DG2CEP: A Density-Grid Stream Clustering Algorithm based on Complex Event Processing for Cluster Detection

Marcos Roriz, and Markus Endler

Department of Informatics
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro – RJ – Brazil

{mroriz, endler}@inf.puc-rio.br

Abstract. *Applications such as fleet and mobile task force management, or traffic control can largely benefit from the on-line detection of collective mobility patterns of vehicles, goods or persons. A common mobility pattern is a cluster, a concentration of mobile nodes in a certain region, e.g., a mass protest, a rock concert, or a traffic jam. Current approaches require previous knowledge of the locations where the cluster might happen. In this paper, we propose* DG2CEP, *an algorithm inspired by data mining algorithms and based on Complex Event Processing, for on-line detection of such clusters. It can detect the formation and dispersion of clusters from streams of position data without the need of specifying the possible locations of clusters in advance.*

1. Introduction

Several distributed applications [Amini et al. 2014], such as fleet and mobile task force management, air/road traffic control, can benefit from an on-line detection of collective mobility patterns of their mobile nodes, *i.e.*, the ability to detect when a given set of mobile nodes are collectively moving according to a certain pattern. A common collective mobility pattern is a cluster, a concentration of mobile nodes in a certain region, *e.g.*, a mass protest, a rock concert, or a traffic jam. The detection of this pattern is motivated by reasons such as ensuring safety of the mobile nodes, identifying suspicious or hazardous behaviors, ensuring availability of resources, or optimizing global operation of the mobile entities. In road traffic control, for example, it is important to early detect some traffic jam caused by a complete or partial obstruction of a street/road. Moreover, in some cases it may be important also to detect the fast dispersion of a cluster, *e.g.* a crowd rushing away from some specific spot in an environmental disaster scenario. This information can be useful for dispatching additional rescue staff to the place.

As can be seen from these examples, unexpected clustering of mobile nodes is a recurrent pattern in several applications. However, detecting such pattern poses several challenges to those applications [Silva et al. 2013]. First, it has to process the high volume of position data sent by mobile nodes in a timely manner. Secondly, it has to design efficient pattern detection algorithms to cope with the complexity of the position data comparisons. Third, it has to consider evolving data streams, *i.e.*, that the set of monitored mobile nodes is open and variant, with new elements joining and other leaving the stream (*e.g.* new vehicles entering or leaving a city perimeter), making the cluster boundary difficult to detect. Finally, it must be able to detect arbitrary clusters shapes, *i.e.*, for example, a traffic jam that ranges over several streets or neighborhoods.

The majority of solutions found in the literature only address partially these problems. For example, a series of data stream clustering algorithms [Aggarwal et al. 2003; Cao et al. 2006; Tu and Chen 2009] do find clusters in position data streams, but do not consider that the set of monitored node is variant. Thus, they are restricted to a static data stream, *i.e.*, a fixed mobile nodes set. Solutions that do consider open/variable node sets [Barouni and Moulin 2012; Kim et al. 2011], require developers to specify the possible location of clusters in advance. Precisely, they can only detect clusters in a region previously specified. This fact restricts the cluster boundary to pre-defined regions.

To address these issues, in this paper, we propose DG2CEP, (Density-Grid Clustering using Complex Event Processing) an on-line algorithm that combines data mining clustering algorithms [Han et al. 2011] with Complex Event Processing (CEP) concepts [Luckham 2001], to detect the formation and dispersion of clusters based on position data streams. DG2CEP, expressed as a set of simple CEP rules, taking advantage of CEP stream-oriented concepts. The main contributions of our paper are twofold:

- Proposal of an on-line cluster detection algorithm, described as a set of CEP rules, that analyses streams of position data streams;
- Description of the cluster dispersion pattern, through alternative CEP rules, but using the same algorithm. Surprisingly, so far, cluster dispersion detection based on position data streams has not been explored and described in literature.

The remainder of the paper is structured as follows. Section 2 gives an overview of the fundamental concepts used in the paper. Section 3 presents our algorithm, DG2CEP, for clustering position data streams. Section 4 presents a proposed evaluation of our work. Section 5 reviews and compares related works our approach. Finally, Section 6 presents some concluding remarks and future works towards this work.

2. Fundamentals

We use the same definition of a cluster that is adopted in DBSCAN [Ester et al. 1996], a classic data mining clustering algorithm. Thus, in this section we briefly review the DBSCAN cluster definition and algorithm. After that, we briefly explain CEP concepts, such as events and rules, that are used throughout the paper to express our algorithm.

2.1 Clustering Algorithms

Clustering is the process of grouping data into one or more sets. In essence, a cluster is a collection of data objects that are similar, to each other [Han et al. 2011], for example, group mobile nodes based in their mutual Euclidean distance. The majority of clustering algorithms are based on K-MEANS [Jain 2010], which divides the data into *k* sets. Since we do not know the number *k* of clusters ahead, we need to use a different approach.

DBSCAN [Ester et al. 1996] is a clustering algorithm based on the concept of node density. The algorithm assigns a density value, called *ε-Neighborhood*, to each node. It defines the set of nodes that are within distance ε of a given node. Thus, in DBSCAN a cluster is found when a mobile node *core* consists of a minimum number (*minPts)* of neighbors in its *ε-Neighborhood*. Both, the *core* mobile node and its neighbors are added to the cluster. The main idea of DBSCAN is to recursively check each neighbor so as to expand the cluster, i.e. for each neighbor added, if it also contains

minPts neighbors, its neighbors are also added to the cluster, which in turn are recursively visited. Thus, DBSCAN recursively processes, visits, adds, and expands the cluster using the nodes' neighbors.

The bottleneck of DBSCAN is the *ε-Neighborhood* function [Han et al. 2011]. During the algorithm, this function is recursively called to retrieve the mobile node density. The problem with this function is that it has to compare all pairs of nodes, to identify those that are within the ε distance. This pairwise mutual comparison take $O(n)$ per node, turning the algorithm quadratic $O(n^2)$. Since the DBSCAN algorithm was designed for static datasets, it can optimize this process, by storing the nodes' position data in a spatial index (*e.g.*, R-Tree or Quad-Tree). These data structures can reduce the neighbor finding function to $O(\log n)$ per node, thus reducing the total complexity of the DBSCAN algorithm to $O(n \log n)$. However, when we consider online cluster detection based on position data (data streams) this primary premises becomes troublesome. It is well known that is practically impossible to maintain a spatial index for online data, both due to its size - each node may have many neighbors - and because it is very costly to continuous update and maintain the spatial data structure [Garofalakis et al. 2002]. As fallback, in data stream, DBSCAN goes back to the strategy of comparing pairwise the node's location updates, returning to the high cost of $O(n^2)$.

On the other hand, grid-based clustering algorithms [Chen and Tu 2007] can be used to scale the detection process, as grids provide static references (*i.e.*, the grid cells) for the clustering problem. In this approach, the mobile nodes are mapped to grid cells, and the cost is proportional to the number of cells rather than the number of nodes [Amini et al. 2014]. Each grid cell thus only "holds" the mobiles nodes that are within the cell's geographic area. The density of each cell is calculated as the number of mobile nodes mapped to the cell divided by the mean density of the grid, *i.e.*, of all cells. Note that this view provides a different density semantics than DBSCAN's notion of density. Here, clusters are detected based on the density of all cells rather than the density of the neighborhood of a node, as in DBSCAN. This semantics requires that the algorithm must store and calculate the density of the entire grid at every location update, which is quite costly when processing streams of position data. In addition, grid based algorithms usually divide the space in a small number of cells to reduce the cost of recalculating the grid density, thus reducing the cluster precision. To complicate even further, these algorithms need to manage each cell, *e.g.*, to store and retrieve the nodes in the grid cells.

2.2 Complex Event Processing (CEP)

To efficiently handle and process position data streams, we explored Complex Event Processing (CEP) [Luckham 2001] concepts. CEP provides a set of stream-oriented concepts that facilitates the processing of events. It defines the data stream as a sequence of events, where each event is an occurrence in a domain. In our case, an event is a location update of a mobile node, which contains the new coordinate of the node's position.

CEP provides concepts for processing events, which includes the consumption, manipulation (aggregation, filtering, transformation, etc.) and production of events. The CEP workflow continuously process the events received, which are then manipulated and sent to event consumers (*e.g.* online monitoring applications), that are interested in receiving notifications about detected situations.

The manipulations of events are described by CEP rules. CEP rules are Event-Condition-Action functions that use operators (*e.g.* logical, quantifying, counting, temporal, and spatial) on received events, checking for correlations among these events, and generating complex (or composite) events that summarize the correlation of the input events. Most CEP systems have the concept of Event Processing Agents (EPAs), which are software modules that implement an event processing logic between event producers and event consumers, encapsulating some operators and CEP rules. The type of an EPA is defined by the behavior of the CEP rules it implements, such as filtering, transformation or specific event pattern detection. In addition, CEP rules can manipulate directly the event stream by adding, removing or updating the raw events. Event streams that enable their raw events to be manipulated are called named window [Codehaus 2014].

An Event Processing Network (EPN) is a network of interconnected EPAs that implements the global processing logic for pattern detection through event processing [Luckham 2001]. In an EPN the EPAs are conceptually connected to each other (*i.e.* output events from one EPA are forwarded and further processed by other EPA) without regard to the particular kind of underlying communication mechanism between them.

Before we insert events into our EPN, we group them into a specific partition using CEP's *context*. A context takes a set of events and classifies it in one or more partitions [Etzion and Niblett 2010]. For example, it may group all events whose coordinate fall inside a latitude interval into a CEP context partition. The CEP context concept resembles grid cells, mentioned previously. However, the primary difference between them is on abstraction level. While a grid cell just stores data, a CEP context partition does the same, but in addition also represents an isolated CEP runtime within that context partition, *i.e.*, EPAs operators will apply to events associated with a same partition, thus immensely reducing the processing load. In addition, they are auto-managed "grid cells" in the CEP engine [Codehaus 2014], that can be dynamically activated and deactivated based on EPAs that refer to them. For example, a context partition is created when it becomes active, *i.e.*, an event is mapped to it. As soon as there is no active EPA, *i.e.*, it is not processing or storing an event window, the partition becomes inactive.

Finally, we also use the CEP concept of *sliding time windows*. The idea of sliding time windows is to consider only recent data within a time interval ($t_{NOW} - \delta$, t_{NOW}) [Amini et al. 2014]. For example, suppose that an EPA operates on a data stream that is using a sliding window of $\delta = 30$ seconds. When the EPA receives an event, its rules will correlate only events received within the past 30 seconds. We use this concept to ensure that we are comparing only the latest location updates in the data stream.

3. DG2CEP: A CEP-Based Position Clustering Algorithm.

To detect the formation and dispersion of clusters, we now present DG2CEP, a density-grid data stream clustering algorithm expressed as a set of CEP rules. DG2CEP provides a DBSCAN-like density-based clustering semantic, being able to approximately identify DBSCAN cluster shapes using CEP context partitions. The main idea of our algorithm is to mitigate the clustering process by first mapping the location update to CEP context partitions, and then clustering the partitions rather than the nodes. Instead of clustering each node, we map them to context partitions and cluster the partitions using a DBSCAN-like semantic. The overall processing flow is illustrated in Figure 1.

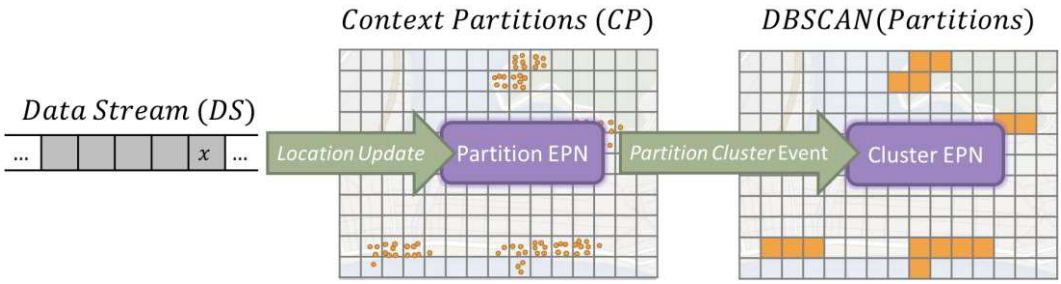

Figure 1. Overall description of DG2CEP processing flow.

3.1. Stream Receiver

First, we need to handle and map the position data stream to a context partition. Consider that the domain we want to monitor is delimited by [*minLat*, *maxLat*] and [*minLng*, *maxLng*], the respective latitude and longitude interval. To provide a precision similar to DBSCAN, we divide this interval into partitions of size ε x ε. Thus, our space is segmented into the following context partitions:

- [*minLat*, OFFSETLAT(*minLat*, ε), OFFSETLAT(*minLat*, 2ε), …, *maxLat*] and
- [*minLng,* OFFSETLNG(*minLng*, ε), OFFSETLNG(*minLng*, 2ε), …, *minLng*]

for latitude and longitude respectively. The functions[1] OFFSETLAT and OFFSETLNG combines an angle with an ε offset (in meters) to return the offset value in angle for the latitude or longitude respectively. The one-line Code 1, written in the Event Processing Language (EPL) [Codehaus 2014; Etzion and Niblett 2010], a language to express CEP rules, creates a context PARTITIONCLUSTER that segments LOCATIONUPDATE events into a specific partition according to the latitude and longitude attribute of the event.

Code 1. Context Partition Creation (in EPL).

```
CREATE context PARTITIONCLUSTER partition by lat and lng from LOCATIONUPDATE
```

We map the mobile node location update to a context partition (x, y) using the combination of its latitude and longitude attribute, as described in Algorithm 1. Since we need to map each location update from the position data stream to a context partition, we store the monitoring intervals in a static data structure, such as segment tree, binary tree or a vector, which allow us to quickly identify the location update partition using binary search. Note that, we need to calculate only once the interval data-structure, since all mobile location updates use the same static structure to identify their partition. Finally, we emit an LOCATIONUPDATE event in our EPN.

Algorithm 1. Stream Receiver Function.

Input:	
DS	Data Stream
ε	Distance Threshold
[MinLat, MaxLat]	Latitude Interval
[MinLng, MaxLng]	Longitude Interval
Output:	Location Update Event

[1] See Ed. Willlians (http://williams.best.vwh.net/avform.htm) for an implementation of these functions.

```
1.  latPosition  ← Initialize a Static Structure for Latitude    (MinLat, MaxLat, ε)
2.  lngPosition ← Initialize a Static Structure for Longitude (MinLng, MaxLng, ε)
3.
4.  while data stream DS is active do
5.      x ← read record from DS
6.
7.      partitionX ← BINARYSEARCH(x.lat, latPosition)
8.      partitionY ← BINARYSEARCH(x.lng, lngPosition)
9.      emit EVENT(node, (partitionX,  partitionY))
10. end
```

3.2. Context Partition Cluster

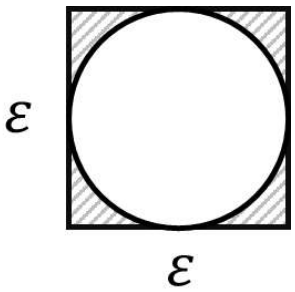

Figure 2. Context Partition and DBSCAN circle.

The events assigned to a context partition are all within distance ε (precisely at maximum $\varepsilon\sqrt{2}$ apart), which is approximately the *ε-Neighborhood* defined by the DBSCAN algorithm, as shown by Figure 2. To compensate the normalization of DBSCAN *ε-Neighborhood* - that is a circle of radius ε - to a context partition of size ε x ε in DG2CEP, we have to consider more nodes to be there for a cluster, as some nodes might be outside the DBSCAN ε-neighborhood (i.e., in the gray area). The minimum number of points to form a cluster in the square ε x ε, considering an uniform distribution of points, is the ratio between the area of the circle and of the square, *i.e.*: $\pi(\varepsilon / 2)^2 / \varepsilon^2 = \pi(\varepsilon^2 / 4) / \varepsilon^2 = \pi / 4 \sim= 1.27$. Thus, to provide a DBSCAN cluster semantic in context partition we have to consider at least 1.27*minPoints* nodes to form a cluster.

Since events in the partition are already close to each other (considering ε), to identify a cluster in our algorithm we just need to count the number of events received, as described by Algorithm 2. If the number of events received in the context partition over a period of Δ seconds (window period) is larger than 1.27*minPts*, then the partition forms a cluster and emits a complex event named PARTITIONCLUSTEREVENT. If it is not, it will store that event for Δ seconds.

Algorithm 2. Context Partition Cluster Detection EPA.

```
Input:
    Δ       Sliding window period (in seconds).
    MinPts  Minimum number of nodes (neighbors in DBSCAN) to form a cluster.
Output:     Partition Cluster Event
```

```
1. CONTEXT PARTITIONCLUSTER
2. INSERT INTO PARTITIONCLUSTEREVENT
3. SELECT   PartitionNodes.x, PartitionNodes.y, win(PartitionNodes.*)
4. FROM     LOCATIONUPDATE.win:time(2Δ sec) as PartitionNodes
5. HAVING   count(*) ≥ 1.27minPts
```

As mentioned in Section 1, it is also important to detect when a detected cluster disperses, *i.e.*, when the context partition cluster is not valid anymore. We can now

build an EPA to expresses this relationship, as shown in Algorithm 3. We define a CEP rule that fires if a PARTITIONCLUSTEREVENT *is not followed by* a PARTITIONCLUSTEREVENT another one within 2Δ seconds (see lines 5 and 6 in Algorithm 3). If the context partition does not generate a following cluster event within 2Δ, this means that the number of location updates mapped to the context partition is no longer larger than 1.27*minPts*, and, thus, no longer constitutes a cluster. We consider a time period of 2Δ, instead of just Δ, in order to tolerate the situation where the location updates mapped to the partition are generated out of sync by the corresponding mobile nodes.

Algorithm 3. Context Partition Dispersion Detection EPA.

Input:
Δ Sliding window period (in seconds).
Output: Partition Disperse Event

```
1. CONTEXT PARTITIONCLUSTER
2. INSERT INTO PARTITIONDISPERSEEVENT
3. SELECT PARTITIONCLUSTEREVENT.lat, PARTITIONCLUSTEREVENT.lng
4. FROM PATTERN
5.   [ EVERY PARTITIONCLUSTEREVENT →
6.   (TIMER:INTERVAL(2Δ sec) AND NOT PARTITIONCLUSTEREVENT) ]
```

3.3. Cluster Detection

Currently, in our EPN flow, we are detecting the formation and dispersion of context partitions clusters. To provide a DBSCAN-like cluster semantics, of finding arbitrary shapes, we must be able to aggregate partition clusters. Thus, we define a named window, CLUSTERS, to merge context partitions events. A named window is an event stream, but with the distinctive feature that it allows manipulation of its raw events. CLUSTERS events are composed of a single field, an array of PARTITIONDISPERSEEVENT, that refers to the context partitions that form the cluster. We manipulate the events in the named window to group/merge and divide/split clusters, *e.g.*, we merge a PARTITIONCLUSTEREVENT event with a cluster when the partition is a neighbor of the cluster, or split a cluster when we receive a PARTITIONDISPERSEEVENT event.

Now, we define an EPA to add or merge partition clusters with the CLUSTERS named window using an extraction query, as shown in Algorithm 4. When we receive a PARTITIONCLUSTEREVENT we check if there is any neighbor cluster in CLUSTERS. The ISNEIGHBOR routine does this verification by comparing the cluster's boundaries. If the cluster partitions is an adjacent with respect to the PARTITIONCLUSTEREVENT, *i.e.*, if difference between their *x*, and *y* indexes are both less than one.

Algorithm 4. Add or Merge Cluster EPA.

Input:
Δ Sliding window period (in seconds).
Output: PartitionCluster

```
1. ON PARTITIONCLUSTEREVENT as PARTITIONCLUSTER
2. SELECT AND DELETE PARTITIONCLUSTER, win(CLUSTER.*) as Neighbors
3. FROM   CLUSTERS.win:time(Δ sec) as CLUSTER
4. WHERE ISNEIGHBOR(PARTITIONCLUSTER, CLUSTER)
```

The past EPA extracts clusters that are neighbors of the PARTITIONCLUSTEREVENT event. If the EPA returns an empty set for *Neighbors*, that is, if there are no cluster that are neighbors of that context partition cluster, then the rule creates and inserts a new cluster in CLUSTERS, that is composed by a single PARTITIONCLUSTEREVENT event. If the *Neighbors* set is non-empty, *i.e.*, the EPA identified clusters that are neighbors with the partition cluster, we need to merge them with the partition. The resulting cluster is formed by the combination of all neighbors' with the PARTITIONCLUSTEREVENT, since the event will serve as a link to connect the neighbors' clusters. Thus, the resulting cluster is formed by the union of all cluster partitions from *Neighbors* with PARTITIONCLUSTEREVENT, which is reinserted in CLUSTERS, as shown in Algorithm 6.

Algorithm 5. Add or Merge Routine.

```
Input:
 PARTITIONCLUSTER   Partition Cluster
        NEIGHBORS   List of neighboring clusters
Output:             Partition Cluster
```

```
1.  partitionArray ← [PARTITIONCLUSTER]
2.  if NEIGHBORS is not empty then
3.     foreach cluster from NEIGHBORS do
4.        partitionArray ← partitionArray ∪ NEIGHBORS.partition
5.     end
6.  end
7.  clusterEvent ← Event(partitionArray)
8.  emit clusterEvent
```

When a partition cluster disperses, we need to reflect the change in CLUSTERS. Thus, we describe an EPA that when receives a PARTITIONDISPERSEEVENT, that will split, if necessary, the cluster that hold the partition into one or more clusters, as shown by Algorithm 6. The only difference between this EPA and Algorithm 4 is the CONTAINS function, which checks if a cluster contains the PARTITIONDISPERSEEVENT. After extracting the cluster that contains the cluster partition that dispersed, one needs to identify the remaining clusters after we remove that partition. This is done, by first removing the dispersed partition and then executing a classic DBSCAN on the remaining set. The separated clusters, if they exist, are then re-inserted in CLUSTERS.

Algorithm 6. Split if Necessary Routine (Disperse EPA output).

```
Input:
 PARTITIONCLUSTER   Partition Cluster
      CLUSTERLIST   Cluster Entry
Output:             Set of Partition Cluster
```

```
1.  ClusterList ← CLUSTERLIST – PartitionCluster
2.  SplitClusters ← DBSCAN(ClusterList)
3.  foreach cluster from SplitClusters do
4.     clusterEvent ← Event(partitionArray)
5.     emit clusterEvent
6.  end
```

4. Proposed Evaluation

We are still implementing and testing DG2CEP. To evaluate our work, we intend to use synthetic and real position data streams from mobile nodes. For real position data, we intend to use the T-Drive dataset [Yuan et al. 2011], which contains a one-week trajectory of 10,357 taxis in Beijing, China. The primary characteristic we want to discover is the percentage of clusters found in DG2CEP compared to those found by running DBSCAN in the dataset, *i.e.*, how close are our clusters from the original algorithm. In addition, we intend to identify the elapsed delay for the on-line cluster pattern detection and how our algorithm can scale with the number of nodes and the frequency of location updates.

5. Related Work

In this section, we briefly discuss some ongoing efforts to address data stream clustering.

Both, D-Stream [Chen and Tu 2007] and DENGRIS-Stream [Amini and Ying 2012] proposes a similar grid-based clustering algorithm for data streams. However, while the nodes position is mapped continuously to the grid cells, their algorithm operates in a bulk scheme. Their cluster algorithm runs at every specific periods, *e.g.*, every minute, thus, clusters formed during this period are not detected until the algorithm is re-executed. The clustering function performs a global search for dense cells on the grid, a high cost for streaming data. In addition, this function also update and remove sporadic cells. This extra cost eventually increases the detection period. We believe the management cost is the reason why the clustering function is only executed at certain periods. Finally, both algorithms cannot identify clusters dispersions.

Considering CEP-based solutions, as mentioned previously, the majority of works encountered [Barouni and Moulin 2012; Kim et al. 2011], requires developers to specify the possible cluster locations in advance, which is not feasible in many systems. In addition, it does not detect clusters with arbitrary shapes, such as a traffic jam in a long avenue. Previously, we proposed a CEP-based clustering approach [Baptista et al. 2013], that does a pairwise comparison between each nodes' location updates. While we successfully detected clusters, we had scaling problems due to the pairwise comparison.

6. Conclusion and Future Works

Cluster is a mobility pattern that appears in several distributed applications [Amini et al. 2014]. Nevertheless, cluster detection is a challenging task to those applications, since it requires efficient and complex algorithms to handle the high volume of position data in a timely manner. To address this issue, we proposed DG2CEP, an on-line algorithm that combines data mining clustering algorithms with CEP concepts, to detect the formation and dispersion of clusters from position data streams. DG2CEP, uses CEP stream-oriented concepts to provide a DBSCAN-like clustering as a simple sequence of CEP rules. Our algorithm is able to detect the formation and dispersion of cluster. As a weakness, we are still implementing and testing our approach, and we show no testing results. In addition, our algorithm also requires more nodes to inform a cluster than DBSCAN.

As future work, we plan to finish the implementation of our algorithm, and test with synthetic and real position data. We are also interested on exploring others stream-processing concepts, such as FGPA and GPU programming. Finally, we intend to explore others collective mobility patterns that can be expressed as a set of CEP-rules.

References

Aggarwal, C., Han, J., Wang, J. and Yu, P. (2003). A framework for clustering evolving data streams. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases - Volume 29*.

Amini, A., Wah, T. and Saboohi, H. (2014). On Density-Based Data Streams Clustering Algorithms: A Survey. *Journal of Computer Science and Technology*, v. 29, n. 1.

Amini, A. and Ying, W. (2012). DENGRIS-Stream: A density-grid based clustering algorithm for evolving data streams over sliding window. In *Proc. International Conference on Data Mining and Computer Engineering*.

Baptista, G. L. B., Roriz, M., Vasconcelos, R., et al. (2013). On-line Detection of Collective Mobility Patterns through Distributed Complex Event Processing. Monografias em Ciência da Computação 12/2013, PUC-Rio, ISSN 0103-9741.

Barouni, F. and Moulin, B. (2012). An extended complex event processing engine to qualitatively determine spatiotemporal patterns. In *Proc. of Global Geospatial Conf.*

Cao, F., Ester, M., Qian, W. and Zhou, A. (2006). Density-Based Clustering over an Evolving Data Stream with Noise. In *Proc. of the 2006 SIAM Conf. on Data Mining*.

Chen, Y. and Tu, L. (2007). Density-based Clustering for Real-time Stream Data. In *Proc. of the 13th Intl. Conf. on Knowledge Discovery and Data Mining*.

Codehaus (2014). Esper - Complex Event Processing. http://esper.codehaus.org/,

Ester, M., Kriegel, H., Sander, J. and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*.

Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. 1st. ed. Greenwich, CT, USA: Manning Publications Co.

Garofalakis, M., Gehrke, J. and Rastogi, R. (2002). Querying and Mining Data Streams: You Only Get One Look a Tutorial. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. , SIGMOD '02. ACM.

Han, J., Kamber, M. and Pei, J. (2011). *Data Mining: Concepts and Techniques*. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Jain, A. K. (2010). Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, v. 31, n. 8, p. 651–666.

Kim, B., Lee, S., Lee, Y., et al. (2011). Mobiiscape: Middleware support for scalable mobility pattern monitoring of moving objects in a large-scale city. *Journal of Systems and Software*, v. 84, n. 11, p. 1852–1870.

Luckham, D. C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. USA: Addison-Wesley., Inc.

Silva, J. A., Faria, E. R., Barros, R. C., et al. (2013). Data Stream Clustering: A Survey. *ACM Comput. Surv.*, v. 46, n. 1, p. 13:1–13:31.

Tu, L. and Chen, Y. (2009). Stream data clustering based on grid density and attraction. *ACM Transactions on Knowledge Discovery from Data*, v. 3, n. 3.

Yuan, J., Zheng, Y., Xie, X. and Sun, G. (2011). Driving with Knowledge from the Physical World. In *Proc. of the 17th Intl. Conf. on Knowledge Discovery and Data Mining*.