** $|convoy \cup x_k| \geq min_t$ *and* $|\mathcal{T}(convoy \cup x_k)| = |\mathcal{T}(X)|$ **then**
          **output** $convoy \cup x_k$; /*Convoy*/
          $gPattern := gPattern \cup (convoy \cup x_k)$;
        **if** $|mc \cup x_k| \geq min_t$ **then**
          **output** $mc \cup x_k$; /*MovingCluster*/
        $convoy := \emptyset; mc := \emptyset$;
    **if** $|gPattern| \geq min_c$ *and* $\frac{size(gPattern)}{|T_{DB}|} \geq min_{wei}$ **then**
      **output** $gPattern$; /*Group Pattern*/

Where: $X$ is itemset, $X[i] := X \cup i, i(X)$ is the last item of $X, \mathcal{T}(X)$ is list of tractions that $X$ belongs to, $\mathcal{J}[j] := \mathcal{T}(X[j])$, $j.time$ is time index of item $j$, $time(X)$ is a set of time indexes of $X$, $|\mathcal{T}(convoy)|$ is the number of transactions that the $convoy$ belongs to, $|gPattern|$ and $size(gPattern)$ respectively are the number of convoys and the total length of the convoys in $gPattern$.

Table 4. Closed Itemset Matrix.

| Block $B$ | | $b_1$ | $b_2$ | |
|---|---|---|---|---|
| Frequent Closed Itemsets $CI$ | | $ci_{11}$ | $ci_{12}$ | $ci_{22}$ |
| $O_{DB}$ | $o_1$ | 1 | | 1 |
| | $o_2$ | 1 | | 1 |
| | $o_3$ | 1 | 1 | |
| | $o_4$ | 1 | 1 | |

matrix CM) into short intervals, called blocks. By applying FCI mining on each block, the data can then be compressed into local FCIs. Thus the length of itemsets and the number of items can be greatly reduced.

For instance, in Figure 8, if we consider $[t_1, t_{100}]$ as a block and $[t_{101}, t_{200}]$ as

another block, the maximum length of itemsets in both blocks is 100 (instead of 200). In addition, the original data can be greatly compressed (i.e., Figure 8b) and only 3 items remain: $ci_{11}, ci_{12}, ci_{22}$. Consequently, the process is much improved.

**Definition 4.1.** *Block.* Given a set of timestamps $T_{DB} = \{t_1, t_2, \ldots, t_n\}$, a cluster matrix $CM$. $CM$ is vertically split into equivalent (in terms of intervals) smaller cluster matrices and each of them is a block $b$. Assume $T_b$ is a set of timestamps of block $b$, $T_b = \{t_1, t_2, \ldots, t_k\}$, thus we have $|T_b| = k \leq |T_{DB}|$.

Assume that we obtain a set of blocks $B = \{b_1, b_2, \ldots, b_p\}$ with $|T_{b_1}| = |T_{b_2}| = \ldots = |T_{b_p}|, \bigcup_{i=1}^{p} b_i = CM$ and $\bigcap_{i=1}^{p} b_i = \emptyset$. Given a set of FCI collections $CI = \{CI_1, CI_2, \ldots, CI_p\}$ where $CI_i$ is mined from block $b_i$. $CI$ is presented as a *closed itemset matrix* which is formed by horizontally connecting all local FCIs: $CIM = \bigcup_{i=1}^{p} CI_i$.

**Definition 4.2.** *Closed Itemset Matrix (CIM).* Closed itemset matrix is a cluster matrix with some differences as follows: 1) Timestamp $t$ now becomes a block $b$, and 2) Item $c$ is a frequent closed itemset $ci$.

For instance, see Table 4, we have two sets of FCIs $CI_1 = \{ci_{11}\}, CI_2 = \{ci_{12}, ci_{22}\}$ which are respectively extracted from blocks $b_1, b_2$. Closed itemset matrix $CIM = CI_1 \cup CI_2$ means that $CIM$ is created by horizontally connecting $CI_1$ and $CI_2$. Consequently, we have $CIM$ as in Table 4.

We have already provided *blocks* to compress original data. Now, by applying FCI mining on the closed itemset matrix $CIM$, we are able to retrieve all FCIs from corresponding data. Note that items (in $CIM$) which are in the same block cannot be in the same frequent closed itemset.

**Lemma 4.1.** *Given a cluster matrix $CM$ which is vertically split into a set of blocks $B = \{b_1, b_2, \ldots, b_p\}$ s.t. $\forall \Upsilon, \Upsilon$ is a frequent closed itemset and $\Upsilon$ is extracted from $CM$ then $\Upsilon$ can be extracted from closed itemset matrix $CIM$.*

**Proof.** Let us assume that $\forall b_i, \exists I_i$ is a set of items belonging to $b_i$ and therefore we have $\bigcap_{i=1}^{|B|} I_i = \emptyset$. If $\forall \Upsilon, \Upsilon$ is a FCI extracted from $CM$ then $\Upsilon$ is formed as $\Upsilon = \{\gamma_1, \gamma_2, \ldots, \gamma_p\}$ where $\gamma_i$ is a set of items s.t. $\gamma_i \subseteq I_i$. In addition, $\Upsilon$ is a FCI and $O(\Upsilon) = \bigcap_{i=1}^{p} O(\gamma_i)$ then $\forall O(\gamma_i), O(\Upsilon) \subseteq O(\gamma_i)$. Furthermore, we have $|O(\Upsilon)| \geq \varepsilon$; therefore, $|O(\gamma_i)| \geq \varepsilon$ so $\gamma_i$ is a frequent itemset. Assume that $\exists \gamma_i, \gamma_i \notin CI_i$ then $\exists \Psi, \Psi \in CI_i$ s.t. $\gamma_i \subseteq \Psi$ and $\sigma(\gamma_i) = \sigma(\Psi), O(\gamma_i) = O(\Psi)$. Note that $\Psi$, $\gamma_i$ are from $b_i$. Remember that $O(\Upsilon) = O(\gamma_1) \cap O(\gamma_2) \cap \ldots \cap O(\gamma_i) \cap \ldots \cap O(\gamma_p)$ then we have: $\exists \Upsilon'$ s.t. $O(\Upsilon') = O(\gamma_1) \cap O(\gamma_2) \cap \ldots \cap O(\Psi) \cap \ldots \cap O(\gamma_p)$. Therefore, $O(\Upsilon') = O(\Upsilon)$ and $\sigma(\Upsilon') = \sigma(\Upsilon)$. In addition, we have $\gamma_i \subseteq \Psi$ so $\Upsilon \subseteq \Upsilon'$. Consequently, we obtain $\Upsilon \subseteq \Upsilon'$ and $\sigma(\Upsilon) = \sigma(\Upsilon')$. Therefore, $\Upsilon$ is not a FCI. That violates the assumption and thus we have: if $\exists \gamma_i, \gamma_i \notin CI_i$. So, $\Upsilon$ is not a FCI. Finally, we can conclude that $\forall \Upsilon, \Upsilon = \{\gamma_1, \gamma_2, \ldots, \gamma_p\}$ is a FCI extracted from $CM$, $\forall \gamma_i \in \Upsilon$, $\gamma_i$ must belong to

**Algorithm 2: Incremental GeT_Move**

**Input** : Occurrence sets $K$, int $\varepsilon$, int $min_t$, double $\theta$, set of Occurrence sets (blocks) $B$, int $min_c$, double $min_{wei}$

```
1  begin
2  |  K := ∅; CI := φ; int item_total := 0;
3  |  foreach b ∈ B do
4  |  |  LCM(b, ε, I_b);
5  |  GeT_Move(K, ε, min_t, CI, θ, min_c, min_wei);
6  LCM(Occurrence sets J, int σ_0, set of items C)
7  begin
8  |  X := I(T(∅)); //The root
9  |  for i := 1 to |C| do
10 |  |  if |T(X[i])| ≥ ε and |X[i]| is closed then
11 |  |  |  LCM_Iter(X[i], T(X[i]), i);
12 LCM_Iter(X, T(X), i(X))
13 begin
14 |  Update(K, X, T(X), item_total++);
15 |  foreach transaction t ∈ T(X) do
16 |  |  foreach j ∈ t, j > i(X), j.time ∉ time(X) do
17 |  |  |  insert j to J[j];
18 |  foreach j, J[j] ≠ φ in the decreasing order do
19 |  |  if |T(J[j])| ≥ ε and J[j] is closed then
20 |  |  |  LCM_Iter(J[j], T(J[j]), j);
21 |  |  Delete J[j];
22 Update(K, X, T(X), item_total)
23 begin
24 |  foreach t ∈ T(X) do
25 |  |  insert item_total into K[t];
26 |  CI := CI ∪ item_total;
```

$CI_i$ and $\gamma_i$ is an item in closed itemset matrix $CIM$. Therefore, $\Upsilon$ can be retrieved by applying FCI mining on $CIM$. □

By applying *Lemma 4.1*, we can obtain all the FCIs and from the itemsets, patterns can be extracted. Note that the Incremental GeT_Move does not depend on the length restriction $min_t$. The reason is that $min_t$ is only used in the **Pattern-Mining** step. Whatever $min_t$ ($min_t \geq$ block size or $min_t \leq$ block size), Incremental GeT_Move can extract all the FCIs and therefore the final results are the same.

The pseudo code of *Incremental GeT_Move* is described in *Algorithm 2*. The main difference between the code of *Incremental GeT_Move* and *GeT_Move* is the *Update* sub-function. In this function, we generate the closed itemset matrix from blocks (line 14 and lines 22-26). Next, we apply the *GeT_Move* to extract patterns (line 5).

### 4.3. *Toward A Parameter Free Incremental GeT_Move Algorithm*

Until now, we have presented the Incremental GeT_Move which splits the original cluster matrix into different equivalent blocks. The experiment results (Section 5) show that the algorithm is efficient. However, the disadvantage of this approach is that we do not know what is the optimal block size. To identify them, different techniques can be applied, such as data sampling in which a sample of data is used to investigate the optimal block sizes. Even if this approach is appealing, extracting such a sample is very difficult.

To address this issue, we dynamically assign blocks to the Incremental GeT_Move. We propose the definition of a fully nested cluster matrix inspired by the segmented nestedness proposed in Ref. 31 (Figure 9c) as follows.

**Definition 4.3.** *Fully Nested Cluster Matrix (resp. Block).* An $n \times m$ 0-1 block $b$ is fully nested if for any two columns $r_i$ and $r_{i+1}$ ($r_i, r_{i+1} \in b$) we have $r_i \cap r_{i+1} = r_{i+1}$.

We can consider that the *LCM* is very efficient when it is applied on dense (resp. (fully) nested) datasets and blocks. Let $E$ be the universe of items, consisting of items $1, \ldots, n$. A subset $X$ of $E$ is called an itemset. In the LCM algorithm process on a common cluster matrix, for any $X$, we make the recursive call for $X[i]$ for each $i \in \{i(X)+1, \ldots, |E|\}$ because we do not know which $X[i]$ will be a closed itemset when $X$ is extended by adding $i$ to $X$. Meanwhile, for a fully nested cluster matrix, we know that only the recursive call for item $i = i(X)+1$ is valuable and the other recursive calls for each item $i \in \{i(X)+2, \ldots, |E|\}$ are useless. Note that $i(X)$ returns the last item of $X$.

Property 4.1. *Recursive Call.* Given a fully nested cluster matrix *nCM* (resp. block), a universe of items $E$ of *nCM*, an itemset $X$ which is a subset of $E$. All the FCIs can be generated by making a recursive call of item $i = i(X)+1$.

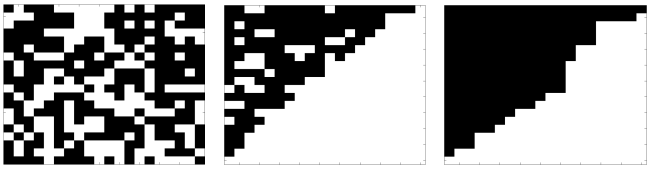
**Proof.** After construction, we have $\forall i \in E, O(i) \cap O(i+1) = O(i+1)$; thus, $O(i+1) \subseteq O(i)$. In addition, $\forall i' \in \{i(X)+2, \ldots, |E|\}$ we need to make a recursive call for $X[i']$ and let assume that we obtain a frequent itemset $X \cup i' \cup X'$ with $X' \subseteq \{i(X)+3, \ldots, |E|\}$. We can consider that $O(i') \subseteq O\big(i(X)+1\big)$ and therefore $O(X \cup i' \cup X') = O\Big(X \cup \big(i(X)+1\big) \cup i' \cup X'\Big)$. Consequently, $X \cup i' \cup X'$ is not a FCI because $(X \cup i' \cup X') \subset \Big(X \cup \big(i(X)+1\big) \cup i' \cup X'\Big)$ and $O(X \cup i' \cup X') = O\Big(X \cup \big(i(X)+1\big) \cup i' \cup X'\Big)$. Furthermore, $\Big(X \cup \big(i(X)+1\big) \cup i' \cup X'\Big)$ can be generated by making a recursive call for $i(X)+1$. We can conclude that it is useless to make a recursive call for $\forall i' \in \{i(X)+2, \ldots, |E|\}$ and additionally, all FCIs can be generated only by making a recursive call for $i(X)+1$. □

By applying *Property 4.1*, we can consider that LCM is more efficient on a fully nested matrix because it reduces unnecessary recursive calls. Therefore, our goal is

**Algorithm 3: Fully Nested Block Partition**

**Input** : a nested cluster matrix $CM_N$
**Output**: a set of blocks $B$

```
1  begin
2      B := ∅; NestedB := ∅; SpareB := ∅;
3      foreach item i ∈ CM_N do
4          if i ∩ (i + 1) = (i + 1) then
5              NestedB := NestedB ∪ i;
6          else
7              NestedB := NestedB ∪ i;
8              if |NestedB| ≤ 1 then
9                  SpareB.push_all(NestedB);
10                 NestedB := ∅
11             else
12                 B := B ∪ NestedB;
13                 NestedB := ∅
14     return B := B ∪ SpareB;
15 where the purpose SpareB.push_all(NestedB) function is to put all items in NestedB to
   SpareB.
```



(a) (b) (c)

Fig. 9. Examples of non-nested , almost nested, and fully nested datasets. Black = 1, white = 0. (a) Original, (b) Almost nested, (c) Fully nested.

to retrieve fully nested blocks to improve the performance of the Incremental GeT Move. In order to reach this goal, we first apply the *nested and segment nested Greedy* algorithm[c] (Ref. 31) to re-arrange the cluster matrix (Figure 9a) so that it now becomes a *nested cluster matrix* (Figure 9b). Then, we propose a sub-function *Nested Block Partition* (Figure 7-(4)) to dynamically split the nested cluster matrix into fully nested blocks (Figure 9c).

By following the Definition 4.3 and scanning the nested cluster matrix from the beginning to the end, we are able to obtain all fully nested blocks. We start from the first column of nested cluster matrix, then we check the next column and if the nested condition is held then the block is expanded; otherwise, the block is set and we create a new block. Note that all small blocks containing only 1 column are merged into a sparse block, denoted $SpareB$. At the end, we obtain a set of

[c]http://www.aics-research.com/nestedness/

Table 5. An example of FCI binary presentation.

| | | $FCIs_{DB}$ | | | $FCIs_{DB'}$ | |
|---|---|---|---|---|---|---|
| $binary(FCI)$ | | $b(ci_1)$ | $b(ci_2)$ | | $b(ci'_1)$ | $b(ci'_2)$ |
| $O_{DB}$ | $o_1$ | 1 | 0 | | 1 | 0 |
| | $o_2$ | 1 | 0 | | 1 | 1 |
| | $o_3$ | 0 | 1 | | 0 | 1 |
| | $o_4$ | 0 | 1 | | 0 | 1 |

fully nested blocks $NestedB$ and a $SpareB$. Finally, the Incremental GeT_Move is applied on $B = NestedB \cup SpareB$.

The pseudo code of the Fully Nested Block Partition sub-function is described in *Algorithm 3*.

### 4.4. *Object Movement Pattern Mining Algorithm Based on Explicit Combination of FCI Pairs*

In real world applications (e.g., cars), object locations are continuously reported by using Global Positioning System (GPS). Therefore, new data is always available. Let us denote the new movement data as $(O_{DB}, T_{DB'})$. In fact, it is cost-prohibitive and time consuming to execute Incremental GeT_Move (or GeT_Move) on the entire database (denoted $(O_{DB}, T_{DB} \cup T_{DB'})$) which is generated by merging $(O_{DB}, T_{DB'})$ into the existing database $(O_{DB}, T_{DB})$. To tackle this issue, we provide an approach which efficiently combines the existing frequent closed itemsets $FCIs_{DB}$ with the new frequent closed itemsets $FCIs_{DB'}$, which are extracted from $DB'$, to obtain the final result $FCIs_{DB \cup DB'}$.

For instance, in Table 5, we have two sets of frequent closed itemsets $FCIs_{DB}$ and $FCIs_{DB'}$. Each FCI will be presented as a $|O_{DB}|$-bit binary numeral. Let us define a set of operations that will be used for object movement pattern mining based on explicit combination of FCI pairs. Given two FCIs $ci$ and $ci'$, we have that:

- $ci \wedge ci'$: returns $b(ci) \wedge b(ci')$.
- $ci \vee ci'$: returns $b(ci) \vee b(ci')$.
- $ci \cup ci'$: returns a set of clusters that are the union of $ci_1$ and $ci'_1$.
- $Size(ci)$ returns the number of '1's in $ci$. Note that $Size(ci) = O(ci) = \sigma(ci)$.

The principle function of our algorithm is to explicitly combine all pairs of $FCIs(ci, ci')$ to generate new FCIs. Let us assume that $ci \wedge ci' = \gamma$, $\gamma = ci \cup ci'$ is a FCI if $\sigma(\gamma)$ is larger than $\varepsilon$ and that there are no subsets of $O(ci)$, $O(ci')$ so that they are supersets of $O(\gamma)$. Here is an explicit combination of a pair of $FCIs(ci, ci')$:

Property 4.2. *Explicit Combination of a pair of FCIs.* Given FCIs $ci$ and $ci'$ so that $ci \in FCIs_{DB}, ci' \in FCIs_{DB'}$, a . $ci \cup ci'$ is a FCI that belongs to $FCIs_{DB \cup DB'}$ if

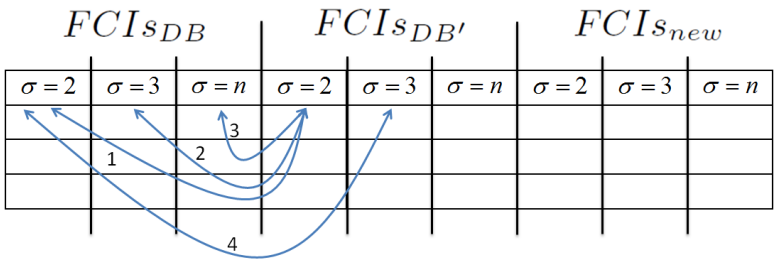



Fig. 10. An example of the explicit combination of pairs of FCIs-based approach.

and only if:

$$\begin{cases} if\ ci \wedge ci' = \gamma\ then \\ (1) : Size(\gamma) \geq \varepsilon \\ (2) : \nexists p : p \in FCIs_{DB}, O(\gamma) \subseteq O(p) \subseteq O(ci) \\ (3) : \nexists p' : p' \in FCIs_{DB'}, O(\gamma) \subseteq O(p') \subseteq O(ci') \end{cases} \tag{4.9}$$

where $ci = \{c_{t_{a_1}}, c_{t_{a_2}}, \ldots, c_{t_{a_p}}\}$ and $ci' = \{c'_{t_{a_1}}, c'_{t_{a_2}}, \ldots, c'_{t_{a_p}}\}$.

**Proof.** After construction, we have $\nexists p : p \in FCIs_{DB}, O(\gamma) \subseteq O(p) \subseteq O(ci)$. We assume that $\exists i$ s.t. $i \in C_{DB}, O(\gamma) \subseteq i$ and $i \notin ci$ therefore $\exists p$ s.t. $p = \{\forall i | i \in C_{DB}, O(\gamma) \subseteq i, i \notin ci\} \cup ci, O(\gamma) \subseteq O(p)$. Consequently, $\forall i \in C_{DB}, O(\gamma) \subseteq i$ then $i \in p$ and therefore $p$ is a FCI and $p \in FCIs_{DB}$. This violates the assumption and therefore $\nexists i$ s.t. $i \in C_{DB}, O(\gamma) \subseteq i$ and $i \notin ci$ or $\forall i$ s.t. $i \in C_{DB}, O(\gamma) \subseteq i$ then $i \in ci$. Similarly, if $\nexists p' : p' \in FCIs_{DB'}, O(\gamma) \subseteq O(p') \subseteq O(ci')$ then $\forall i'$ s.t. $i' \in C_{DB'}, O(\gamma) \subseteq i'$ then $i' \in ci'$. Consequently, if $\forall i \in C_{DB \cup DB'}, O(\gamma) \subseteq i$ then $i \in ci \cup ci'$. In addition, $Size(\gamma) = \sigma(\gamma) \geq \varepsilon$ and therefore $ci \cup ci'$ is a FCI and $ci \cup ci' \in FCIs_{DB \cup DB'}$. □

We can consider that if $ci \cup ci'$ is a FCI, they must respectively be the two longest FCIs which contain $O(\gamma)$ in $FCIs_{DB}$ and $FCIs_{DB'}$. $(O(\gamma), ci \cup ci')$ is a new FCI and it will be stored in a set of new frequent closed itemsets, named $FCIs_{new}$. To efficiently make all combinations, we first partition $FCIs_{DB}, FCIs_{DB'}$ and $FCIs_{new}$ into different partitions in terms of support so that the FCIs, that have the same support value, will be in the same partition (Figure 10). Secondly, partitions are combined from the smallest support values (resp. longest FCIs) to the largest ones (resp. shortest FCIs). New FCIs will be added into the right partition in $FCIs_{new}$. By using this approach, it is guaranteed that the first time there is $ci \wedge ci' = \gamma, Size(\gamma) \geq \varepsilon$ then $ci \cup ci'$ is a new FCI because they are the two longest FCIs which contain $O(\gamma)$. Therefore, we just ignore the later combinations which return $\gamma$ as the result. Furthermore, to ensure that already exists in $FCIs_{new}$ or not, we only need to check items in the $FCIs_{new}$ partition whose support value is equal to $Size(\gamma)$. We can consider that by partitioning $FCIs_{DB}, FCIs_{DB'}$ and $FCIs_{new}$, the process is much improved. Additionally, we also propose a pruning

rule to speed up the approach by ending the combination running of a FCI $ci'$ as follows:

**Lemma 4.2.** *The combination running of $ci'$ is ended if:*

$$\exists ci \in FCIs_{DB} \text{ s.t. } ci \wedge ci' = ci', ci \cup ci' \text{ is a FCI.} \tag{4.10}$$

**Proof.** Assume that $\exists \Upsilon : \Upsilon \in FCIs_{DB}, \sigma(\Upsilon) \geq \sigma(ci), \Upsilon \wedge ci' = ci'$. If $O(ci) \subseteq O(\Upsilon)$ then we have $ci \in FCIs_{DB}, O(ci') \subseteq O(ci) \subseteq O(\Upsilon)$ and this violates the condition 2 in *Property 4.2*, therefore $\Upsilon \cup ci'$ is not a FCI. If $O(ci) \nsubseteq O(\Upsilon)$ then $\exists i \in C_{DB}$ s.t. $O(ci') \subseteq i$ and $i \notin \Upsilon$. Furthermore, $\exists p : p = \{\forall i | i \in C_{DB}, O(ci') \subseteq i, i \notin \Upsilon\} \cup \Upsilon$. So, $\forall i, i \in C_{DB}, O(ci') \subseteq i$ then $i \in p$ and therefore $p$ is a FCI and $p \in FCIs_{DB}$. In addition, $O(ci') \subseteq O(p) \subseteq O(\Upsilon)$. This violates the condition 2 in *Property 4.2*, therefore $\Upsilon \cup ci'$ is not a FCI. Consequently, we can conclude that $\nexists \Upsilon$ s.t. $\Upsilon \in FCIs_{DB}, \sigma(\Upsilon) \geq \sigma(ci), \Upsilon \wedge ci' = ci'$ and $\Upsilon \cup ci'$ is a FCI. Therefore, we do not need to continue the combination running of $ci'$. □

Similar to *Lemma 4.2*, in the explicit combination process, $ci$ will be deactivated for further combinations when there is a $ci'$ so that $ci \wedge ci' = ci$ and $ci \cup ci'$ is a FCI. After generating all new FCIs in $FCIs_{new}$, the final result $FCIs_{DB \cup DB'}$ is created by collecting FCIs in $FCIs_{DB}, FCIs_{DB'}$, and $FCIs_{new}$. In this step, some of them will be discarded such that:

Property 4.3. *Discarded FCIs in $FCIs_{DB \cup DB'}$ Creating Step.* All the FCIs which satisfy the following conditions will not be selected as a FCIs in the final result.

$$\begin{cases} (1) : \forall ci \in FCIs_{DB}, \text{ if } \exists ci' \in FCIs_{DB'} \text{ s.t. } ci \wedge ci' = ci \\ (2) : \forall ci' \in FCIs_{DB'}, \text{ if } \exists ci \in FCIs_{DB} \text{ s.t. } ci \wedge ci' = ci' \end{cases} \tag{4.11}$$

Note that during the explicit combination step, the FCIs which will not be selected for the final results are removed. It means that we only add all suitable FCIs into $FCIs_{DB \cup DB'}$. Therefore it is optimized and much less costly. In the worst case scenario, the complexity of explicit combination of pairs of FCIs step is $O(|FCIs_{DB}| \times |FCIs_{DB'}| \times \frac{|FCIs_{new}|}{\#partitions(FCIs_{new})})$. In fact, $T_{DB'}$ is much smaller than $T_{DB}$ and therefore $FCIs_{DB'}, FCIs_{new}$ are very small compare to $FCIs_{DB}$. Consequently, the process could be significantly improved when compare to the executing of Incremental GeT Move on the entire database $(O_{DB}, T_{DB \cup DB'})$.

The pseudo code of the *Object Movement Pattern Mining Algorithm Based on Explicit Combination of FCI Pairs* is described in *Algorithm 4*.

## 5. Experimental Results

A comprehensive performance study has been conducted on real and synthetic datasets. All the algorithms are implemented in C++, and all the experiments

**Algorithm 4: Explicit Combination of Pairs of FCIs-based Object Movement Pattern Mining Algorithm**

**Input** : a set of FCIs $FCIs_{DB}$, Occurrence sets $K$, int $\varepsilon$, int $min_t$, double $\theta$, set of Occurrence sets (blocks) $B'$, int $min_c$, double $min_{wei}$

```
1  begin
2      FCIs_DB' := ∅; FCIs_new := ∅; FCIs_DB∪DB' := ∅;
3      FCIs_DB' := Incremental GeT_Move* (K, ε, min_t, CI, θ, B', min_c, min_wei);
4      foreach partition P' ∈ FCIs_DB' do
5          foreach FCI ci' ∈ P' do
6              foreach partition P ∈ FCIs_DB do
7                  foreach FCI ci ∈ P do
8                      γ := ci ∧ ci';
9                      if Size(γ) ≥ ε and FCIs_new.notContain(γ, Size(γ)) then
10                         γ := ci ∪ ci';
11                         FCIs_new.add(γ, Size(γ));
12                         if γ = ci then
13                             FCIs_DB.remove(ci);
14                         if γ = ci' then
15                             FCIs_DB.remove(ci');
16                         go to line 5;
17     FCIs_DB∪DB' := FCIs_DB ∪ FCIs_DB' ∪ FCIs_new;
18     foreach FCI X ∈ FCIs_DB∪DB' do
19         PatternMining(X, min_t); /*X is a pattern?*/
20 Where: Incremental GeT_Move* is an Incremental GeT_Move without PatternMining
   sub-function, FCIs_new.notContain(γ, Size(γ)) returns true if there does not exists γ in
   partition which has the support value is Size(γ).
```

are carried out on a 2.8GHz Intel Core i7 system with 4GB Memory. The system runs Ubuntu 11.10 and g++ version 4.6.1.

As in Ref. 6, the two following real datasets[d] have been used during experiments: *Swainsoni dataset* includes 43 objects evolving over 764 different timestamps. The dataset was obtained from July 1995 to June 1998. *Buffalo dataset* concerns 165 buffaloes and the tracking time from year 2000 to year 2006. The original data has 26610 reported locations and 3001 timestamps.

Similar to Ref. 6, 3, 21, we first use linear interpolation to fill in the missing data. For study purposes, we needed the objects to stay together for at least $min_t$ timestamps. As Ref. 6, 3, 21, DBScan[5] ($MinPts = 2, Eps = 0.001$) is applied to generate clusters at each timestamp.

### 5.1. *Effectiveness*

We proved that mining object movement patterns can be similarly mapped into itemset mining issue. Therefore, in theoretical aspect, our approaches can provide

[d]http://www.movebank.org

the correct results. A further comparison is performed to obtain the object movement patterns by employing traditional algorithms such as, $CMC, CuTS^{*}$[3e] (convoy mining), $ObjectGrowth$[6] (closed swarm mining), and our approaches. To apply our algorithms, we split the cluster matrix into blocks such as each block $b$ contains 25 consecutive timestamps. In addition, to retrieve all the patterns, in the reported experiments, the default value of $\varepsilon$ is set to 2 (two objects can form a pattern), $min_t$ is 1. Note that the default values are the hardest conditions for examining the algorithms. Then in the following we mainly focus on different values of $min_t$ in order to obtain different sets of convoys, closed swarms and group patterns. Note that for group patterns, $min_c$ is 1 and $min_{wei}$ is 0.

The results show that our proposed approaches obtain the same results compared to the traditional algorithms. An example of patterns is illustrated in Figure 11.

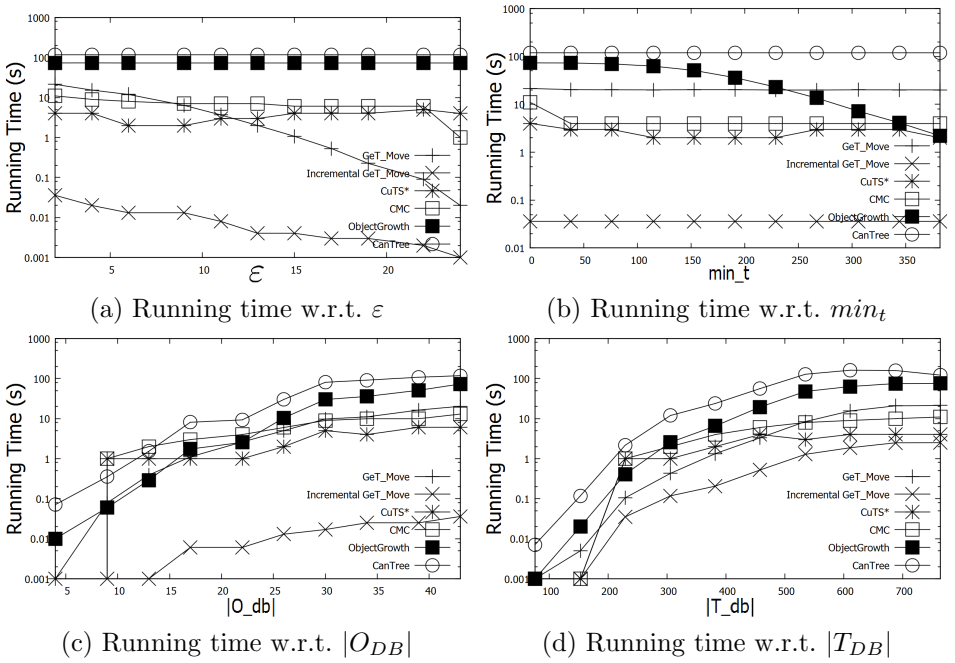
[e]The source code of $CMC, CuTS^{*}$ is available at http://lsirpeople.epfl.ch/jeung/source_codes.htm


(a) One of discovered closed swarms.


(b) One of discovered convoys.


(c) One of discovered group patterns.

Fig. 11. An example of patterns discovered from Swainsoni dataset.




(a) Running time w.r.t. $\varepsilon$ (b) Running time w.r.t. $min_t$

(c) Running time w.r.t. $|O_{DB}|$ (d) Running time w.r.t. $|T_{DB}|$

Fig. 12. Running time on Swainsoni dataset.

For instance, see Figure 11a, a closed swarm is discovered within a frequent closed itemset. The yellow, green, and red lines represent three Swainsonies. They are migrating together from North America to South America via a narrow corridor through Central America. Furthermore, from the itemset, a convoy and a group pattern are also extracted (i.e., Figures 11b, 11c).

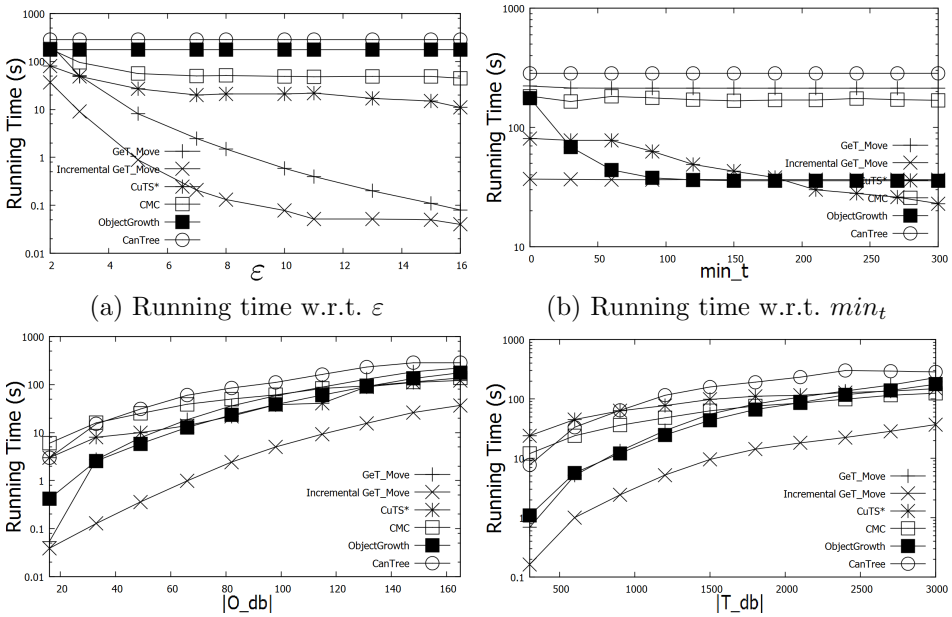
## 5.2. *Efficiency*

### 5.2.1. *Incremental GeT_Move and GeT_Move Efficiency*

To show the efficiency of our algorithms, we generate larger synthetic datasets using Brinkhoff's network-based generator[f] of moving objects as in Ref. 6. We generate 500 objects ($|O_{DB}| = 500$) for $10^4$ timestamps ($|T_{DB}| = 10^4$) using the generator's default map with a low moving speed (250). There are $5 \times 10^6$ points in total. The DBScan ($MinPts = 3, Eps = 300$) is applied to obtain clusters for each timestamp.

In the efficiency comparison, we employ $CMC, CuTS^*$ and *ObjectGrowth*. Note that, in Ref. 6, *ObjectGrowth* outperforms *VG-Growth*[21] (a group pattern mining algorithm) in terms of performance. Therefore we will only consider the *ObjectGrowth* instead of both. The GeT_Move and the Incremental GeT_Move extracted closed swarms, convoys, and group patterns. Meanwhile the $CMC$ and

[f]http://iapg.jade-hs.de/personen/brinkhoff/generator/



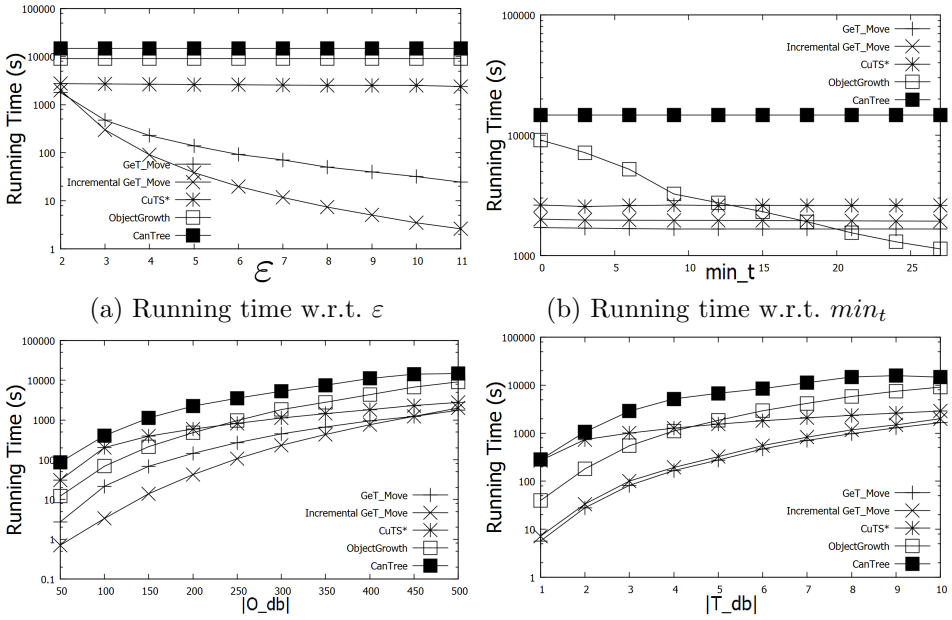

(a) Running time w.r.t. $\varepsilon$ (b) Running time w.r.t. $min_t$

(c) Running time w.r.t. $|O_{DB}|$ (d) Running time w.r.t. $|T_{DB}|$

Fig. 13. Running time on Buffalo dataset.

the *CuTS** only extracted convoys.The *ObjectGrowth* extracted closed swarms. In addition, we also employ a state-of-the-art incremental frequent itemset mining algorithm, *CanTree*[35], which is known as a very efficient method. In the CanTree, the cluster matrix is horizontally sliced into blocks each of them contains 10% number of objects in $|O_{DB}|$. By doing so the CanTree can be utilized in an incremental way.

**Efficiency w.r.t.** $\varepsilon, min_t$. Figures 12a, 13a, 14a show running time w.r.t. $\varepsilon$. It is clear that our approaches outperform other algorithms. The ObjectGrowth and the CanTree are the slowest ones and the main reason is that with low $min_t$ (default $min_t = 1$), the *Apriori Pruning* rule (the most efficient pruning rule) is no longer effective. Therefore, the search space is greatly enlarged ($2^{|O_{DB}|}$ in the worst case). In addition, there is no pruning rule for $\varepsilon$ and therefore the change of $\varepsilon$ does not directly affect the running time of the ObjectGrowth. Furthermore, the reasons why the CanTree is slower than the other algorithms are: 1) the CanTree has to build the tree from which closed frequent itemsets are extracted while, other algorithms do not need to build this tree; and 2) most of frequent itemset mining algorithms are designed for short transactions and the CanTree is one of them. Meanwhile, in moving object context, a transaction is very long (thousand of clusters). Furthermore, the GeT_Move is lower than the Incremental GeT_Move. The main reason is that the GeT_Move has to process with long transactions. Meanwhile, thanks to blocks, the number of items is greatly reduced and transactions are not long as the ones in the GeT_Move.




(a) Running time w.r.t. $\varepsilon$ (b) Running time w.r.t. $min_t$

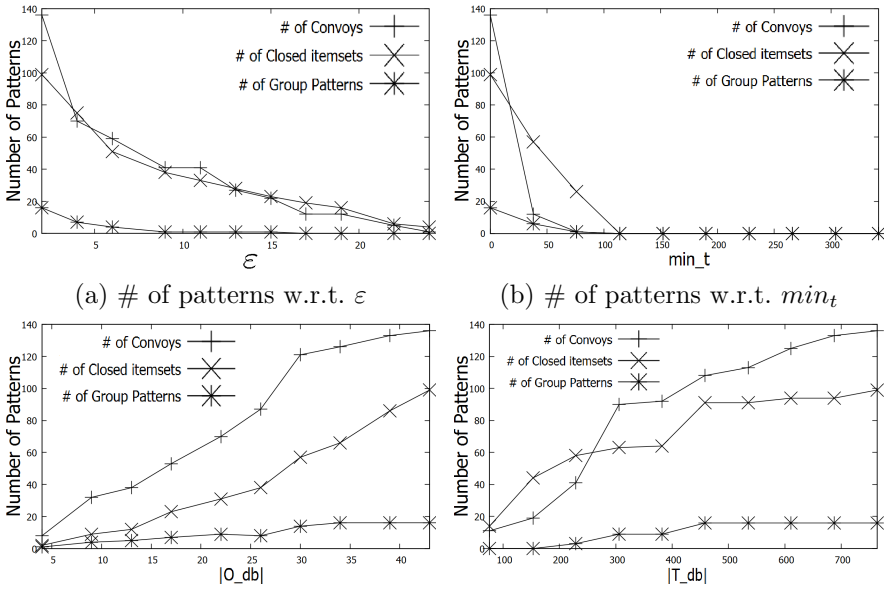
(c) Running time w.r.t. $|O_{DB}|$ (d) Running time w.r.t. $|T_{DB}|$

Fig. 14. Running time on Synthetic dataset.

Figures 12b, 13b, 14b show running time w.r.t. $min_t$. In almost all cases, our approaches outperform the other algorithms. With low $min_t$, The Incremental GeT_Move is much faster than the others. However, when mint is higher ($min_t > 200$ in Figure 13b, $min_t > 20$ in Figure 14b) our algorithms take more time than the CuTS* and the ObjectGrowth. This is because with high value of $min_t$, the number of patterns is significantly reduced (Figures 15b, 16b, 17b) (i.e., no extracted convoy when $min_t > 100$ (resp. $min_t > 200$, $min_t > 10$), Figure 15b (resp. Figures 16b, 17b)). Therefore the CuTS* and the ObjectGrowth are faster. Meanwhile, our proposed approaches have to work with FCIs.

**Efficiency w.r.t.** $|O_{DB}|, |T_{DB}|$. Figures 12c-d, Figures 13c-d, Figures 14c-d show the running time when varying $|O_{DB}|$ and $|T_{DB}|$ respectively. In all cases, the Incremental GeT_Move outperforms the other algorithms. However, with synthetic data and lowest values of $\varepsilon = 2$ and $min_t = 1$ (Figure 14d), the GeT_Move is a little bit faster than the Incremental GeT_Move. This is because the Incremental GeT_Move does not have any information to obtain the better partitions (blocks).

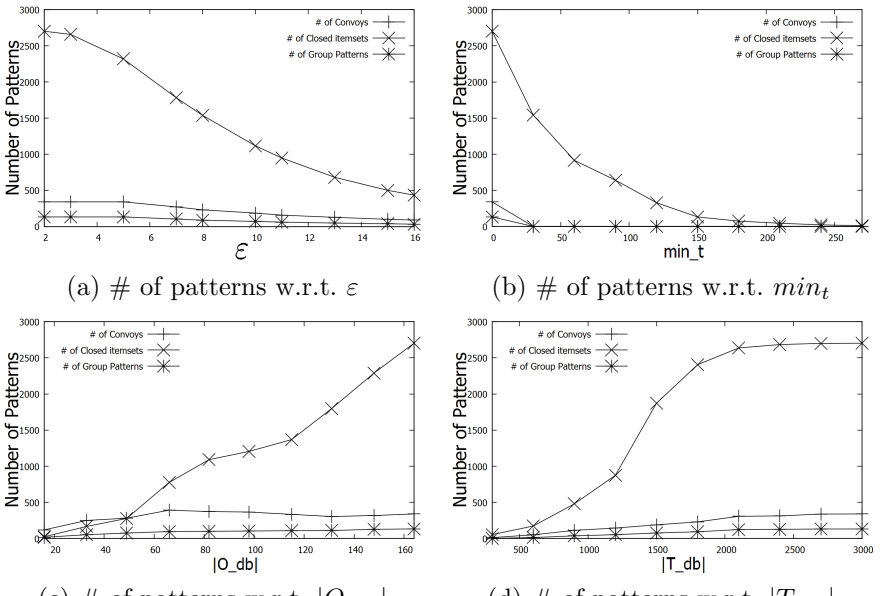
**Scalability w.r.t.** $\varepsilon$. We can consider that the running time of algorithms does not change significantly when varied $min_t, |O_{DB}|$, and $|T_{DB}|$ in synthetic data (Figure 16). However, they are quite different when varying $\varepsilon$ (default $min_t = 1$). Therefore, we generate another large synthetic data to test the scalability of algorithms on $\varepsilon$. The dataset includes 50,000 objects moving during 10,000 timestamps and it contains 500 million locations in total. The executions of the CMC and the




(a) # of patterns w.r.t. $\varepsilon$ (b) # of patterns w.r.t. $min_t$

(c) # of patterns w.r.t. $|O_{DB}|$ (d) # of patterns w.r.t. $|T_{DB}|$

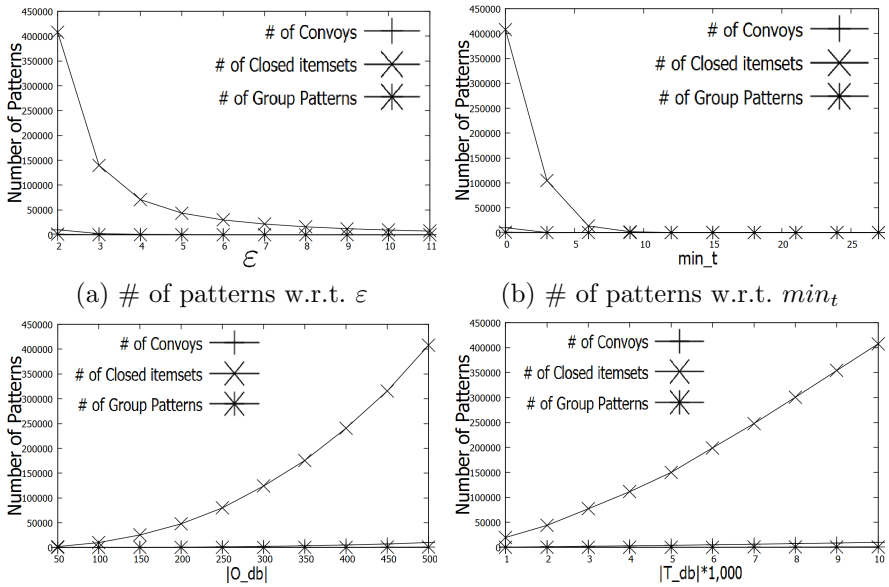
Fig. 15. #Patterns on Swainsoni dataset. # of closed itemsets is # of closed swarms.




(a) # of patterns w.r.t. $\varepsilon$ (b) # of patterns w.r.t. $min_t$

(c) # of patterns w.r.t. $|O_{DB}|$ (d) # of patterns w.r.t. $|T_{DB}|$

Fig. 16. #Patterns on Buffalo dataset. Note that # of closed itemsets is # of closed swarms.




(a) # of patterns w.r.t. $\varepsilon$ (b) # of patterns w.r.t. $min_t$

(c) # of patterns w.r.t. $|O_{DB}|$ (d) # of patterns w.r.t. $|T_{DB}|$

Fig. 17. #Patterns on Synthetic dataset. Note that # of closed itemsets is # of closed swarms.

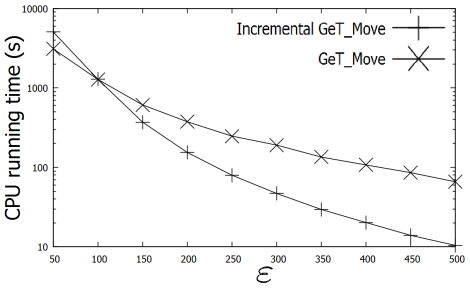



Fig. 18. Running time w.r.t $\varepsilon$ on large Synthetic dataset.

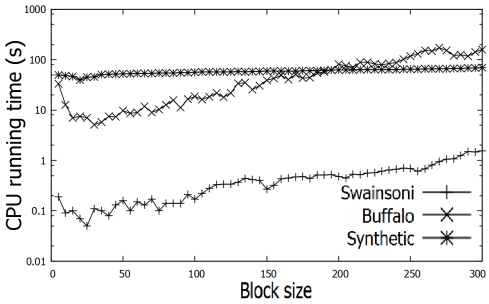



Fig. 19. Running time w.r.t block size.

CuTS* stop due to a lack of memory capacity after processing 300 million locations. In addition, the ObjectGrowth cannot provide the results after 1 day running. The main reason is that with low $min_t$ (= 1), the search space is significant larger ($\approx 2^{50,000}$). Meanwhile, thanks to the LCM approach, our algorithms can provide the results within hours (Figure 18).

**Efficiency w.r.t.** *Block-size*. To investigate the optimal value of block-size, we examine the Incremental GeT_Move by using the default values of $\epsilon$ and $min_t$ with different block-size values on real datasets and synthetic dataset ($|O_{DB}| = 500, |T_{DB}| = 1,000$). The optimal block-size range can be from 20 to 30 timestamps within which the Incremental GeT_Move obtains the best performance for all the

datasets (Figure 19). This is because objects tend to move together in suitable short interval (from 20 to 30 timestamps). Therefore, by setting the block-size in this range, the data is efficiently compressed into FCIs. Meanwhile, with larger block-size values, the movements of objects are quite different; therefore, the data compressing is not so efficient. Regarding to small block-size values (from 5 to 15), we have to face up to a large number of blocks so that the process is slowed down. In the previous experiments, block-size is set to 25.

5.2.2. *Parameter Free Incremental GeT_Move Efficiency*

So far, the Incremental GeT_Move and the GeT_Move outperform the other algorithms. In addition, our proposed approaches can work with low values of $\varepsilon$ and $min_t$. In this section, we examine the efficiency of the Parameter Free Incremental GeT_Move algorithm. In this experiment, we compare performances of six algorithms: 1) the Parameter free Incremental GeT Move, named Nested Incremental GeT_Move, 2) a Nested GeT_Move which is the application of the GeT_Move on nested cluster matrices $CM_N$, 3) the Incremental GeT_Move which is executed with the optimal values of block size on original cluster matrices $CM$, 4) the GeT_Move which is applied on original cluster matrices CM, 5) the CanTree which is applied on original cluster matrices, and 6) a Nested CanTree which is the application of the CanTree on (*horizontal*) nested cluster matrices[g].

**Efficient w.r.t.** *Real datasets.* Figures 20, 21 show that the Nested Incremental GeT_Move greatly outperforms the other algorithms. This is because of the better performance of the LCM algorithm on nested cluster matrices (resp. fully nested blocks) compared to the original cluster matrices. Essentially, with the nested cluster matrix, the number of combinations of frequent itemsets $X$ and items $i$ to ensure the closeness is greatly reduced. Therefore, the performance of the LCM algorithm is much improved. In fact, the Nested GeT_Move is always better than the GeT_Move (Figures 20, 21, 22). In addition, the Swainsoni and Buffalo datasets contain many fully nested blocks (Table 6 and Figure 6). Consequently, the Nested Incremental GeT_Move is more efficient than the other algorithms.

**Efficient w.r.t.** *Synthetic dataset.* In fact, the Nested Incremental GeT_Move is quite similar to the Nested GeT_Move (Figure 22). This is because: 1) the synthetic data is very sparse, 2) there are few fully nested blocks, and 3) the nested blocks contain a very small number of items (i.e., 0.1% matrix fill by '1' and only 8 fully nested blocks which average length is 2, see Table 6). Therefore, the processing time of nested blocks is quite short. Meanwhile, there is a large nested sparse block which is the main partition that need to be processed by both the Nested Incremental GeT_Move and the Nested GeT_Move.

In addition, thanks to the nested sparse block, the performance of LCM is improved a lot. Therefore, the Nested Incremental GeT_Move and the Nested

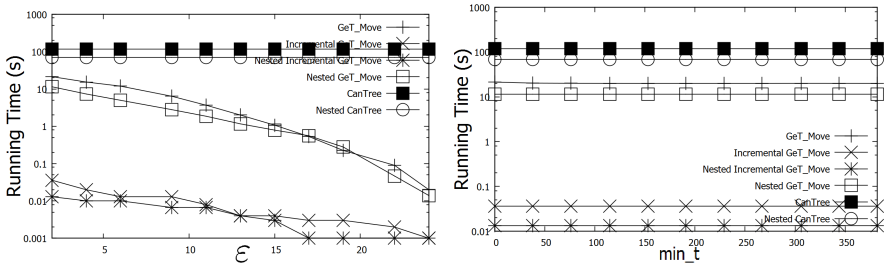
[g]Horizontal nested cluster matrix is a nested matrix so that any two rows $r_i$ and $r_{i+1}$ we have $r_i \cup r_{i+1} = r_{i+1}$.

Table 6. Fully nested blocks on datasets.

| Dataset | Matrix fill | #Nested blocks | avg.length |
|---|---|---|---|
| Swainsoni | 17.8% | 102 | 4.52 |
| Buffalo | 7.2% | 602 | 2.894 |
| Synthetic | 0.1% | 8 | 2.00 |

Table 7. Computational results using real and synthetic datasets, running time [s].

| Dataset | BiMax | FCPMiner | Incremental GeT_Move | Nested GeT_Move | Nested Incremental GeT_Move |
|---|---|---|---|---|---|
| Swainsoni | 95.32 | 1.45 | 0.016 | 11.21 | **0.012** |
| Buffalo | 323.45 | 68.9 | 39.23 | 105.82 | **1.9** |
| Synthetic | NA | 144.15 | 118.47 | 29.35 | **29.35** |



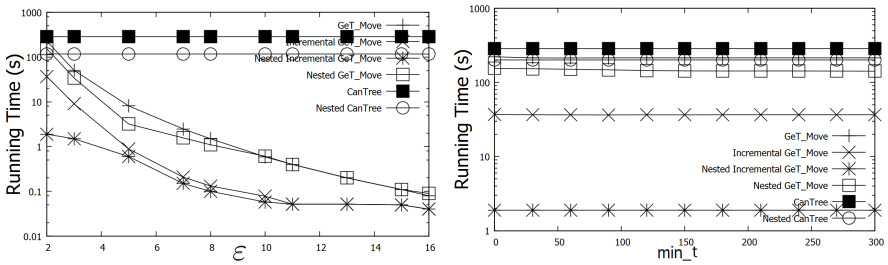

(a) Running time w.r.t. $\varepsilon$ (b) Running time w.r.t. $min_t$

(c) Running time w.r.t. $|O_{DB}|$ (d) Running time w.r.t. $|T_{DB}|$

Fig. 20. Running time on Swainsoni dataset.

GeT_Move are better than the others in most of cases. With small number of objects $|O_{DB}|$ (i.e., $|O_{DB}| = 50$, Figure 22c) or high $\varepsilon$ (i.e., $\varepsilon \geq 9$, Figure 22a), the Incremental GeT_Move is slightly better than the Nested Incremental GeT_Move and the Nested GeT_Move. The main reason is that the Incremental GeT_Move splits the cluster matrix $CM$ into different small blocks within which there are a small number of items and FCIs. Thus, the computation cost is reduced. On the other hand, the Nested Incremental GeT_Move and the Nested GeT_Move need to work with a large nested sparse block.

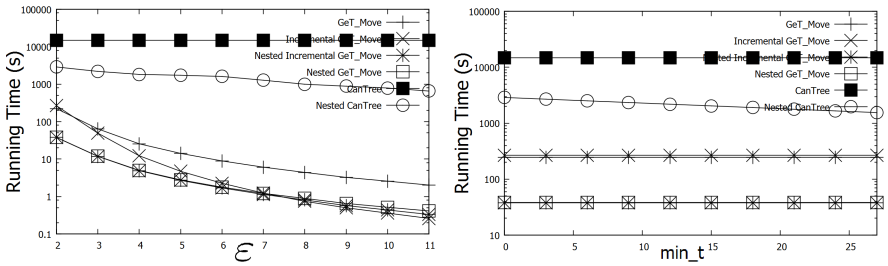
To further illustrate the efficiency of our proposed parameter free Incremental GeT_Move, we compare our models with state-of-the-art biclustering algorithms,




(a) Running time w.r.t. $\varepsilon$ (b) Running time w.r.t. $min_t$

(c) Running time w.r.t. $|O_{DB}|$ (d) Running time w.r.t. $|T_{DB}|$

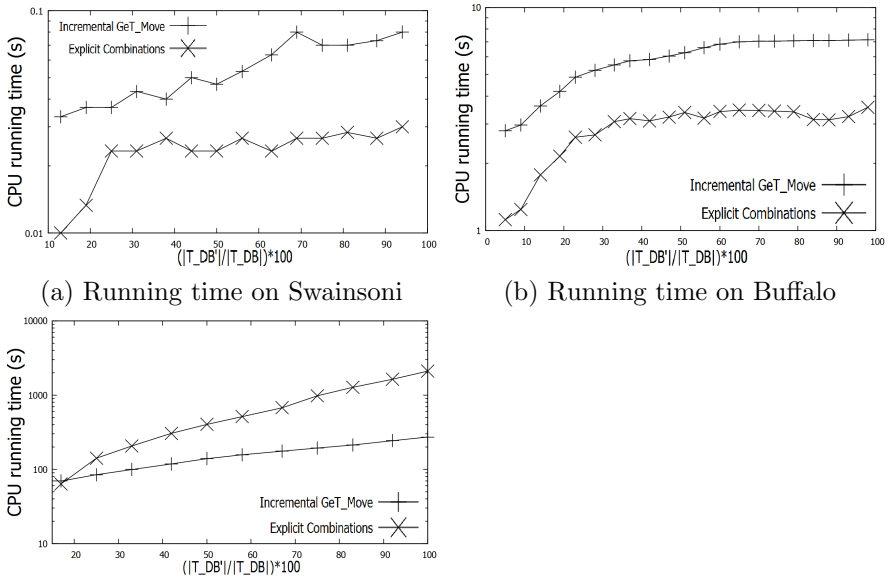
Fig. 21. Running time on Buffalo dataset.




(a) Running time w.r.t. $\varepsilon$ (b) Running time w.r.t. $min_t$

(c) Running time w.r.t. $|O_{DB}|$ (d) Running time w.r.t. $|T_{DB}|$

Fig. 22. Running time on Synthetic dataset.




(a) Running time on Swainsoni

(b) Running time on Buffalo

(c) Running time on Synthetic data

Fig. 23. Explicit combination algorithm efficiency.

i.e., BiMax[43] and FCPMiner[44], in terms of executing time. Our algorithms including Incremental GeT_Move, Nested GeT_Move, and Nested Incremental GeT_Move extract all the possible closed swarms, convoys, and group patterns existing in the datasets. Meanwhile, the BiMax and FCPMiner only extract closed frequent itemsets from the datasets. Table 7 shows the computation time of the five algorithms. Each algorithm is executed 10 times, then we take the average computational time. We can see that the Incremental GeT_Move and Nested Incremental GeT_Move outperform other algorithms including BiMax and FCPMiner. FCPMiner is slower than our Incremental GeT_Move and Nested Incremental GeT_Move because it recursively build up the frequent closed patterns by taking the consecutive rows one-by-one and recording only those column in very large binary matrices of the datasets. The Nested Incremental GeT_Move algorithm achieves the best computation time.

### 5.2.3. *Object Movement Pattern Mining Algorithm Based on Explicit Combination of FCI Pairs*

In this section, an experiment is designed to examine the movement pattern mining algorithm based on explicit combination of FCI pairs and to identify when we should update the database. We first use half of the Swainsoni, Buffalo and Synthetic datasets as a $DB$. Then the other half is used to generate $DB'$ which is increased step by step up to the maximum size (Figure 23). In this experiment, the

Incremental GeT_Move is employed to extract FCIs from $DB$ and $DB'$.

For the real datasets (Swainsoni and Buffalo), the explicit combination algorithm is more efficient than the Incremental GeT_Move in all cases (Figures 23a, b). This is because we already have $FCIs_{DB}$ and therefore we only need to extract $FCIs_{DB'}$ and then combine $FCIs_{DB}$ and $FCIs_{DB'}$. In addition, the Swainsoni and Buffalo are sufficiently dense (i.e., 17.8% and 7.2% with large number of fully nested blocks, see Table 6) so that the numbers of FCIs in $FCIs_{DB}$ and $FCIs_{DB'}$ are not huge. Consequently, the number of combinations is reduced and thus the algorithm is more efficient. In Figures 23a-b, we can consider that the running time of the explicit combination algorithm significantly changes when $|T_{DB'}| > 15\%|T_{DB}|$. This means that it is better to update the database when $|T_{DB'}| < 15\%|T_{DB}|$. Regarding the synthetic dataset, the explicit combination algorithm is only efficient on small $DB'$ (i.e., $|T_{DB'}| < 20\%|T_{DB}|$, Figure 23c) because the dataset is very sparse. In fact, the number of FCIs in $FCIs_{DB'}$ is enlarged when the size of $DB'$ increases. Thus, the explicit combination algorithm is not efficient because of the huge number of combinations. Overall, we can consider that the explicit combination algorithm obtains good efficiency when $T_{DB'}$ is smaller than 15% of $T_{DB}$.

To summarize, the Incremental GeT_Move and the GeT_Move outperform the other algorithms. Our algorithms can work with low values of $\varepsilon$ and $min_t$. To reach the optimal efficiency, we propose a parameter free Incremental GeT_Move (resp. the Nested Incremental GeT_Move) which dynamically assigns fully nested blocks for the algorithm from the nested cluster matrix. The experimental results show that the efficiency is greatly improved with the Nested Incremental GeT_Move and the Nested GeT_Move. Furthermore, by storing FCIs in a closed itemset database (see Figure 7), it is possible to reuse them whenever new object movements arrive. The experimental results show that it is better to update the database when $T_{DB'}$ is smaller than 15% of $T_{DB}$ by applying the explicit combination algorithm.

## 6. Conclusions

In this paper, we propose a (parameter free) unifying incremental approach to automatically extract different kinds of object movement patterns by applying frequent closed itemset mining techniques. Their effectiveness and efficiency have been evaluated by using real and synthetic datasets. Experiments show that our approaches outperform traditional ones.

Another issue we plan to address is how to take into account the arrival of new objects which were not available for the first extraction. Now, as we have seen, we can store the results (resp. FCIs) to improve the process when new object movements arrive. Recently, we take the hypothesis is that the number of objects remains the same. However in some applications these objects could be different.

## References

1. J. Gudmundsson and M.V. Kreveld. *Computing longest duration flocks in trajectory data*. In GIS'06, pp. 35-42.

2. M.R. Vieira, P. Bakalov, and V.J. Tsotras. *On-line Discovery of Flock Patterns in Spatio-Temporal Data*. In GIS'09, pp.286-295.
3. H. Jeung, M.L. Yiu, X. Zhou, C.S. Jensen, and H.T. Shen. *Discovery of Convoys in Trajectory Databases*. PVLDB 2008, 1(1):1068-1080.
4. P. Kalnis, N. Mamoulis, and S. Bakiras. *On Discovering Moving Clusters in Spatio-temporal Data*. In SSTD'05, p.. 364-381.
5. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*. In KDD'96, pp. 226-231.
6. Z. Li, B. Ding, J. Han, and R. Kays. *Swarm: Mining Relaxed Temporal Moving Object Clusters*. In VLDB'10, pp. 723-734.
7. Y. Li, J. Han, and J. Yang. *Clustering moving objects*. In KDD'04, pp. 617-622.
8. R. Taouil, N. Pasquier, Y. Bastide, and L. Lakhal. *Mining bases for association rules using closed sets*. In ICDE'00, pp. 307-307.
9. M.J. Zaki. *Mining non-redundant association rules*. DMKD, 9(3):223-248, 2004.
10. C. Lucchese, S. Orlando, and R. Perego. *Fast and memory efficient mining of frequent closed itemsets*. TKDE, 18(1):21-36, 2006.
11. R. Agrawal and R. Srikant. *Fast algorithms for mining association rules*. In VLDB'94, pp. 487-499.
12. B. Goethals and M.J. Zaki. *Frequent Itemsets Mining Implementations*. In ICDM'03 Workshop on Frequent Itemset Mining Implementations, volume 90 of CEUR Workshop Proceedings, 2003.
13. Z. Li, M. Ji, J.-G. Lee, L. Tang, Y. Yu, J. Han, and R. Kays. *Movemine: Mining moving object databases*. In SIGMOD'10, pp. 1203-1206.
14. J. Gudmundsson, M.J. van Kreveld, and B. Speckmann. *Efficient detection of motion patterns in spatio-temporal data sets*. In GIS'04, pp. 250-257.
15. P. Laube and S. Imfeld. *Analyzing relative motion within groups of trackable moving point objects*. In GIS'02, pp. 132-144.
16. H. Jeung, X. Zhou, and H. T. Shen. *Convoy Queries in Spatio-Temporal Databases*. In ICDE'08, pp. 1457-1459.
17. F. Verhein. *Mining Complex Spatio-Temporal Sequence Patterns*. In SDM'09, pp. 605-616.
18. C.S. Jensen, D. Lin, and B.C. Ooi. *Continuous clustering of moving objects*. In KDE'07, pp. 1161-1174, issn: 1041-4347.
19. W.A. Kosters, W. Pijls, and V. Popova. *Complexity Analysis of Depth First and FP-Growth Implementations of APRIORI*. In MLDM'03, pp. 284-292.
20. V. Bogorny and S. Shekhar. *Spatial and Spatio-Temporal Data Mining*. Tutorial on Spatial and Spatio-Temporal Data Mining, ICDM'10.
21. Y. Wang, E.-P. Lim, and S.-Y. Hwang. *Efficient Mining of Group Patterns from User Movement Data*. In DKE'06, pp. 240-282.
22. H. Cao, N. Mamoulis, and D.W. Cheung. *Discovery of Collocation Episodes in Spatiotemporal Data*. In ICDM'06, pp.823-827.
23. J.-g. Lee and J. Han. *Trajectory Clustering: A Partition-and-Group Framework*. In SIGMOD'07, pp. 593-604.
24. C. Lucchesse, S. Orlando, and R. Perego. *DCI-Closed: A fast and memory efficient algorithm to mine frequent closed itemsets*. In FIMI'04, volume 126 of CEUR Workshop Proceedings, 2004.
25. N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. *Mining, Indexing, and Querying Historical Spatiotemporal Data*. In KDD'04, pp. 236-245.
26. T. Uno, M. Kiyomi, and H. Arimura. *LCM ver. 2: Efficient mining algorithms for*

*frequent/closed/maximal itemsets*. In FIMI'04, volume 126 of CEUR Workshop Proceedings, 2004.
27. J. Han, H. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation*. In SIGMOD'00, pp. 1-12.
28. J. Han, Z. Li, and L.A. Tang. *Mining Moving Object, Trajectory and Traffic Data*. In DASFAA'10 (tutorial), 2010.
29. A. O. C. Romero. *Mining moving flock patterns in large spatio-temporal datasets using a frequent pattern mining approach*. Master Thesis, University of Twente, faculty ITC, March 2011.
30. T. Uno, T. Asai, Y. Uchida, and H. Arimura. *An efficient algorithm for enumerating closed patterns in transaction databases*. In DS'04, pp. 16-31.
31. H. Mannila, E. Terzi. *Nestedness and Segmented Nestedness*. In KDD'07, pp. 480-489.
32. P. N. Hai, P. Poncelet, and M. Teisseire. *GeT_Move: An Efficient and Unifying Spatio-Temporal Pattern Mining Algorithm for Moving Objects*. In IDA'12, pp. 276-288.
33. P. N. Hai, D. Ienco, P. Poncelet, and M. Teisseire. *Extracting Trajectories through an Efficient and Unifying Spatio-Temporal Pattern Mining System*. In ECML-PKDD'12 (demo paper), pp. 820-823.
34. H. Yoon, and C. Shahabi. *Accurate Discovery of Valid Convoys from Moving Object Trajectories*. In SSTDM'09, pp. 636-643.
35. C. K.-S. Leung, Q. I. Khan, and T. Hoque. *CanTree: A Tree Structure for Efficient Incremental Mining of Frequent Patterns*. In ICDM'05, pp. 274-281.
36. H. Steinhaus. *Sur la division des corp materiels en parties Bull*. Acad. Polon. Sci, Vol. 1 (1956), pp. 801-804.
37. L. Kaufman and P.J. Rousseeuw. *Clustering by means of Medoids*. In Statistical Data Analysis Based on the L_1-Norm and Related Methods, pp. 405-416, 1987.
38. J. A. Hartigan. *Direct Clustering of a Data Matri*. JASA, volume 67, number 337, 1972, pages 123-129.
39. Y. Cheng and G.M. Church. *Biclustering of Expression Data*. In ISMB'00, pp. 93-103.
40. L. Lazzeroni and A. Owen. *Plaid models for gene expression data*. Statistica Sinica, volume 12, 2002, pages 61-86.
41. A. Ben-Dor, B. Chor, R. Karp, and Z. Yakhini. *Discovering Local Structure in Gene Expression Data: The Order-preserving Submatrix Problem*. J Comput Biol. 2003;10(3-4):373-84.
42. S. Bergmann, J. Ihmels, and N. Barkai. *Iterative signature algorithm for the analysis of large-scale gene expression data*. Phys Rev E Stat Nonlin Soft Matter Phys, volume 67, number 3, 2003.
43. A. Prelic, S. Bleuler, P. Zimmermann, et al. *A systematic comparison and evaluation of biclustering methods for gene expression data*. Bioinformatics 2006:22(9):11229.
44. A. Kiraly, A. Laito, J. Abonyi, and A. Gyenesei. *Novel techniques and an efficient algorithm for closed pattern mining*. Expert Systems with Applications, volume 41, issue 11, 2014, pages 5105-5114.
45. G. Kou, Y. Lu, Y. Peng, and Y. Shi. *Evaluation of Clustering Algorithms for Financial Risk Analysis using MCDM Methods*. IJITDM, volume 41, number 01, 2012, pages 197-225.
46. Y. Peng, G. Kou, Y. Shi, and Z. Chen. *Evaluation of Classification Algorithms using MCDM and Rank Correlation*. IJITDM, volume 7, number 04, 2008, pages 639-682.

Table 8. Jaccard similarity between extracted closed swarms under different clustering methods on Swainsoni dataset. Radius-based rigid algorithm with $r = 3$ and K-mean with $k = 5$.

| | DBScan | Radius-based rigid | K-mean |
|---|---|---|---|
| DBScan | 1 | 0.32258 | 0.39288 |
| Radius-based rigid | 0.32258 | 1 | 0.31969 |
| K-mean | 0.39288 | 0.31969 | 1 |

## Appendix A. Obtaining Clusters

The clustering method is not fixed in our system. Users can cluster cars along highways using a density-based method, or cluster birds in 3 dimension space using the k-means algorithm[36]. Clustering methods that generate overlapping clusters are also applicable, such as EM algorithm or using a rigid definition of the radius to define a cluster. Moreover, clustering parameters are decided by users' requirements or can be indirectly controlled by setting the number of clusters at each timestamp.

Usually, most of clustering methods can be done in polynomial time. In our experiments, we used DBScan[5], which takes $O(|O_{DB}|log(|O_{DB}|) \times |T_{DB}|)$ in total to do clustering at every timestamp. To speed it up, there are also many incremental clustering methods for moving objects. Instead computing clusters at each timestamp, clusters can be incrementally updated from last timestamps.

In fact, different clustering algorithms can generate different object-cluster matrix and thus the extracted movement patterns will be different as well. However, the clustering algorithm depends on application domains and indeed users have many ways to reprocess the data. Indeed, we can show the common movement patterns under different clustering methods. For instance, Table 8 expresses the *Jaccard similarity* between sets of closed swarms under three different clustering algorithms. The Jaccard similarity can be defined as follows:

**Definition Appendix A.1.** *Jaccard Similarity for Movement Patterns.* Given two sets of closed swarms $CS = \{cs_1, \ldots, cs_n\}$ and $CS' = \{cs'_1, \ldots, cs'_m\}$, the jaccard similarity between $CS$ and $CS'$ is given as follows:

$$Jaccard(CS, CS') = \frac{\sum_{cs \in CS} \sum_{cs' \in CS'} (\frac{|O(cs) \cap O(cs')|}{|O(cs) \cup O(cs')|} \times \frac{|T_{cs} \cap T_{cs'}|}{|T_{cs} \cup T_{cs'}|})}{|CS| \times |CS'|} \quad \text{(A.1)}$$

Given the nature of data, some trajectories could exist under different clustering algorithms. However, they usually are short segments in much longer trajectories. It is hard to analyze and utilize these short segments. In our experiments, we did extract them however they do not give us any meaningful results. There is no guarantee that the common ones are meaningful for the end users. Indeed, in many cases the common patterns express already known information which is not interesting for analysts. For instance, Incremental GeT_Move has also been successfully applied to extract the interactions between genes after patients taken HIV treatments and the clustering method is a gene segment-based clustering since other clustering

methods have been proved to provide irrelevant results. Note that we have ground trust given by bio-informatics experts. Obviously, the clustering approach for re-processing data clearly depends on the application domain and thus our concern is focusing on movement pattern mining from object-cluster matrix.

In addition, the clustering algorithms used in our experiments belong to different categories and they have different characteristics. The extracted trajectories could be different. Therefore, the Jaccard similarities might be low ($\approx 0.3$). Given similar clustering algorithms such as k-means[36] and k-medoids[37], the Jaccard similarity potentially is higher. In addition, clustering algorithms can be evaluated by using MCDM methods [45,46] before applying our proposed algorithms to extract movement patterns.