JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 469-488 (2018)
DOI: 10.6688/JISE.201803_34(2).0010

Priority-based Clustering in Weighted Graph Streams

MOHSEN SAADATPOUR[1], SAYYED KAMYAR IZADI[2], MOHAMMAD NASIRIFAR[1]
AND HAMED KAVOUSI[1]
[1]*Department of Computer Science*
Shahid Beheshti University
Tehran, 1983969411, Iran
[2]*Department of Computer Science*
Alzahra University
Tehran, 1993891176, Iran
E-mail: k.izadi@alzahra.ac.ir; m.saadatpour@aut.ac.ir; {m.nasirifar; h.kavousi}@sbu.ac.ir

Nowadays, analyzing social networks is one of interesting research issues. Each network could be modeled by a graph structure. Clustering the vertices of this graph is a proper method to analyze the network. However, huge amount of changes in the graph structure as a result of social network interactions implies the need of an efficient clustering algorithm to process the stream of updates in a real-time manner.

In this paper, we propose a novel algorithm for dynamic networks clustering based on the stream model. In our proposed algorithm, called *Priority-based Clustering of Weighted Graph Streams* (*PCWGS*), we provide a measure based on the importance of the frequency of recent interactions in the network to have more acceptable clusters. In *PCWGS* algorithm, a timestamp coupled with the weighted mean of the number of interactions of the network vertices are used to account edge weights. It is worth noting that, we present a data structure, which can keep useful information about the current state of the edges in the network based on update times and their weights while minimizing the required memory space in our proposed algorithm. Our simulations on real data sets reveal that *PCWGS* algorithm yields clustering with high quality and performance compared to previous state-of-the-art evolution-aware clustering algorithms.

Keywords: graph clustering, graph mining, node clustering, graph stream, social network

1. INTRODUCTION

Graphs or networks are one of the most powerful models to represent a set of entities and links between them with wide applications such as social networks, citations in scientific papers, network of the World Wide Web, and hyperlink analysis in web pages. Many algorithms are designed to analyze and extract information from these graphs [24]. Graph clustering is the process of dividing vertices of a graph into subsets, in each of which, vertices are related to each other by a similarity measure.

In online systems, most of the computer programs deal with massive data sets such as those found in online traffic monitoring, finding user forums in real time, automatic transaction monitoring of ATMs. These networks rapidly change over time, and the huge continuous streams of data they produce, need to be handled by efficient algorithms. The stream model, which is capable of handling such data sets, is usually used for algorithms that process the data in one or few passes under the restriction of limited memory [11].

The term "graph stream" usually presents itself in two models. In the first one,

Received April 29, 2017; revised June 17 & July 17, 2017; accepted August 8, 2017.
Communicated by Shyi-Ming Chen.

graph stream stands for a sequence of graphs and the purpose of clustering is to find groups of similar graphs, while in the second one, the sequence of atomic graph which changes over time is considered as graph stream. These atomic changes can be insertions or deletions of edges/vertices of the graph and the clustering aims to find groups of vertices by their structural similarity in real time or to discover stable clusters over time in evolving graphs. In this paper we examine the second model of graph streams.

1.1 The Problem Description

We Similar to the problem introduced in [28], are interested in clustering the vertices of a graph based on edge weights with respect to some constraints. Given an undirected weighted graph $G = (V, E)$, where V is the set of vertices and E is the set of edges $\{u, v, w\}$ where $u, v \in V$ and w is the associated weight. We partition the vertices of the graph G into clusters $P = \{C_1, C_2, \ldots, C_k\}$ in such a way that the total weight of inter-cluster edges (cut size) is minimized, and each cluster has at most L vertices. We assume L is fixed and $C_i \cap C_j = \varnothing$. More formally,

$$Cut(P) = Min(\{\sum w_{i,j} \mid e = (C_i, C_j, w_{i,j}) \in E, 1 \le i \le k\}) \quad (1)$$

We consider a dynamic and online version of this problem in the stream scenario, thus according to graph data stream model, queries include insertions and deletions of edges and vertices over time. We suppose G_t is the update of graph G and $P_t = \{C_1, C_2, \ldots, C_k\}$ is the set of clusters G_t is decomposed to at time t. The algorithm will incrementally update the clusters over time.

Due to massive scale of graph's input and memory limitation in stream scenario, different approaches have been devised, including random sampling [23], sliding window [7], graph sketches [3], *etc.* In our algorithm, we use a new technique to focus on recent changes and limit input data for processing. With these assumptions we must represent an online and incremental algorithm to cluster dynamic graphs in sliding window model with many changes in a short period of time. Previous offline graph clustering approaches such as [14] cannot capture cluster evolution and are not incremental. It is worth noting that these algorithms ignore the number of interactions that occur between a pair of vertices over time.

In this paper, we propose a new algorithm called *Priority-based Clustering of Weighted Graph Stream* (*PCWGS*). We consider each connected component as a cluster with respect to a constraint L for each component. To keep recent edges and strong interactions, we employ a measure, based on the time-stamp of an edge and the mean value of the number of interactions that occur between vertices of this edge in time-stamps, as the edge weight. When two clusters are merged or splitted, we are interested in stronger and more recent edges than weaker and older ones. With the purpose of having efficient processing, in our method vertices will not be allowed to move between clusters. In addition, edges will be cut in case of constraint violation if they have the minimum weight.

1.2 Motivation

The majority of available methods for the stream model are for un-weighted graphs or they ignore the importance and frequency of strong and stable interactions over time.

The evolution-aware clustering algorithm [28] uses only the timestamp parameter as the edge weight to study the process of graph clusters' evolution. Introducing a new method to cover both weighted and un-weighted graphs is one of our main goals considering the features of stream model. Working on new data structures to improve the performance of graph structure storage and considering history of edges is the next goal.

Table 1. Assigned weight and number of interactions at each time-stamp.

Time-stamp	t_1	t_2	t_3	…	t_n
Weight	C_n	${}_2C_n$	${}_3C_n$	…	${}_nC_n$
Number of interactions in each timestamp	λ_1	λ_2	λ_3	…	λ_n

Example 1: Suppose vertices A and B have consecutive interactions with each other in n timestamps, and at the $(n+1)$th timestamp the interaction is interrupted. In the meantime another interaction takes place between two other vertices C and D which had a weak interaction in the previous n timestamps. According to the restriction of maximum number of vertices in each cluster and based on the *EAC* algorithm, in case of edge deletion, the older edge (AB) should be deleted, in spite of the fact that this edge had a good history over time and there is a chance of repetition of this interaction in the future. Thus in this way the importance of the frequency of interactions over time (edges weight) is neglected. In order to further clarify the difference between *EAC* and *PCWGS* algorithms, we present another scenario.

Suppose the edges are formed in 3 timestamps. In Fig. 1, the edges are colored blue, orange, and green in the first, second and third timestamps, respectively. Edge weight represents the number of interactions between two vertices that occurred in each timestamps. If we consider the maximum number of vertices to be 5, in the second timestamp, the graph would have two separate clusters which are shown by the two clouds. However, in the third timestamp some interactions happen between the two clusters, based on their weights, which result in merging of the clusters. According to the constraint of maximum number of vertices in each cluster some edges must be deleted. As a result, a clustering based on the *EAC* algorithm yields three clusters *i.e.*, $\{A, C, D, G, E\}$, $\{F\}$ and $\{B\}$, which has overlooked the strong interactions in the three time-stamps and only considers the differences in the third timestamp. On the other hand, clustering based on the proposed *PCWGS* algorithm yields two clusters of $\{A, B\}$ and $\{C, D, F, E, G\}$ which preserves the differences in the third timestamp as well as the weight and interactions' repetition over the three timestamps these results are showing in Fig. 2. ❑

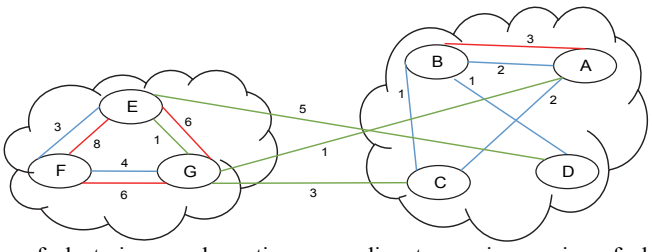
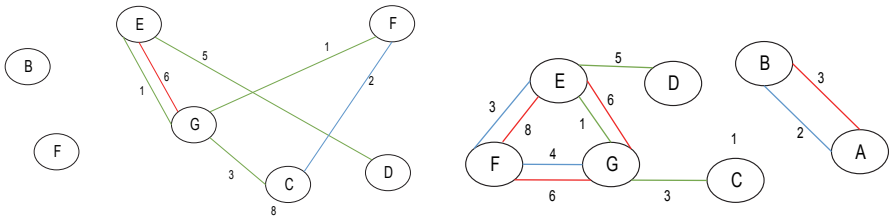


Fig. 1. The case of clustering graph vertices according to maximum size of clusters and edge weights in different time-stamps.



(a) *EAC* (b) *POWGS*

Fig. 2. The case of merging and splitting clusters according to maximum size of clusters and edge weights in different time-stamps with different algorithms; (a) Using the *EAC* algorithm and (b) using the *PCWGS* algorithm.

1.3 Our Contributions

Our main contribution in this paper is as follows:

- Considering number of recent communications as a parameter to identify clusters and their changes enables us to provide acceptable clustering using current status of graph without repeating calculations.
- A new data structure to capture useful information of edges in network based on weighted average and timestamp, with acceptable space requirement, enable us to manage the cost of complexity and storage of required information.
- Improvement of quality of clustering besides performance comparing the proposed method with ones like *EAC* and *EIC*.
- The same method for weighted and un-weighted networks are supported.

1.4 Related Works

There are two types of graph clustering algorithms, within-graph and between-graph clustering. In within-graph we attempt to cluster the nodes or edges of a single graph into groups based on a similarity measure, while in between-graph or object clustering we attempt to cluster a set of graphs that have arisen from a common set of nodes and are structurally similar. Object clustering has been studied by Aggarwal and Philip [1], which is based on structural similarity in a stream of large numbers of small graphs. In this paper, we focus on the node-clustering problem. Traditional node-clustering algorithms such as minimum cut problem and graph partitioning have been studied widely [13]. These algorithms cannot handle large and time evolving graphs where some clusters change rapidly at certain time points. Many of these algorithms are mainly functional in offline settings and therefore cannot be used efficiently in online or stream scenarios. In addition, they need to store the entire input before processing.

Granular computing [5, 8, 16, 19, 27] is a global computing model for managing big data, information and knowledge. In this model, each object is displayed as an information granule and can be linked together through some degree of similarity, functionality or indistinguishability. These connections can lead to a granular hierarchy or a network [15].

Granular computing has been commonly used for furthering other research areas,

such as data mining [15, 18] machine learning [26], computational intelligence [17], fuzzy rule-based systems [2], data classification [4] and bioinformatics [12]. It is of future research interest to apply granular computing techniques to solve risk assessment problems. On the other hand, it is worth noting that risk is highly related to uncertainty [10]. Uncertainty modeling is an important parameter in dynamic clustering where many elements of the clusters can change over time. Recently, Peters *et al.* [18] presented a method of guiding further developments of fuzzy clustering algorithms, especially dynamic fuzzy clustering algorithms. They combined granular computing with clustering analysis and used Dynamic Clustering Cubes (DCCs) to categorize existing dynamic granular clustering algorithms. Many of these algorithms are not provided for graph clustering in the stream model.

One of the widely used offline graph clustering algorithms is *METIS* [29], which is a type of multilevel graph partitioning algorithm with high-quality and balanced partitions. Kilot *et al.* [20] worked on graph partitioning in online and dynamic scenarios as well, but similarly they had to store the whole graph input prior to processing, which is not suitable for a streaming model. *FENNEL* is a one-pass graph partitioning algorithm for large scale graphs proposed by Charalampos *et al.* [22]. It uses modularity maximization [6] by a greedy assignment method. This algorithm works un-weighted graphs, and does not take into account the number of interactions between each pair of nodes. Zanghi *et al.* [29] proposed an online graph clustering algorithm that partitions online input of vertices by maximizing a global likelihood function. The algorithm is sensitive to the number of clusters and works well when the number of clusters is small. In addition, it is an insert-only model that does not support the deletion of vertices and edges. Eldawy *et al.* [9] proposed a method for clustering of streaming graphs by considering each connected component as a cluster according to the limited number of nodes in each cluster and updated these clusters incrementally based on the graph changes. Due to memory limitation, they used a random sampling method by assigning a random number to each edge and selecting the edges whose number was below a sampling threshold. Their algorithm is referred to as *EIC* and is not sensitive to cluster evolution [28]. In order to capture time evolving clusters, they solely used a sliding window approach in the streaming model. In [28] they proposed a new approach in order to process streaming graphs for evolution-aware clustering of vertices (*EAC*). They also considered individual connected components as clusters. Due to the capture time of evolving clusters, they assigned the timestamp of an edge as its weight. They also favored the more recent edges in a clustering and used the sliding window model to capture evolution. This method and most of the existing works only consider un-weighted edges in which the number of interactions between pairs of nodes over the time is ignored. However, in weighted graph stream models, it is important to consider the more recent interactions as well as their frequency, *i.e.* the edges with large number of interactions in recent time are much stronger than those with smaller number of interactions. Therefore, it is necessary to take into account the expiration time property of interactions to capture evolving clusters in streaming model [25]. The authors dealt with the problem of dynamic community detection, by considering the underlying social behavior over different graph regions. They designed a new structure called Local Weighted, Edge-based Pattern (*LWEP*) to describe homogeneous regions. They defined an exponential decaying weight for each edge to capture evolving cluster in the stream scenarios and used weighted Jaccard similarity to identify

communities. In their method the constant factor of the exponential function needs be estimated. It seems that the average number of interactions between two entities over time is an important parameter that is ignored in pervious approaches.

2. THE ALGORITHM

In this section we describe our clustering method in details and how it works in streaming scenario. The intuition behind our method is to consider both the weighted edges and the last occurrence of a connection (in terms of timestamp). Clearly in dynamic graph streams, changes in recent timestamps are more important than changes in older connections. But the caveat arises in weighted graph streams where we also need to take into account the weights of the connections. In order to capture the relationship between a pair of nodes, and the evolution in streaming graphs, we propose a method named *Priority-based Clustering in Weighted Graph Streams* (*PCWGS*). This algorithm, and in general any edge-comparison based methods for clustering dynamic graph streams, comprises of two main parts. First, a measurement criterion which preserves both the recent update time and the weight of a connection use it for edge comparisons. Second, a data structure which can handle efficient insertion/deletion of connections while decreasing the storage complexity and also keeps track of edges based on the measure provided in the first step.

We first start proposing our algorithm by introducing the incremental weighted graph streams, and then define the two parts of the *PCWGS* method.

Definition 1: The incremental representation of a graph stream is $G_1, G_2, \ldots, G_t, \ldots$ where $G_t = (V_t, E_t)$ is the incremental graph, V_t signifies the attached nodes until time t, $E_t = \{u_i, v_j, \lambda_{ij}^t \vee u_i, v_j \in V_1 \cup V_2 \cup \ldots \cup V_t\}$ with $\lambda_{ij}^t = \lambda_{ji}^t$ denoting the number of interactions between nodes u_i and v_j that happen between timestamps $t-1$ and t. ❑

We use the connected components as clusters and retain these clusters incrementally for graph updates. We use the combination of the time-stamp and weighted mean of number of interactions as an edge's weight. Edges in each cluster are stored in order of recent time-stamp and their mean of number of interactions that has happened recently.

For instance, assume each day as a time-stamp and assign the incremental linear weight i. C_i to time-stamp i such that $0 \leq i \leq n$ and C_i is a statistical factor that we will define later. Assigned weight and number of interactions at each time-stamp are shown in Table 1.

Definition 2: Weighed Mean: Assume that interactions between nodes u_i and v_j are recorded at timestamps $t_0 < t_1 < \ldots. < t_n$. t_0 is the timestamp at which u_i and v_j began to interact and $t_n = t$ is the current timestamp. $\bar{\lambda}_n$ in statistics, is the mean of interactions between u_i and v_j that has happened until t_n.

$$\bar{\lambda}_n = \sum_{i=1}^{n} iC_n\lambda_i \tag{2}$$

λ_i is the number of interactions between u_i and v_j that happens in time-stamp i and

C_n is a positive number such that:

$$\sum_{i=1}^{n} iC_n = 1 \rightarrow C_n = \frac{2}{n(n+1)}. \tag{3}$$

We define the expiration weight W_{ij}^t for an edge between two nodes u_i and v_j, as the following:

$$W_{ij}^t \approx t_n \cdot (\bar{\lambda_n}). \tag{4}$$

W_{ij}^t is a measure to determine the average number of interactions over time and according to importance of interactions that have occurred in recent timestamps. ❑

Theorem 1: At timestamp t_{n+1} for an existing edge with new interactions, its expiration weight W_{ij}^t is updated as:

$$W_{ij}^{t+1} = t_{n+1} \cdot \left(\frac{n}{n+2} \bar{\lambda}_n + \frac{2}{n+2} \lambda_{t+1}^{ij} \right), \tag{5}$$

$$\overline{\lambda_{n+1}} = \frac{n}{n+2} \bar{\lambda}_n + \frac{2}{n+2} \lambda_{n+1}^{ij}. \tag{6}$$

***Proof*:** We have

$$\bar{\lambda}_n = \sum_{i=1}^{n} iC_n \lambda_i, \tag{7}$$

$$C_n = \frac{2}{n(n+1)}, \tag{8}$$

$$C_{n+1} = \frac{2}{(n+1)(n+2)}. \tag{9}$$

According to Eqs. (6)-(9) the $\overline{\lambda_{n+1}}$ will be as:

$$\overline{\lambda_{n+1}} = C_{n+1} \sum_{i=1}^{n} i\lambda_i + (n+1) C_{n+1} \overline{\lambda_{n+1}}, \tag{10}$$

$$\sum_{i=1}^{n} i\lambda_i = \frac{\bar{\lambda}_n}{C_n}, \tag{11}$$

$$\overline{\lambda_{n+1}} = \frac{C_{n+1}}{C_n} \bar{\lambda}_n + (n+1)\lambda_{n+1}. \tag{12}$$

By substituting Eqs. (9) and (11) into Eq. (10), Eq. (13) would be resulted as follows:

$$\overline{\lambda_{n+1}} = \frac{n}{n+2} \bar{\lambda}_n + \frac{2}{n+2} \lambda_{n+1}^{ij}. \tag{13}$$

❑

2.1 Data Structure and Storage Minimization

As it is mentioned in the related works, the simple sliding window fails to handle the edge weights as a significant part of the information, effectively and decreasing the storage complexity is not a trivial task and is handled differently in different methods.

For this purpose we try to present a data structure which can keep useful information about the edges in the network based on update times and weights whilst keeping the storage as low as a constant factor.

The overall scheme of the algorithm consists of two parts which are the data handler and the graph handler. The data handler decides which information should be kept in the structure and which have to be omitted. It also manages the overall storage and prohibits it from exceeding a constant number of entries. The graph updates in different time-stamps are given to the data handler and upon each update, requests go back and forth between the data handler and the graph handler until the graph structure is stable. The data handler maintains the upper bound of the storage during this whole procedure.

The graph handler keeps the structure of the network and efficiently makes changes to it based on the requests from the data manager. It also creates a framework in which queries such as "which cluster does a particular node belong to?" or "which nodes are currently in a particular cluster?" can be answered efficiently.

Data Handler: The data handler consists of two smaller data structures of the type *Weighted Mean Priority Based Edge-Container*. One list is called the *main edge list* and the other one is called the *reserve edge list*. The *Weighted Mean Priority Based Edge-Container* provides access to a balanced binary search tree under the hood which keeps the edges in sorted order according to the value in Eq. (4) yields for each edge and allows efficient insertion/deletion. The balanced binary search tree embedded in the *Weighted Mean Priority Based Edge-Container* is allowed to use a constant amount of storage, that is if the number of elements in the *Weighted Mean Priority Based Edge-Container* is more than a constant factor, say k, The *Weighted Mean Priority Based Edge-Container* will report the fact that elements should be deleted form the back of the list (Lowest elements, since The *Weighted Variance Priority Based Edge-Container* keeps the elements sorted in non-increasing order) until the size of the balanced binary search tree that the data structure uses is less than or equal to the constant factor k.

The second list, the reserve edge-list similar in type to that of the main edge-list (*Weighted Variance Priority Based Edge-Container*), handles data in a manner identical to that of the main edge-list. The only difference between the two instances of the *Weighted Mean Priority Based Edge-Containers* is the constant factor k. The constant factor which denotes the maximum size of one list can differ from the other and therefore the two can vary independently. The purpose of having two different data structures with the exact same behavior will be discussed shortly.

Let M be the size of the *main edge-list* and N the size of the *reserve edge-list*. The graph handler component performs a clustering only based on M edges which are already kept in the *main list*. That is, the current state of the graph depends on the entries of the *main list*. A similar result can be obtained in the other direction too: at a particular time-stamp, the edges in the *reserve list* do not have any influence on clustering and structure.

It is also worthwhile to mention that the values in the two lists are pairwise distinct. That is if there is a connection record between two vertices A and B in the *main list*, there is certainly no sign of an entry representing the connection between A and B in the *reserve list*. The update queries are presented in the form of incremental graphs, each of which denotes a set of edges and their weights, and a timestamp which grows strictly increasing during different updates.

Although the entries of the two *Weighted Mean Priority Edge-Lists* and those of the incremental graph, are identically called edges, there is an important difference between them. An edge in the incremental graph denotes how close is the connection between two nodes between the arrival timestamp of this edge and the previous timestamp (or the initial connection weight between them in case of timestamp = 1). The entries in *Priority Based Edge Lists* however, represent how close is the connection between pairs of nodes in this timestamp, according to the entire data we have received. This means that not only the entries of the two Priority Lists are pairwise distinct, but also there is no duplicate of an edge in each of them either. That means no matter how many updates a pair of vertices has undergone previously, there is at most one record related to those two nodes in the lists. What actually makes this approach work is the pair of values (namely weighted mean and timestamps) which is stored (encapsulated) alongside the pair of vertices incident to each edge, and the formulas that are used to update them over time. Therefore the entry representing a pair of vertices (if it exists) in either of the Priority Based Lists is all the data we have stored for the connection between the pair of vertices. This property causes Priority Based Lists with maximum size of K to store much more information than the simple sliding windows of the same size, since there could be a lot of duplicate records of the connections between the same pair of vertices.

At each update if the connection between a pair of vertices has been remain strong enough based of measurement criteria, the related edge lies in the main list. Of course, when, the connection made weak, the edge lies in the reserve list, waiting for the future timestamp to get a chance to get into the main list, saving its previous history and be a part of the actual graph representation.

There is however a third case in which there is no present record of the connection between this pair of nodes which means that either the connection between the two has been so weak that they are popped off the reserve list as well as the main list, or this is the first time a connection between these two nodes is being supplied to the algorithm. Each time we try to do the insertion on an edge from the incremental graph, we give that edge a chance to be in the main list and therefore the edge is inserted into the graph, which in turn results in a change in how the graph is decomposed into clusters. After that, the cluster which the newly inserted edge is now part of it, must preserve the maximum cluster size constraint. If this constraint is violated, all of the edges of the cluster are erased and all of them except the one with the lowest weight among them are re-inserted into the graph using a different insertion algorithm, since they are not actual insertion queries and should not update Priority Lists. The edge with the lowest weight is deleted from the main list and sent to the reserve list, since its weight has not been significant, compared to other edges. After these insertions the cluster size constraint is checked again and in case of violation the same procedure takes place. After all the insertion queries are handled, the reserve list is possibly populated with some new edges and could be violating the size constraint. Therefore while the size of the list exceeds the maximum size of the reserve list, the lowest weighted edge is popped from the list, as a consequence of having a very low weight.

What explicitly happens is described in more detail below:

HandIncrementalGraph Algorithm: For each edge in the incremental graph do *IncrementalGraphEdgeInsertion*. It is shown in Algorithm 1.

IncrementalGraphEdgeInsertion Algorithm: when a tuple $e = \{a, b, w, t\}$ as an insertion query which states that a connection with weight w between nodes a and b has been recorded at timestamp t and the previous timestamp arrives, there are a number of cases:

Case 1: Edge $\{a, b\}$ is already in the main list: the weight of the edge is updated with respect to the formula in section 3 and no further action is needed and therefore the graph structure remains the same.

Case 2: Edge $\{a, b\}$ is in the reserve edge list: the weight of the edge is updated using the formula (4) and the edge is deleted from the reserve list and inserted into the main edge list and an insertion query *InsertIntoGraph* $\{a, b, w'\}$ where a and b are the nodes and w' is the new connection weight between a and b is sent to the graph handler component. At the end, deletion queries of the lowest-weighted edges in the main list are sent to the graph while the size of the main list exceeds M (Algorithms 5 and 6). Afterwards the deleted edges are in turn pushed into the reserve edge-list and entries are also deleted from the end of the reserve list while the list does not preserve the size constraint (N).The intuition behind this part of the algorithm is that whatever the connection weight is, an edge is given a chance to be in the main list and therefore in the graph, each time an insertion query arrives.

Case 3: Edge $\{a, b\}$ is in neither in the main or reserve edge-lists: this means that we have no data related to the edge $\{a, b\}$. This is either because of the fact that this is the first connection between these two edges or the previous connections have been significantly old and/or weak and was not eligible to be a part of the graph or being kept in the reserve-list. Therefore these edges are given weights using Eq. (4) and are again given a chance to be added to the actual graph structure as we take an approach similar to case 2 after we add the edge to the main list.

Deleting data from the reserve list, which means discarding any previous connections between two nodes, can seem unfair but it should be noted that as it can be inferred from the algorithm, if we discard the value of an edge, then the edge has been popped from the main list and the reserve edge list at least once. That means that the calculated weighted variance for this edge has been so low that even though it has been given a chance to be a part of the main list on each of its insertions, it has somehow been popped not only from the main list but also from the reserve list. (Since there have been connections with greater weight compared to $\{a, b\}$).

Graph Component Insertion: If a and b are not in the same cluster, their related clusters have to be merged. If the size of the cluster which contains a (or b) exceeds the cluster size limit, then delete the lowest weighted edge in the cluster using the deletion method in graph component. (Algorithms 4)

Graph Component Deletion:

- Insert the deleted edge into the reserve list.
- Delete all other edges in the cluster from the cluster, and the main list.
- Call the insert function each of the edges into the main list.

As depicted in the above pseudopods, the main algorithm is based on the four function. *EvaluateIncrementalGrpah* is the first one (Algorithm 1) which manages the graph updates in a specific timestamp. For each input edge *EdgeArrival*(*e*) function is called and then the length limitation of reserve list is verified. When the length is above the specified limit, edges with minimum weight are removed until the reserve list obey its length limitation. The complexity of this function is $O(e*O(EdgeArriavl) + e_n log(e_t))$. The number of input edges in timestamp t is shown by e and e_n is the number of edges which have to be removed from the reserve list because of its length limitation. The number of edges in the graph structure is shown by e_t and $\log(e_t)$ is the cost of deletion a node from the binary search tree.

Algorithm 1. EvaluateIncrementalGrpah

```
EvaluateIncrementalGrpah (G: Incremental Graph)
Foreach Edge e(a, b, w) in G:
     EdgeArrival(e)
While size of the reserve list violates the maximum size:
      Delete the lowest weighted edge from the reserve list
```

Algorithm 2. EdgeArrival

```
EdgeArrival(e (a, b, w): Edge):
If e is in the main list:
   Update the weight of the (a-b) entry in the main list using the formula e(4).
   Update the weight of the edge in the cluster which holds it.
Else if e is in the reserve list:
  Temp(a, b, w) ← (a-b) entry in the reserve list
  Delete (a-b) entry from the reserve list
  Update temp using the formula and e
  Insert temp into the main list
  NewEdgeUpdate(temp)
Else
  Insert e into the main list
  NewEdgeUpdate(e)
```

Algorithm 3. NewEdgeUpdate

```
NewEdgeUpdate(e(a, b, w):Edge):
cla ← cluster which contains a
clb ← cluster which contains b
Merge(cla, clb).
```

Algorithm 4. Merge

```
Merge(cla: Cluster, clb: Cluster):
Foreach Edge e in clb:
   Insert e into cla
   Remove e from clb
HandleOversizedCluster(cla)
```

Algorithm 5. HandleOversizedCluster

```
HandleOversizedCluster(cla:Cluster):
  If size of the nodes in cla violates the maximum size
      Delete the lowest weighted edge from cla and from the main list
      Remove cla, and assign all the nodes previously contained in cla to their initial clusters
  Foreach Edge e(a, b, w) that was previously in cla:
     NoUpdateInsert(e)
```

Algorithm 6. NoUpdateInsert

```
NoUpdateInsert(e(a, b, w): Edge):
cla ← cluster which contains a
clb ← cluster which contains b
Insert e into cla
If cla != clb:
   Merge(cla, clb)
```

In Algorithm 2, if the input edge exists in the main list, its weight is updated and the graph structure remains unchanged. If the input edge is in the reserve list it is removed from the reserve list and its weight is updated. Then this edge is transferred to the main list and the graph structure. When the input edges is new and it is not exists in both of lists, *NewEdgeUpdate* is called. In the worst case when the entire input nodes are new edges the complexity of Algorithm 2 is *e***O*(*NewEdgeUpdate*) which *e* is the number of input edges.

The two clusters which have vertex u and vertex v s are identified in *NewEdgeUpdate* (Algorithm 3) and then the above selected clusters are merged using *Merge* function (Algorithm 4). If the size of newly merged cluster is above the specified limit, it is managed by *HandleOversizedCluster* function. $e_{c(v)}*(e_{c(u)})+O(HandleOversizedCluster)$ is the complexity of *Merge* function.

As mentioned above oversized clusters are pruned in *HandleOversizedCluster* function (Algorithm 5) by selecting edge with minimum weight is removed from the cluster and the main list and is inserted into the reserve list. If such an edge has less weight than the entire reserve list entries then it is directly deleted. When an edge is removed, its corresponded cluster may be divided to the two new smaller clusters.

In order to preserve the graph connected, like *EAC* method, first the entire cluster edges are removed, then the required ones are inserted again. Considering that there is no need to modify main and reserve lists structure and the cluster structure has to be updated, we proposed to insert edges of cluster $C_{(v)}$ using *NoUpdateInsert* function. Therefore, the complexity of *HandleOversizedCluster* function is$e_{c(v)}*O(NoUpdateInsert)$. It ios worth noting that *NoUpdateInsert* function as the last step, merges two clusters corresponding to vertices u and v using *Merge* function.

3. EXPERIMENTAL RESULTS

In this section we compare the results of the proposed *PCWGS* algorithm with the *EAC* algorithm on the four aforementioned real data sets using three measures. First, using the Normalized Mutual Information (*NMI*) to assess the quality and similarity in clustering of the two algorithms. The second measure is the cut size in the clustering process and third measure involves the comparison of execution time of the two algorithms. In addition, to gain a better understanding of the evolutionary clustering process. The procedure is implemented on a system with a 2.0 GHz Core 2 duo CPU, with 4GB of RAM and operating under Windows 8 64bit. We also took advantage of Java as the programming language.

3.1 Data Sets and Experimental Settings

Four real data sets are used in our experimental evaluation algorithms. A summary of the characteristics of the four data sets is shown in Table 2.

Table 2. Summary of the test datasets: the overall number of nodes, edges, average degree and metadata are listed.

Data set	# of Nodes	# of Edges	Average degree	Metadata
Dutch college	32	3,062	191.38 edges / vertex	Timestamps
Haggle	274	28,244	206.16 edges / vertex	Timestamps
Infectious	410	17,298	84.380 edges / vertex	Timestamps
Facebook friendships	63,731	817,035	25.640 edges / vertex	Timestamps

Dutch College Data Set [30]: This directed network contains friendship ratings between 32 university freshmen who mostly did not know each other prior to starting university. Each student was asked to rate the other student at seven different time points. We note that the origin of the timestamps is not accurately known but the distance between two timestamps is given. A node represents a student and an edge between two students shows that the left rated the right one. The edge weights show how good their friendship is in the eye of the left node. The weight ranges from −1 (showing a risk of getting into conflict) to +3 (showing a very close relationship).

Haggle Data Set [32]: This undirected network represents contacts between people measured by carried wireless devices. A node represents a person, and an edge between two persons shows that there was a contact between them.

Infectious Data Set [33]: This network describes the face-to-face behavior of people during the exhibition INFECTIOUS: STAY AWAY in 2009 at the Science Gallery in Dublin. Nodes represent exhibition visitors; edges represent face-to-face contacts that were active for at least 20 seconds. Multiple edges between two nodes are possible and denote multiple contacts. The network contains the data from the day with the most interactions.

Facebook Friendships Data Set [31]: This undirected network contains friendship data of *Facebook* users. A node represents a user and an edge represents a friendship between two users.

We designed three kinds of experiments:

Quality: Using the cut-size quality measure, we show that our algorithm gives reasonable results compared to the *EAC* Algorithm, which is assumed to be close to the best previous state-of-the-art clustering algorithms.

Performance: We test the performance of each approach in terms of average runtime. We show that our approach has better performance than the *EAC* approach.

NMI: Normalized mutual information is one of the most widely used measures of clustering quality [21].

Given a data set D of size n, the clustering labels β of C clusters and the other clustering labels α of C' clusters. A confusion matrix is built where entry (i, j) defines the number $n_i^{(j)}$ of points in the ith cluster of β and the jth cluster of α. then NMI can be computed from the confusion matrix [25]:

$$NMI = \frac{2\sum_{l=1}^{C}\sum_{h=1}^{C'}\frac{n_l^{(h)}}{n}\log\frac{n_l^{(h)}n}{\sum_{i=1}^{C}n_i^{(h)}\sum_{i=1}^{C'}n_i^{(i)}}}{H(\beta)+H(\alpha)}. \tag{14}$$

Where $H(\beta) = -\sum_{i=1}^{C}\frac{n_i}{n}\log\frac{n_i}{n}$ and $H(\alpha) = -\sum_{j=1}^{C'}\frac{n^{(j)}}{n}\log\frac{n^{(j)}}{n}$ are the Shannon entropy of

cluster labels β and α, respectively, with $n^{(j)}$ denoting the number of points in the ith cluster of β and in the jth cluster of α, respectively. A high *NMI* value indicates that the two clustering algorithms match well.

3.2 Weighted Version vs. Un-weighted Version

We compared the proposed algorithm with the *EAC* algorithm because *EAC* has proven to be more efficient in online and stream model [28], compared to algorithms such as *EIC* and extended *METIS* in terms of clustering quality and efficiency. Thus we implemented the *EAC* algorithm in a similar condition and environment in order to perform the evaluation and comparison. The *NMI* measure, which is a similarity measure between two clustering methods, is obtained in four time intervals. In these networks due to large number of timestamps, the edges' input time is divided into five time intervals and for each interval the *NMI* is calculated. In addition the Dutch College dataset has only 7 timestamps, which are considered for the calculation of *NMI*. As the proposed algorithm can be used both for weighted and un-weighted graphs, we first considered the un-weighted networks and regardless of the frequency of edges over time, which showed that both algorithms have a similar performance. This is because we used *EAC* as the basis of the proposed *PCWGS* algorithm. On the other hand if we take into account edge weights, frequency of edges over time, and significance of edges in recent time intervals, as important parameters in clustering, the results show that the algorithms perform differently. Moreover these parameters maintain the evolutionary process and strong interactions relatively well. In Fig. 3 the *NMI* comparison results are illustrated for both algorithms over four networks, for both weighted and un-weighted cases.

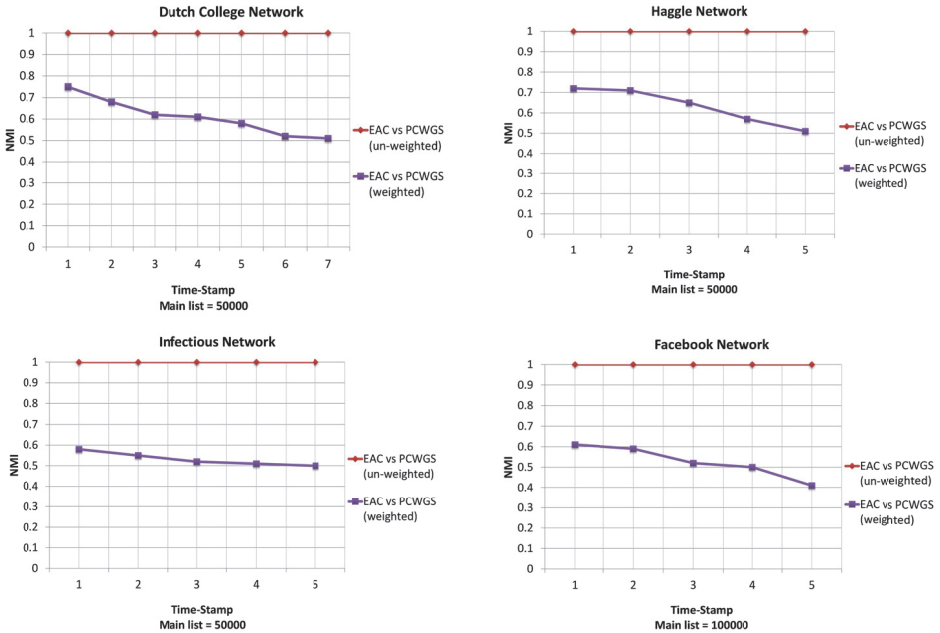


Fig. 3. Comparison of the *NMI* values for weighted and un-weighted *PCWGS* algorithm with different time-stamp and the *EAC* algorithm.

3.3 Quality and Performance

Another measure that is employed to compare the clustering quality of the proposed method and the *EAC* method is the cut size. Due to the difference in weight assigning parameters in the two algorithms, the number of edges in the cut is used instead of the size of cut weight. This measure is calculated for the maximum number of distinct vertices in each cluster. The results show that the proposed algorithm has a better cut size compared to the *EAC* algorithm. In Fig. 4 the results of the clustering quality based on this measure are illustrated for four data sets. Consequently it shows an average improvement rate of 1.72 for the Haggle network, 1.34 for the Dutch College network, 1.43 for the Infectious network, and 1.12 for the Facebook network. These results are shown in Fig. 4.

The next comparison measure is the execution time of the algorithm. The experimental results show that the proposed algorithm performs more efficiently than the *EAC* algorithm in identical conditions. As it was mentioned, in the *PCWGS* algorithm the balanced binary tree data structure is used to store edges and their related information and in the case of re-insertion of an edge from a cluster and deleting another edge in the same cluster the *NoUpdateInsert* algorithm, which was discussed in the previous section, is used. On the contrary, the *EAC* algorithm does not use this data structure and employs the ordinary insertion algorithm. In Fig. 5 the execution time of these two algorithms over four data sets with different maximum clustering size and maximum main list size (The maximum size of the window in *EAC*) is illustrated.

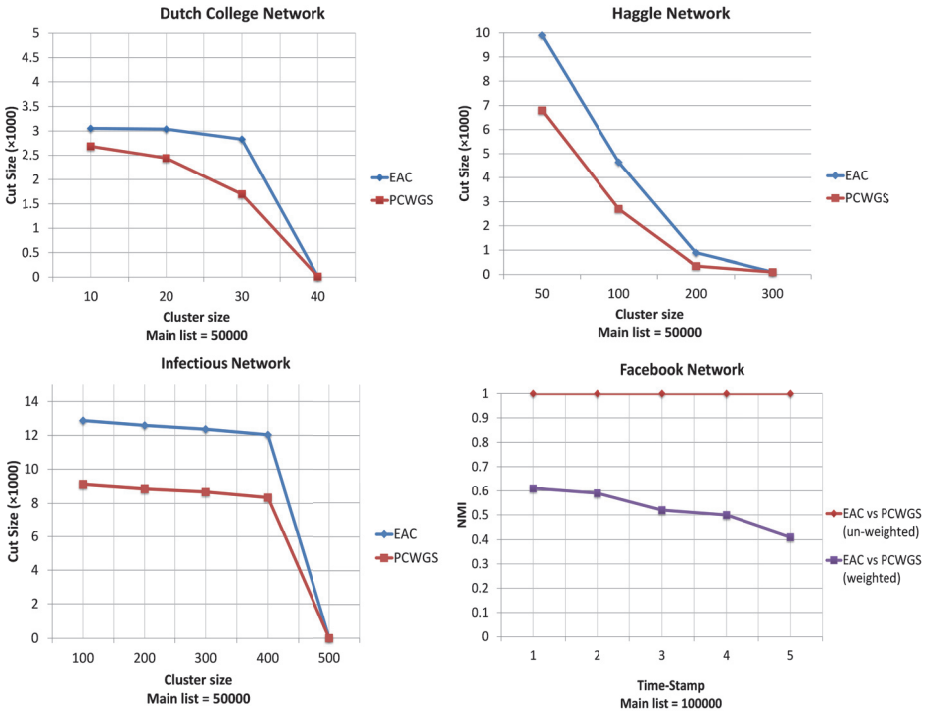


Fig. 4. Cut size experiments with different maximum cluster sizes.

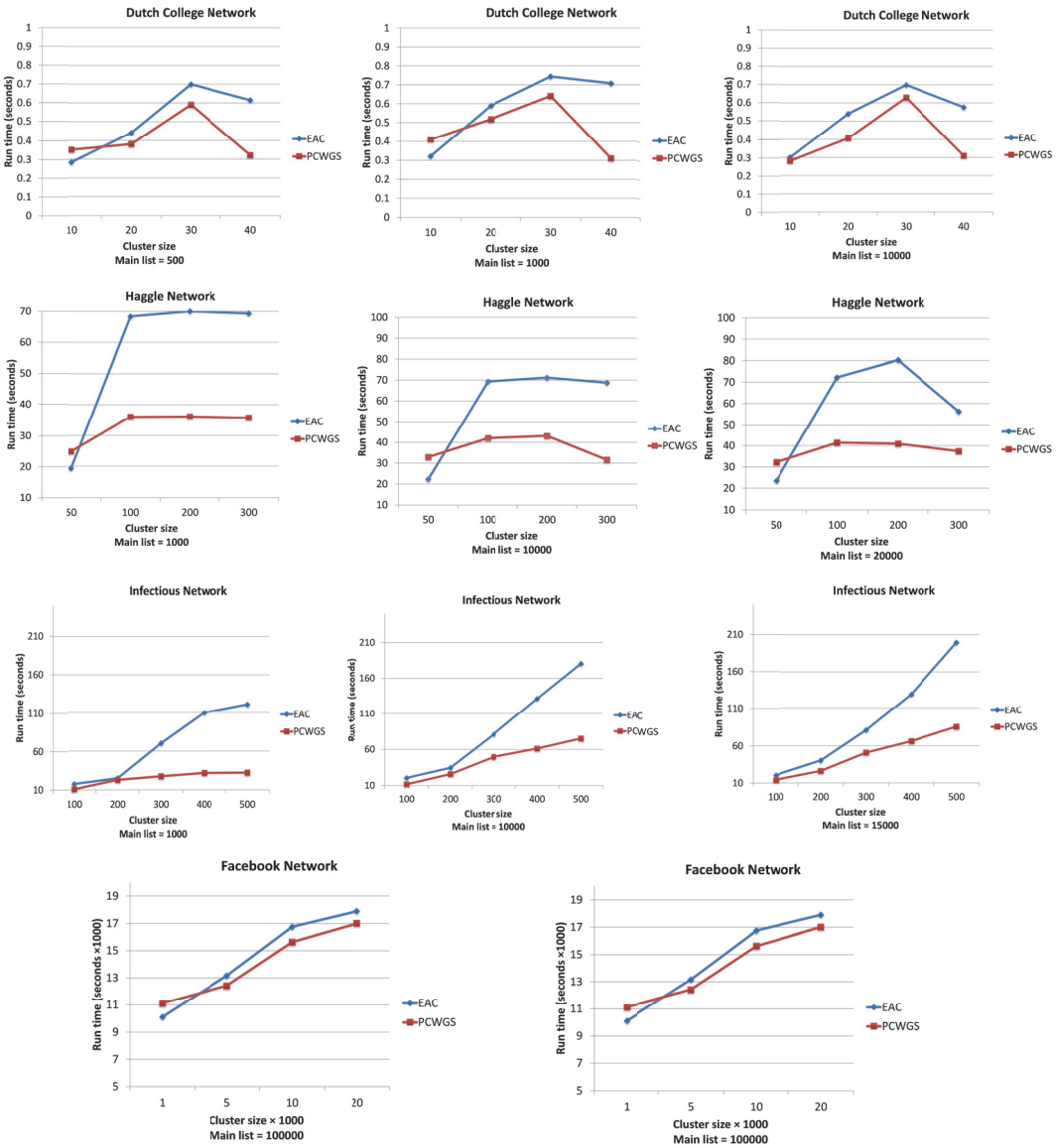


Fig. 5. Comparing the runtime of the *PCWGS* and *EAC* algorithms with different maximum cluster size and maximum main list size (sliding window size).

4. CONCLUSIONS

In this paper we presented a data structure and a measure based on the frequency of recent interactions in the network and access to the occurred clustering changes over time. Our *PCWGS* algorithm is able to cluster the network at any moment via the network state at a previous moment, without re-initialization of the network calculations.

According to the streaming model this property has caused the algorithm to incrementally cluster weighted and un-weighted graphs and to yield satisfactory results. Moreover, the obtained results from different data sets illustrated how algorithms can behave differently in terms of accessing changes and evolutionary trends. The *EAC* algorithm has proven more efficient in clustering with respect to other algorithms. Thus we compared our *PCWGS* algorithm with the *EAC* algorithm. For un-weighted networks in which the number of interactions is not taken into account, the *PCWGS* behaves similar to the *EAC* algorithm. However, if the network is weighted and other parameters such as number of interactions over time is considered, the results of two algorithms differ and *PCWGS* will have a higher quality than the *EAC* in terms of cut size. Moreover, comparing the execution time of both algorithms we found that with regard to network type, the *PCWGS* algorithm has a relatively lower execution time.

As a future direction, the *PCWGS* algorithm can be implemented with parallel capabilities in distributed environments in order to improve the execution time and the power of the algorithm. Furthermore, in order to gain a higher quality in clustering process, other measures that can identify graph clusters with respect to the frequency of recent interactions can be explored. Also an area of future interest is to analysis of granular computing for clustering dynamic networks in a given stream model that has not been used yet.

REFERENCES

1. C. C. Aggarwal, Y. Zhao, and P. S. Yu, "On clustering graph streams," in *Proceedings of SIAM International Conference on Data Mining*, 2010, pp. 478-489.
2. S. S. Ahmad and W. Pedrycz, "The development of granular rule-based systems: a study in structural model compression," *Granular Computing*, Vol. 2, 2017, pp. 1-12.
3. K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: Sparsification, spanners, and subgraphs," in *Proceedings of SIGMOD Symposium on Principles of Database Systems*, Vol. 2012, 2012, pp. 5-14.
4. M. Antonelli, P. Ducange, B. Lazzerini, and F. Marcelloni, "Multi-objective evolutionary design of granular rule-based classifiers," *Granular Computing*, Vol. 1, 2016, pp. 37-58.
5. D. Ciucci, "Orthopairs and granular computing," *Granular Computing*, Vol. 1, 2016, pp. 159-170.
6. A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, *Statistical Physics*, *Plasmas*, *Fluids*, *and Related Interdisciplinary Topics*, Vol. 70, 2004, p. 66111.
7. M. S. Crouch, A. McGregor, and D. Stubbs, "Dynamic graphs in the sliding-window model," in *Proceedings of European Symposium on Algorithms*, LNCS, Vol. 8125, 2013, pp. 337-348.
8. D. Dubois and H. Prade, "Bridging gaps between several forms of granular computing," *Granular Computing*, Vol. 1, 2016, pp. 115-126.
9. A. Eldawy, R. Khandekar, and K.-L. Wu, "Clustering streaming graphs," in *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems*, Vol. 1, 2012, pp. 466-475.

10. X. Dong, H. Lu, X. Yuanpu, and X. Ziming, "Decision-making model under risk assessment based on entropy," Entropy 18.11 404, 2016.
11. T. Hartmann, A. Kappes, and D. Wagner, "Clustering evolving networks," arXiv Preprint, 2014.
12. D. Guzenko and S. V. Strelkov, "Granular clustering of de novo protein models," *Bioinformatics*, Vol. 33, 2017, pp. 390-396.
13. D. R. Karger, "Random sampling in cut, flow, and network design problems," in *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, Vol. 24, 1994, pp. 648-657.
14. G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, Vol. 20, 1998, pp. 359-392.
15. P. Lingras, F. Haider, and M. Triff, "Granular meta-clustering based on hierarchical, network, and temporal connections," *Granular Computing*, Vol. 1, 2016, pp. 71-92.
16. H. Liu, A. Gegov, and M. Cocea, "Rule-based systems: A granular computing perspective," *Granular Computing*, Vol. 1, 2016, pp. 259-274.
17. L. Livi and A. Sadeghian, "Granular computing, computational intelligence, and the analysis of non-geometric input spaces," *Granular Computing*, Vol. 1, 2016, pp. 13-20.
18. G. Peters and R. Weber, "DCC: a framework for dynamic granular clustering," *Granular Computing*, Vol. 1, 2016, pp. 1-11.
19. A. Skowron, A. Jankowski, and S. Dutta, "Interactive granular computing," *Granular Computing*, Vol. 1, 2016, pp. 95-113.
20. I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2012, p. 1222.
21. A. Strehl and J. Ghosh, "Cluster ensembles – A knowledge reuse framework for combining multiple partitions," *Journal of Machine Learning Research*, Vol. 3, 2002, pp. 583-617.
22. C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs categories and subject descriptors," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, 2014, pp. 333-342.
23. J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, Vol. 11, 1985, pp. 37-57.
24. N. Wale, X. Ning, and G. Karypis, *Managing and Mining Graph Data*, Springer, Berlin, 2010.
25. C.-D. Wang, J.-H. Lai, and P. S. Yu, "Dynamic community detection in weighted graph streams," in *Proceedings of SIAM International Conference on Data Mining*, 2013, pp. 151-161.
26. G. Wilke and E. Portmann, "Granular computing as a basis of human-data interaction: a cognitive cities use case," *Granular Computing*, Vol. 1, 2016, pp. 181-197.
27. Y. Yao, "A triarchic theory of granular computing," *Granular Computing*, Vol. 1, 2016, pp. 145-157.

28. M. Yuan, K.-L. Wu, G. Jacques-Silva, and Y. Lu, "Efficient processing of streaming graphs for evolution-aware clustering," in *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, 2013, pp. 319-328.
29. H. Zanghi, C. Ambroise, and V. Miele, "Fast online graph clustering via Erd\Hos–Rényi mixture," *Pattern Recognition*, Vol. 41, 2008, pp. 3592-3599.
30. Dutch college network dataset – KONECT.76, http://konect.uni-koblenz.de/networks/moreno_vdb, 2015.
31. Facebook friendships network dataset – KONECT.76, http://konect.uni-koblenz.de/networks/facebook-wosn-links, 2015.
32. Haggle network dataset – KONECT. 77, http://konect.uni-koblenz.de/networks/contact, 2015.
33. Infectious network dataset – KONECT.76, http://konect.uni-koblenz.de/networks/sociopatterns-infectious, 2015.

Mohsen Saadatpour received the B.Sc. degree in Computer Science from Amirkabir University of Technology (Tehran Polytechnic), and received the M.Sc. degree in Computer Science from Shahid Beheshti University of Technology, Iran. His research interests include social networks, communication networks, data stream, data mining, graph mining.

Sayyed Kamyar Izadi is an Associate Professor at Alzahra University. He obtained his Ph.D. degree from Iran University of Science and Technology in 2011. He received his B.Sc. degree from the Isfahan University of Technology in 2001. He finished his M.Sc. degree at the Iran University of Science and Technology in 2003. He joined the DBIS research group lead by Prof. Härder and participated in the XTC project (a native XML database management system) for one year to the end of June 2008.

Mohammad Nasirifar is a B.Sc. in Computer Science student at Shahid Beheshti University. His research interests are in the areas of machine learning, robotics, and control theory.

Hamed Kavousi is a B.Sc. in Computer Science student at Shahid Beheshti University. His research interests are in the areas of computer vision, machine learning, data mining.