A Bounded-size Clustering Algorithm on Fully-dynamic Streaming Graphs

Article *in* Intelligent Data Analysis · December 2018

DOI: 10.3233/IDA-173573

CITATION
1

READS
191

4 authors:

Jianpeng Zhang
Eindhoven University of Technology
28 PUBLICATIONS **65** CITATIONS
SEE PROFILE

Yulong Pei
Eindhoven University of Technology
28 PUBLICATIONS **114** CITATIONS
SEE PROFILE

George H. L. Fletcher
Eindhoven University of Technology
91 PUBLICATIONS **928** CITATIONS
SEE PROFILE

Mykola Pechenizkiy
Eindhoven University of Technology
226 PUBLICATIONS **4,817** CITATIONS
SEE PROFILE

A Bounded-size Clustering Algorithm on Fully-dynamic Streaming Graphs

Jianpeng Zhang[1,2], Yulong Pei[1],
George Fletcher[1], and Mykola Pechenizkiy[1]

[1] Eindhoven University of Technology, 5600 MB Eindhoven, the Netherlands
[2] National Digital Switching System Engineering & Technological R&D Center, Zhengzhou, China
Email:{j.zhang.4,y.pei.1,g.h.l.fletcher,m.pechenizkiy}@tue.nl

Abstract. Many contemporary data sources in a variety of domains can naturally be represented as fully-dynamic streaming graphs. How to design an efficient online streaming clustering algorithm on such graphs is of great concern. However, existing clustering approaches are inappropriate for this specific task because: (1) static clustering approaches require expensive computational cost to cluster the graph for each update and (2) the existing streaming clustering neither could fully support insertion/deletion of edges nor take temporal information into account. To tackle these issues, in this work, firstly we propose an appropriate streaming clustering model and design two new core components: *streaming reservoir* and *cluster manager*. Then we present an evolution-aware bounded-size clustering algorithm to handle the edge additions/deletions. It requires the clusters to satisfy the maximum cluster-size constraint, and maintains the recency of edges in the temporal sequence and gives high priority to the recent edges in each cluster. The experimental results show that the proposed *BSC* algorithm outperforms current online algorithms and is capable to keep track of the evolution of graphs. Furthermore, it obtains almost one order of magnitude higher throughput than the state-of-the-art algorithms.

Keywords: Streaming graph; clustering algorithm; streaming reservoir; cluster manager

1 Introduction

With the rapid development of information technology, the amount of network-structured data sets in various applications are growing explosively, and the relationships between various entities are becoming increasingly large and complex. Graph, as a generic data structure, can be a good representation of the complex relationships among entities of networks [10]. In practice, many real-world graphs are fully-dynamic where the relationships among entities change along with time rapidly such as social and web networks. In such dynamic scenario, they can be represented as streaming graphs in which edges/vertices insertions

and deletions occurring in an arbitrary (even adversarial) order [19][3]. To understand the structures of graph and capture the rapid change in real-time, effective and efficient online graph clustering methods are of great demand.

Graph clustering [5][6][17] is one of the most important issues in graph mining. Its aim is to group the vertices into clusters with dense intra-cluster link and sparse inter-cluster connectivity. There are numerous studies on graph clustering in the literature but two challenges exist in previous studies. The first challenge of this work is that most networks in practice are large-scale and in a streaming fashion. These real graphs might contain millions or even billions of vertices/edges and fully-dynamic, but most of existing algorithms mainly focus on clustering in an offline setting on the premise that the entire graph is given beforehand [13][14]. Several types of static clustering algorithms ,e.g., overlapping [6], non-overlapping [7], hierarchical [20], multi-level [17] clustering have been studied widely. However, current offline methods are not suitable for the fully-dynamic streaming graphs which involve the additions and deletions of vertices/edges, because they require to re-cluster the entire graph for each update which leads to the expensive computational cost. Furthermore, the newly obtained clusters may significantly differ from the preceding ones, but they are supposed to be smooth with the previous clusters, as offline methods can not react in a continuous way to the smooth changes in the graph. Another challenge is that existing streaming clustering algorithms do not fully support insertion/deletion of edges while taking temporal priority into account. Incorporating the ability to delete edges is important in a fully-dynamic streaming setting, for example when clustering is performed over a sliding window (i.e., edges are deleted from the tail end of the sliding window). As graphs encountered in real applications are often fully-dynamic, temporal priority is of crucial importance to capture the formation of clustering structures and track the cluster changes in real time. If the method does not consider the importance of the arrival sequence, it is not capable to be sensitive to evolving events. The challenge is how to incrementally capture the clustering evolution as it happens. The intuition is to use edge recency as a measure of weight to study the process of graph clusters' evolution.

Addressing the above issues, the streaming version of clustering model is preferred for such kind of graphs, and it should process a stream of edge insertions/deletions in an incremental manner, where each edge can be processed only once, and extremely fast under the limited memory [4][11][23]. To satisfy the memory limit, the common practice is to use the specific time-window to control the size of streams and fix it into a manageable size and give approximate results. Meanwhile, the model should support both static large-scale graphs (stored as a list of edges streaming from storage) and streaming graphs. Thus, the clustering algorithm designed to process fully-dynamic streams is also applicable for static large-scale graphs.

In this paper, we propose a streaming clustering model and devise an incremental clustering algorithm which is suitable for fully-dynamic streaming setting where both edge additions and deletions are allowed. It is capable of maintain-

ing a high quality clustering and also can capture the evolving events (namely, evolution-aware) while updates are performed on the fly. Specifically, our contributions can be summarized as follows:

1) We propose an appropriate streaming clustering model and design two new core components: *streaming reservoir* and *cluster manager*. *Streaming reservoir* is designed to keep the sampled graph (i.e., a set of finite non-empty clusters) up-to-date in real time and make it satisfy two properties: conformity and maximality. Meanwhile, *cluster manager* creates a new augmented *union-find* data structure to store, find, merge and remove clusters efficiently.
2) We present an evolution-aware bounded-size clustering algorithm for fully-dynamic streaming graphs in which edge insertions and deletions are allowed under specific time-window settings (i.e., to satisfy the memory limitation). It treats *individual connected components* as clusters subject to a constraint on the maximum cluster-size. Furthermore, it keeps the recency of edges in the temporal sequence and gives high priority to the recent edges in each cluster of the sampled graph such that it is capable to capture the clustering evolution on the fly.
3) We conduct quality and throughput experiments on synthetic and real-world networks. The results show the proposed *BSC* algorithm outperforms current online clustering algorithms in terms of the clustering quality. In addition, our algorithm performs almost one order of magnitude higher throughput than the state-of-the-art algorithms.

The remainder of the paper is organized as follows. We give the problem definition in Section 2, and then the existing work of streaming graph clustering are provided in Section 3 . In Section 4 we outline the proposed streaming clustering model, and provide a detailed description of the associated update operation. Experimental evaluation on both synthetic and real-world networks is given in Section 5. The paper is concluded and future work is presented in Section 6.

2 Basic Notation & Problem Statement

We focus on the problem of bounded-size clustering in a fully-dynamic streaming graph where edge insertions and deletions are allowed. Khandekar, et al. [15] have proved that the BSC (bounded-size clustering) problem is NP-hard. Thus it motivate us to develop a heuristic approximation method for this problem. Firstly, we give the definition of fully-dynamic streaming graph. Formally, for any discrete time-stamp $t \geq 0$, consider an undirected graph $G_t = (V_t, E_t)$, where V_t is a finite set of vertices and $E_t \subseteq V_t \times V_t \times \mathbb{R}^+$ is a finite set of edges. Here, $\mathbb{R}^+$ denotes the set of positive real numbers. Each edge e_t is in the form of $\langle u, v, w \rangle$, where u and v are the two vertices of the edge and w the associated temporal weight. Note that the temporal weight can be defined as the arrival sequence, the relative time and even the fading time function accordingly to the situation. Initially at time-stamp $t = 0$ we have $V_t = E_t = \emptyset$, and for any $t > 0$,

at time-stamp $t+1$ we receive a new update $e_{t+1} = (\bullet, \langle u, v, w\rangle)$ from the edge streams, where $\bullet \in \{insertion, deletion\}$. The graph $G_{t+1} = (V_{t+1}, E_{t+1})$ at time-stamp $t+1$ can be updated as follows:

$$E_{t+1} = \begin{cases} E_t \cup \langle u, v, w\rangle & \text{if } \bullet = \text{“insertion”} \\ E_t \setminus \langle u, v, w\rangle & \text{if } \bullet = \text{“deletion”} \end{cases} \tag{1}$$

For simplicity, we do not allow the same edge to appear multiple times in the streams and do not execute deletion if the edge does not exist in the model.

Secondly, the bounded-size clustering problem we address is to partition the vertices into clusters of size at most a given budget and minimize the total edge weights across the clusters on the fully-dynamic streaming graphs. There exists a constraint on the maximum number of vertices in each cluster denoted by B. The goal is to partition the vertices V_t at time-stamp t into a set of connected components (namely, clusters) $\Omega = \{C_1(t), ..., C_k(t)\}$ where each $C_i(t)$ ($i \in [1, k]$) is a cluster such that

- the size of each cluster is bounded: $|C_i(t)| \leq B$ for all i ($i \in [1, k]$);
- the total edge-weight across the clusters is minimized and the sum weight of the intra-clusters is maximized under the maximum cluster-size constraint:

$$\min \sum_{C \in \Omega} \sum_{\substack{u \in C, v \notin C, \\ \langle u,v,w\rangle \in E_t}} w \tag{2}$$

$$\max \sum_{C \in \Omega} \sum_{\substack{u \in C, v \in C, \\ \langle u,v,w\rangle \in E_t}} w \tag{3}$$

- each cluster contains at most one individual connected component, i.e., there is a path from u to v for every pair of vertices u and v in the cluster.

Concretely, we can see that these three restrictions make a clustering satisfy: (i) the first constraint (i.e., conformity), which means each cluster should satisfy the maximum cluster-size constraint; (ii) the second constraint (i.e., maximality), which means the sum weight of the intra-cluster of each cluster is maximized and no more edges can be added to each cluster without violating the constraint; and (iii) each cluster contains at most one connected component.

3 Background & Related Work

Graph clustering is a fundamental optimization problem with applications to a variety of areas like recommendation systems, image segmentation, online marketing analysis, discovering communities in social networks, etc. Most of contemporary algorithms are suitable for an offline setting in which the entire graph is given beforehand. The most well-known offline clustering/partitioning algorithm is *METIS* [14]. It is based on the multilevel graph partitioning paradigm and is shown to produce high-quality (balanced) partitions. However, when used

in streaming graph, it is generally inefficient since offline algorithms need to re-cluster all the edges for each new update. For the streaming setting, there exist two research baselines to handle streaming graph which are denominated as *balanced k partitioning* [22][21][25][18] and *bounded-size clustering* [9][15] problems, respectively. They look similar but turn out to have different emphasis. The *balanced k partitioning* assumes that the number of clusters is an input parameter, and seeks to minimize the number of edge cut such that the partitions have nearly equal size. Its aim is not network analysis but the preprocessing of graphs for parallel computing tasks. It is not our focus of this work and the reader can refer to [22] for more detail.

Unlike *balanced k partitioning* problem which needs prior knowledge of cluster number and balances each cluster evenly, the *bounded-size clustering* problem uses a different cluster constraint, i.e., a maximum cluster-size as an input and the balance constraint is removed. [29] solved the graph clustering problem in an online fashion using an Erdős–Rényi mixture model, but it does not allow deletion or modification of the graph. Also it does not scale well with the number of clusters. Aggarwal et al. [1] designed the hash-compressed micro-clusters and found structurally similar graphs in a stream of large number of small graphs. However, their algorithm does not deal with edge deletions, making it not applicable to cluster streaming graphs in the face of sliding windows. [2] utilized constraint reservoir sampling and proposed an outlier-detection *GOutlier* algorithm for detecting temporal outliers in graph streams. Their scheme is based on finding multiple vertex clusterings in a streaming graph and only supports edge additions. Besides, this algorithm is likely to yield a clustering with many small clusters. In [26] the algorithm was used for dynamic clustering in weighted graph streams. A local weighted-edge-based structure is devised to describe a local homogeneous region and is computed efficiently by maintaining top-k neighbor lists and top-k candidate lists. In [27], a hypergraph clustering algorithm was proposed in an online fashion, but the inconsistency issue of online sparsification needs to be soloved. The *structural-sampler* algorithm proposed in [9] samples over the edges in the random order and discards the edges exceeding a sample threshold p, and then maintains a reservoir sample of the edges. One downside to this algorithm is that it does not consider the importance of the arrival sequence and is not sensitive to evolving graphs. Another one is once the maximum cluster-size constraint is violated, unwarranted deletions might occur to clusters whose sizes actually do not exceed the constraint and the mistaken deleted edges need to be reinserted to their corresponding clusters, and it is a time-consuming process. Subsequently, [28] described an efficient evolution-aware clustering (EAC) approach. This algorithm is sensitive to the clustering evolution by giving proper weights to the more recent edges in the time window, but when it decides to delete the edges because of constraint violation, the edges are not considered to be recycled. Without the edge-recycling mechanism, it makes the clustering structure loss of important links and no longer satisfy the maximality criterion.

To the best of our knowledge, there are no recent methods that could track the changes of clusters in real time and make it always satisfy maximality and conformity criterions [9] to handle fully-dynamic streaming graph (i.e., a sequence of edges additions and deletions in arbitrary order). Therefore, we design an appropriate streaming clustering model and propose two core components: *streaming reservoir* and *cluster manager*, and present a new bounded-size clustering algorithm to solve those problems. This approach allows for both edge additions and deletions, and is capable to capture the clustering evolution on the fly.

4 The Evolution-aware Bounded-size Clustering

In this section, first we generalize the streaming clustering model and describe the details of core components. Then, we will describe the new evolution-aware bounded-size clustering (BSC) algorithm.

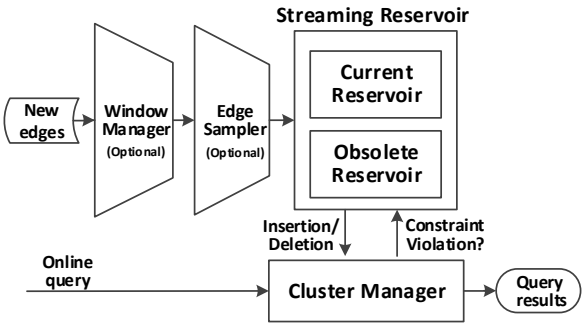


Fig. 1. Main components of the streaming clustering model.

4.1 Streaming Clustering Model

As shown in Fig. 1, the streaming clustering model consists of four components including: *window manager*, *edge sampler*, *streaming reservoir* and *cluster manager*.

Window manager is a vital component when one needs to maintain a graph over a certain time window and to satisfy the restriction of the limited memory. There exist two classic time-windows, *sliding window* and *tumbling window*, to process the streaming data [8]. They usually require a single parameter to limit the amount of edges which only need a fixed amount of memory space. *Tumbling window* processes the edges in batch modes. Once the window is full, it empty the accumulated edges in the window and start a new window. *Sliding window* keep the most recent edges of a graph. As new edges continue to stream in,

old edges are removed from the time-window. In our model, we mainly focus on *sliding window* because it does not accumulate edges and process the edge streams in near real-time.

Edge sampler is optional to speed up the clustering process and it requires the user to specify a sampling threshold p ($p \in (0, 1]$) that is fixed for the entire streams. Initially, it assigns each edge with a random probability independently, and then it needs to filter the edges of which random probabilities exceed the predefined threshold p. Note that if the threshold p is set too small, it will provide a suboptimal estimation.

In our model, in order to maintain a decent clustering in real time and keep track of the evolution of graphs, we mainly design the new *streaming reservoir* and *cluster manager* and we will discuss them in the next section in more details.

4.2 New Component Description

In brief, *streaming reservoir* is designed to maintain a set of sampled edges and keep the clusters formed by the sampled edges up-to-date, and store the outdated edges so far which need to be recycled at an appropriate time. Meanwhile, *cluster manager* is devised to create a new augmented *union-find* data structure to store, find, merge and remove clusters efficiently.

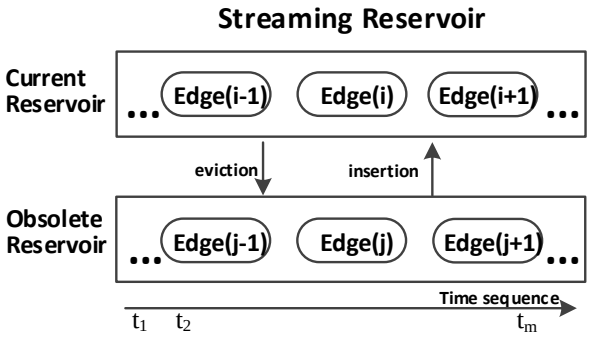


Fig. 2. The evolution-aware *streaming reservoir*.

Streaming Reservoir. As shown in Fig. 2, *streaming reservoir* maintains two sub-reservoirs: *current reservoir* and *obsolete reservoir*, and we treat connected components formed by the currently sampled edges in *current reservoir* as the sampled graph. *Streaming reservoir* is configured in this way in order to make the sampled graph up-to-date and satisfy two properties: conformity and maximality. Conformity means that each cluster of the sampled graph satisfies the maximum cluster-size constraint. Maximality means that no more edges can be added to the sampled graph without violating the constraint and it makes sure that as

many edges as possible are sampled into each cluster of sampled graph. The edge-recycling mechanism can guarantee that the sum weight of the intra-cluster of each cluster is maximized. Without edge-recycling, the maximality will not be satisfied. Now we give the formal definitions as follows:

Definition 1: *Current reservoir* is defined as a set of currently sampled edges and all the edges $\{e_1, e_2, ..., e_m\}$ are sorted by the associated temporal weight such that $w(e_1) < w(e_2), ..., < w(e_m)$, where $w(e)$ denotes the weight associated with edge e.

Definition 2: The sampled graph is defined as a set of finite non-empty clusters $\Omega = \{C_1(t), ..., C_k(t)\}$ formed by the currently sampled edges $\{e_1, e_2, ..., e_m\}$ in *current reservoir* where each $C_i(t)$ $(i \in [1, k])$ is a connected component.

Definition 3: *Obsolete reservoir* is defined as a set of obsolete edges which are removed from *current reservoir* because of violating the conformity constraint (i.e., maximum cluster-size). All edges are sorted by their temporal weights $w(e)$ and are used to recover the properties (i.e., conformity and maximality) of sampled graph.

Cluster Manager. The *union-find* data structure is quite simple yet efficient to model a collection of disjoint sets (i.e., connected components). Hence, we utilize it in *cluster manager* to store, find, merge and remove clusters efficiently. Note that some dynamic connectivity data structures were proposed [24][24] for fully-dynamic graph connectivity. However, the connectivity query (i.e., whether two vertices are connected) is not the main task of the cluster manager.

Cluster manager is configured to receive edges from the *streaming reservoir* to guarantee that the maximum cluster-size constraint is satisfied and it creates the new *union-find* data structure to store, find, merge and remove clusters efficiently. Furthermore, *cluster manager* needs to keep track of changes in the *union-find* structure in order to check whether the edge update violates the maximum cluster-size constraint or not. Compared with [9], we create the up-tree *union-find* structure which maintains linked lists of vertices belonging to different clusters, and each element of linked lists points to the corresponding vertex in the up-tree structure. Meanwhile, we augment the *union-find* data structure to store the the actual cluster-size of each vertex in the linked lists. Once the new edge is transmitted to *cluster manager*, we construct the up-tree *union-find* structure and the vertices in the same up-tree should belong to the same cluster. The basic operations of *cluster manager* can be described as follows:

1) For the query operation, we need to find which cluster a given vertex belongs to, and it is efficient to access to the vertex in the up-tree structure and climb up the up-tree until we reach the root. The index of root is the cluster ID of the vertices in the linked-list.

2) For the edge removal operation, we design a new re-cluster method. First it breaks the cluster apart into individual vertices by disconnecting them in the up-trees, and then rebuilds the up-trees using all edges existing among the af-

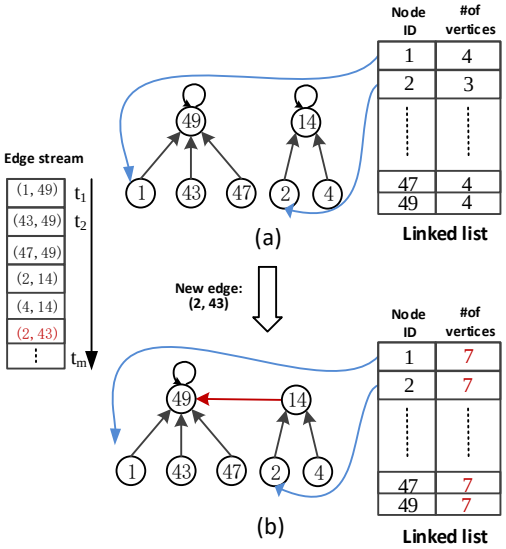


Fig. 3. The up-tree *union-find* data structure in *cluster manager*. (a) The initial up-tree structure formed by the edges received so far. (b) The up-tree structure and the number of vertices belonging to the same cluster for each vertex are updated when the new edge stream in.

fected vertices. This method can efficiently re-construct the corresponding edges chronologically by using the up-tree structure of the linked list.

3) For the merge operation, if two clusters are merged by the new edge, we make the root of one up-tree as a child of the root of the other cluster. Meanwhile, we calculate the actual cluster-size of each cluster and check whether it exceeds the maximum cluster-size. If it occurs, we need to remove the oldest edges successively until it satisfies the cluster-size constraint.

A concrete example illustrates the construction of linked list and up-tree structure in Fig. 3. Initially, we construct the up-tree *union-find* structure by the edge streams $\{(1, 49), (43, 49), (47, 49), (2, 14), (4, 14)\}$. When the new edge (2,43) arrives, two clusters are merged by the edge and we make the root (node id: 14) of one up-tree as a child of the root (node id: 49) of the other cluster. Meanwhile, we calculate and update the actual cluster-size of each cluster and check whether it exceeds the threshold. If so, it is finished, otherwise, we need to delete the edges from old to new and invoke the edge removal operation iteratively until satisfying the cluster-size constraint.

Besides, the clustering queries are sent directly to *cluster manager* which responsively produces query results. The queries can be divided into three types: 1) Given a vertex u, output the index i such that $u \in C_i$; 2) Given a vertex u, output all related vertices in the same cluster with u; 3) Output the clustering result at a specific time-stamp t.

4.3 The Evolution-aware Bounded-size Clustering Algorithm

Based on the streaming clustering model, we propose the evolution-aware bounded-size clustering (*BSC*) algorithm. The algorithm executes an edge update process each time a new edge update is received. The edge update includes the edge insertion and deletion. The following parts will describe the insertion process and deletion process, respectively.

Insertion process. As shown in Algorithm 1, the *BSC* algorithm executes the insertion process each time a new edge e is arriving. Initially, assume that we have received initial edges with timestamps and added them to the *streaming reservoir* and *cluster manager* according to the temporal order. Then, the process inserts the new edge e into *current reservoir* at a specific position based on its temporal weight. Next, it needs to check whether the sampled graph conformity by examining the actual cluster-size in the augmented *union-find* structure. If the cluster-size constraint is satisfied, the method update the corresponding structure in *current reservoir* and *cluster manager*. Otherwise, we need to restore the conformity and maximality properties of the sampled graph. The restoration process consists of two steps:

- **Conformity restoration**: this step removes the outdated edges, one after another, from *current reservoir* and *cluster manager* according to chronological temporal sequence. Meanwhile, we place them into *obsolete reservoir* temporarily in sorted order. This step is repeated iteratively until the conformity constraint is satisfied;
- **Maximality restoration (insertion)**: this step searches for the edges whose vertices do not belong to the cluster which has reached the limit of the cluster-size constraint in *obsolete reservoir*. We try to insert the edges of search results into *current reservoir* in reverse chronological order one by one. If the edge does not violate the constraint, then we move it back to *current reservoir* and *cluster manager*. Otherwise, it is left in *obsolete reservoir*. The loop is repeated until all edges in search results are processed.

The reason for the maximality restoration is that by eliminating edges from the sampled graph during the conformity restoration, the maximality property of the sampled graph may no longer be satisfied. Accordingly, the maximality restoration ensures the maximality of the sampled graph by adding the eligible edges from *obsolete reservoir* to *current reservoir*. After the two restoration steps, we can guarantee that the sampled graph meets the conformity and maximality criterions, and then the process updates the corresponding structure in *current reservoir*, *obsolete reservoir* and *cluster manager*.

Deletion process. Algorithm 2 shows the deletion process of the *BSC* algorithm. It executes a deletion process each time an edge is to be deleted. The edge to be deleted might be either in *current* or *obsolete reservoir*. Firstly, the method determines whether the edge is in *obsolete reservoir*. If so, the edge can

Algorithm 1 Insertion process of the *BSC* algorithm.

Input:
New edge: e
Output:
Current reservoir: $CurrentRes$
Obsolete reservoir: $ObsoleteRes$
Cluster manager: $ClusterManager$

Initial $CurrentRes$, $ObsoleteRes$ and $ClusterManager$
Insert e into $CurrentRes$ according to the arrival sequence
Insert e into the augmented *union-find* in $ClusterManager$
Check if the clusters in $ClusterManager$ have reached the maximum cluster-size
if the clusters in $ClusterManager$ do not violate the constraint **then**
  Update the currently sampled edges in $CurrentRes$
  Update the augmented *union-find* structure in $ClusterManager$
  return
else
  ###{**Conformity restoration**}
  while the conformity properties are not satisfied **do**
    Removes the edges from *current reservoir* and *cluster manager* according to chronological temporal order
    Place them into *obsolete reservoir* temporarily in sorted order
  end while
  ###{**Maximality restoration**}
  Searches for the edges whose vertices do not belong to the cluster which has reached the limit of the cluster-size constraint in *obsolete reservoir*
  Try to insert the edges of search results into *current reservoir* in reverse chronological order.
  if the edge does not violate the constraint **then**
    Move it back to *current reservoir* and *cluster manager*
  else
    Leave it in *obsolete reservoir*
  end if
  Update the currently sampled edges in $CurrentRes$;
  Update the obsolete edges in $ObsoleteRes$;
  Update the augmented *union-find* structure in $ClusterManager$.
  return
end if

be deleted directly. Otherwise, the edge should be removed from *current reservoir* and it also needs to be deleted from the augmented *union-find* structure in *cluster manager*. Notice that deleting an edge from *current reservoir* might cause it no longer maximal, so it will invoke the maximality restoration process which is different from insertion process.

- **Maximality restoration (deletion)**: it is intuitive that deleting an edge from a cluster does not affect other clusters, so the method only needs to search for the edges in *obsolete reservoir* whose vertices belong to the same cluster with the edge to be deleted. After that, the procedure is similar with

Algorithm 2 Deletion process of the *BSC* algorithm.

Input:
Edge to be deleted: e
Output:
Current reservoir: $CurrentRes$
Obsolete reservoir: $ObsoleteRes$
Cluster manager: $ClusterManager$

```
Check whether e is in CurrentRes or in ObsoleteRes;
if e is in ObsoleteRes then
    Delete it directly from ObsoleteRes;
end if
if e is in CurrentRes then
    Delete it from CurrentRes and ClusterManager
    ###{Maximality restoration}
    while the two properties are not satisfied do
        search for the edges in obsolete reservoir whose vertices belong to the same
        cluster with the edge to be deleted
        Insert the edge in search results from new to old into current reservoir
        if it does not violate the cluster-size constraint then
            Move the edge from obsolete reservoir to current reservoir
        else
            Leave it in obsolete reservoir
        end if
        Update the currently sampled edges in CurrentRes;
        Update the obsolete edges in ObsoleteRes;
        Update the augmented union-find structure in ClusterManager.
    end while
end if
```

the maximality restoration of insertion process. The method attempts to insert the edge in search results from new to old into *current reservoir*. If it does not violate the cluster-size constraint, then we move the edge from *obsolete reservoir* to *current reservoir* until all edges in search results are processed. Otherwise, we move it back to *obsolete reservoir*.

As a result, the method adds those eligible edges to *current reservoir* and *cluster manager*. Thereby the maximality of the sampled graph is restored.

4.4 Complexity Analysis of BSC.

Table 1 shows the analysis of the time complexity. For the sake of brevity, we assume that the size of sliding window is SW, and there are $\lambda_1 * SW$ edges in *current reservoir* and $\lambda_2 * SW$ edges in *obsolete reservoir*, where λ_1 and λ_2 are the ratio of edges in *current reservoir* and *obsolete reservoir*, respectively.

(i)*Inseration*: In the best case of insertion, the *BSC* algorithm only needs to insert the edge into *current reservoir* and *cluster manager*, adding it into *current reservoir* only needs constant time while inserting it into union-find structure

(implement with path compression and union by rank) of *cluster manager* costs only $\mathcal{O}(a(\lambda_1 * SW))$ where $a(\cdot)$ is the inverse Ackermann function. This function has a value $a(n) < 5$ for any practical value of n, so the operation takes place in essentially constant time. In the worst case, an insertion operation can trigger the restoration process. For the **conformity restoration** step, we need to delete at most $\lambda_1 * SW - B$ edges. For each deletion operation, the connected component in union-find structure might split or unaffected, we need to recluster all the edges in the adjacency of affected nodes and the number of edges should be less than $\frac{B*(B-1)}{2}$. Thus, the complexity of this step is $\mathcal{O}(a(\lambda_1 * SW) * \frac{B*(B-1)}{2} * (\lambda_1 * SW - B))$. After that, for the **maximality restoration** step, we need to recycle the edges of *obsolete reservoir* and try to move them to *current reservoir* and *cluster manager*. It will cost $\mathcal{O}(a(\lambda_1 * SW) * (\lambda_1 * SW - B + \lambda_2 * SW))$.

(ii)*Deletion*: For the best case of edge deletions, the *BSC* will delete edges directly from *obsolete reservoir* and cost $\mathcal{O}(1)$ time. For the worst case, we need to delete it from *current reservoir* and *cluster manager*, and it costs $\mathcal{O}(a(\lambda_1 * SW) * \frac{B*(B-1)}{2}$. Then re-insertion of the edges from *obsolete reservoir* and the complexity is $\mathcal{O}(a(\lambda_1 * SW) * (\lambda_2 * SW))$.

Note that the time complexity of the algorithm heavily relies on the structure of real-world graph streams and the cluster-size constrain B. On one hand, in some graph streams, if the nodes in each cluster are tightly connected but few connections to other clusters, we set the cluster-size constraint B is bigger than any of clusters, then the whole edge streams will be processed in the best case. On the other hand, if the graph streams are highly evolved (e.g., new clusters always appear in a short time while old clusters do not update), it will invoke the conformity and maximality restoration frequently, then the graph streams are processed under the worst case in most instances.

Table 1. The complexity analysis of *BSC* algorithm. Note that $a(\cdot)$ is the inverse Ackermann function, where $a(n) < 5$ for any practical value of n.

Complexity	Insertion		Deletion	
BSC	Best case	Worst case	Best case	Worst case
	$\mathcal{O}(a(\lambda_1 * SW))$	$\mathcal{O}(a(\lambda_1 * SW) * SW))$	O(1)	$\mathcal{O}(a(\lambda_1 * SW) * SW))$

5 Experiments

5.1 Experimental Setup

The experiments are performed on a single machine, with 2.40 GHz CPU and 16 GB main memory. We implement the proposed *BSC* algorithm in *C++* and compare it against three representative algorithms: *METIS* [14], *structural-sampler* [9] and *EAC* [21]. Note that the offline *METIS* algorithm is not appropriate for streaming graph, so we modify it through keeping the full list of

the edges in the time window and execute it only when the clustering queries are required. *METIS* needs the number of clusters k as input and balances the number of vertices per cluster while the other three online algorithms use a maximum cluster-size B as cluster constraint. Thus, we utilize $k = |V|/B$ as the number of clusters, where $|V|$ is the total number of vertices and B is the maximum cluster-size. The default setting of B is set to the average cluster-size of the graph.

5.2 Datesets

Synthetic graphs. We adapt the dynamic network benchmark [12] to generate a set of step graphs with embedded clustering structures. The changes between step graphs are controlled through the injection of a user-specified number of evolving events of a specific type, e.g., merging/splitting, birth/death. These step graphs share similar characteristics and have the ground-truth embedded in them. In order to produce the streaming edges, we record the generated and extinct orders of edges as their temporal weights, and in this way we can give the quantitative evaluation along with edge updates.

In our implementation we generate four synthetic networks for four different event types, including merging/splitting, birth/death, expansion/contraction and switching node types, over five time steps. Meanwhile, at each subsequent time step, a number of instances of corresponding events occur. The edge updates are streamed in the system according to chronological order and all the metrics are calculated at specific five snapshots in which ground-truth is known. The step graphs share a number of parameters according to [12] to simulate real-world graphs: the number of vertices: $N = 1000$, the average degree: $D = 9$, the maximum degree: $maxD = 15$, the average ratio of external degree to total degree: $\mu = 0.2$, and the minimum and maximum cluster size: $maxC = 40$, $minC = 60$, respectively. Subsequently, the four synthetic benchmarks embedded events are generated as follows:

- Merging/splitting (MS) benchmark: we consider the case where merging and splitting events are embedded in the streams. Based on the initial set of clusters, at each subsequent time step, 2 instances of clusters splitting occur, together with 2 instances of existing clusters are merged.
- Birth/death (BD) benchmark: for this case, we create 2 additional clusters by removing nodes from other existing clusters, and randomly remove 2 existing clusters for each step graph.
- Expansion/contraction (EC) benchmark: we create rapid cluster expansion and contraction in each step graph where 2 randomly selected clusters expand or contract by 25% of the previous size. In the case of expansion, the new cluster members are chosen at random from other clusters.
- Switching node (SN) benchmark: we generate it to simulate the natural movement of vertices between clusters over time and a low level of overlap and 10% membership switching at each time step.

Real-world graphs. The real-world graphs are chosen from different domains (e.g., scholar citations, social media, web-crawling graphs). To normalize the timescales in different networks, we use a sequence of integers as associated temporal weights sorted by the arrival timestamps of edges. The descriptions of networks are listed as follows:

Table 2. Summary of real-world graphs used in the experiments. Abbreviations are described as follows: *# of vertices*: number of vertices of the end stream; *# of edges*: number of edges of the end stream; *Clustering-coefficient*: the average clustering-coefficient for all vertices in the end stream; Diameter: the greatest distance between any pair of vertices.

Datasets	# of vertices	# of edges	Clustering-coefficient	Diameter
Enron-employee	151	50,572	0.5210	4
Citations	27,770	352,807	0.3120	13
Higgs-reply	38,918	32,523	0.0058	29
Web-Notre-Dame	325,729	1,497,134	0.2346	46

- The *enron-employee* network: It covers email communication within 151 Enron employees between 1999 and 2003. Vertices in the network are individual employees and edges are individual emails. It is possible to send an email to oneself, thus this network contains self-loops.
- The *citations* dataset of *High Energy Physics*: It collects the related papers from the e-print $arXiv$ in the range of 124 months and covers $27,770$ papers with $352,807$ citation relations. Note that the number of vertices in this dataset increases gradually with time changes because old papers still appear as they are cited by newer ones.
- The *higgs-reply* network: It is a Twitter reply network within five days and this dataset is the most sparse among the four datasets. There are $38,918$ vertices and $32,523$ edges in total.
- The *web-Notre-Dame* network: It is a directed network of hyperlinks between the web pages of the University of Notre Dame in 1999. There are $325,729$ vertices and $1497,134$ edges in it. Note that the dataset has no timestamps, so we assign a random sequence to the edges in the graph.

5.3 Quality Experiments

For the synthetic graphs in which the ground-truth is already known, we utilize supervised quality metrics (i.e., δ-*precison* and δ-*recall*) proposed in [30] and *normalized mutual information (NMI)* [16] to measure the clustering results by using the ground-truth information. Higher value of δ-*precision* means that the obtained clusters in Ω are more precisely representative of the ground truth in G while higher value of δ-*recall* indicates the ground truth in G are more successfully covered by the obtained clusters in Ω. Meanwhile, *NMI* is a normalization of the mutual information between the obtained clusters in Ω and

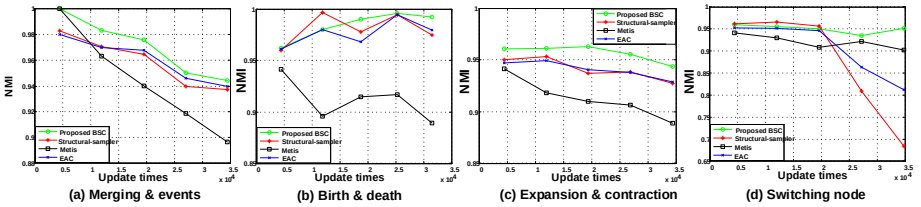
the ground truth in G. Besides, we also employ the unsupervised metric, i.e., *weighted cut-size* [28], to calculate the weighted cuts among the clusters of Ω. Note that the total numbers of edges in selected algorithms might be very different, we normalize *weighted cut-size* through dividing it by the total weight of edges in the graph as a new quality metric, namely, *normalized weighted cut-size*. Meanwhile, we consider the entire edge streams that flow in the model and fix the sampling threshold p at 1.0. We can observe the changes of various metrics when graphs encounter the evolving events.

For the large real graphs, we need to choose the *sliding window* which eliminates the outdated edges outside the time window and adds new edges, and we fix the sampling threshold p in light of the actual conditions and the default value is 1.0. In addition, the real-world streaming graphs are difficult to obtain the ground-truth clusters, so we only use the *normalized weighted cut-size* to assess the clustering results.

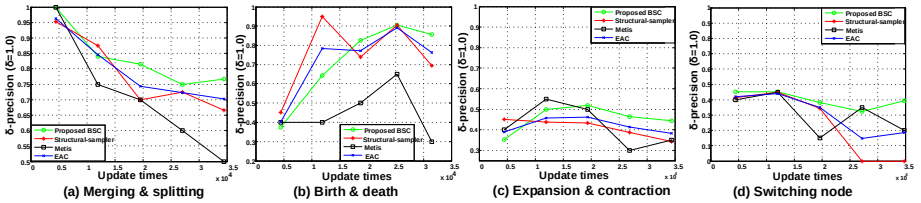
Synthetic graph results. In the experiment we calculate the values of metrics over the five-step graphs and the experimental results are shown in Fig. 4. We can draw conclusions as follows:

- The total charts of these results are conclusive. These four benchmarks exhibit considerable volatility in the clustering structure and associated internal/external edges between time steps. As expected, we observe that in most cases *METIS* outperform other algorithms in terms of *normalized weighted cut-size* and *weighted cut-size* metrics. The reason is that its objective function is to minimize the cut-size and it works in an offline mode.
- For *NMI*, *δ-precison* and *δ-recall* metrics which are calculated according to the ground-truth. It is interesting to note that *METIS* performs poorly than other online clustering algorithms. It is caused by the fact that *METIS* has more restrictions which allocate all the vertices evenly across clusters and keeps the *cut-size* minimized, but it does not take the clustering structures into consideration.
- For *normalized weighted cut-size* and *weighted cut-size* metrics, we can observe that the clustering results produced by the proposed *BSC* algorithm is second only to *METIS*. Meanwhile, *BSC* algorithm outperforms other algorithms in *δ-precison*, *δ-recall* and *NMI* metrics. This is because that *BSC* algorithm maintains the recency of edges in each cluster and is capable to keep the sampled graph tracking the clusters over time more accurately. For *EAC* algorithm, the outdated edges which violate the constraint are no longer considered to be recycled and it makes the clusters in the sampled graph loss of some non-trivial links that already been deleted. Since *structual-sampler* algorithm is unsensitive to the evolving of the graph in the steaming setting, it performs worse than them.

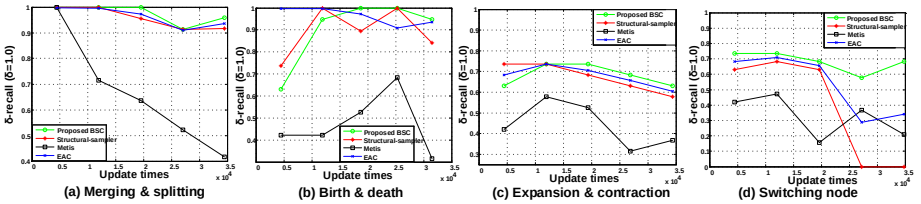
Real-world graph results. In order to evaluate the clustering quality in a streaming setting, we take a snapshot every 2000 edge updates and compute



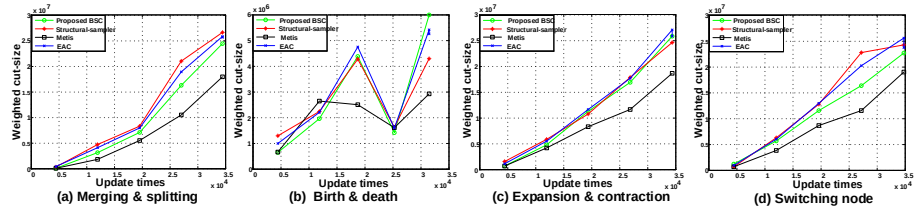
(a) *Normalized mutual information (NMI)*



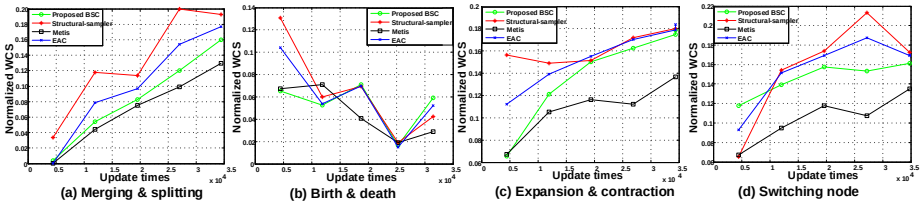
(b) *δ-precison* metric



(c) *δ-recall* metric



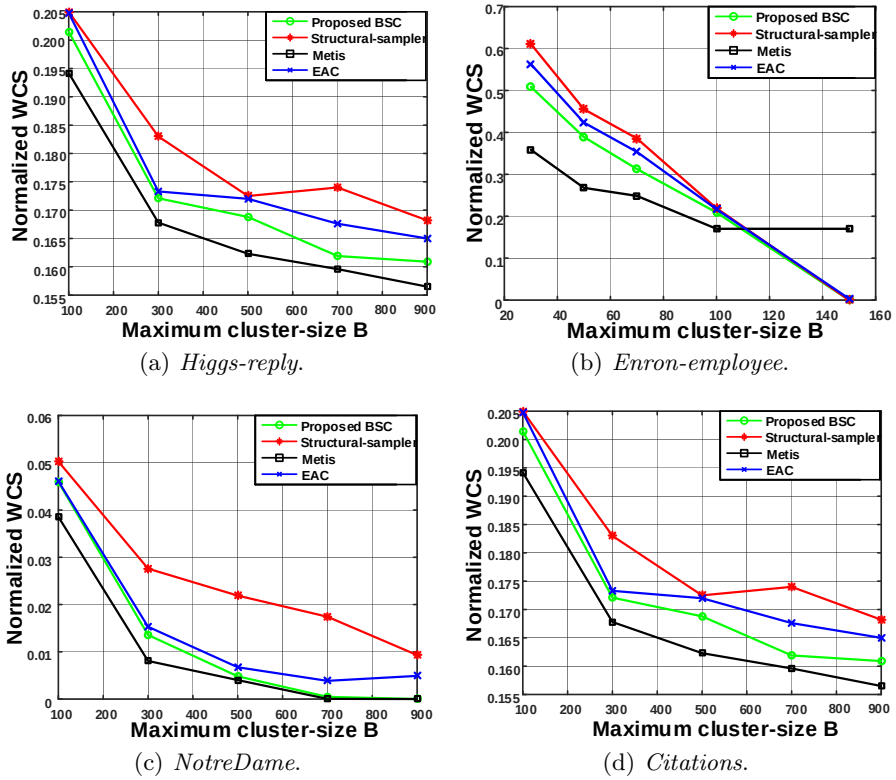
(d) *Weighted cut-size*



(e) *Normalized weighted cut-size*

Fig. 4. Quality comparison of different clustering algorithms on synthetic benchmarks containing different types of embedded evolving events.

the values of quality metrics for each snapshot, then we calculate the "average" value of *normalized weighted cut-size* over all the snapshots to assess the overall clustering quality. We conduct the experiments on the entire real-world networks with varying the maximum cluster-size using the sampling threshold p of 1.0. Because of the limited memory constraint, we control it by using the *sliding window* with a fixed size of 5000, i.e., we only keep the most recent 5000 edges of a graph. Fig. 5 shows the clustering results on these graphs and we can draw the following conclusions:



(a) *Higgs-reply.* (b) *Enron-employee.*

(c) *NotreDame.* (d) *Citations.*

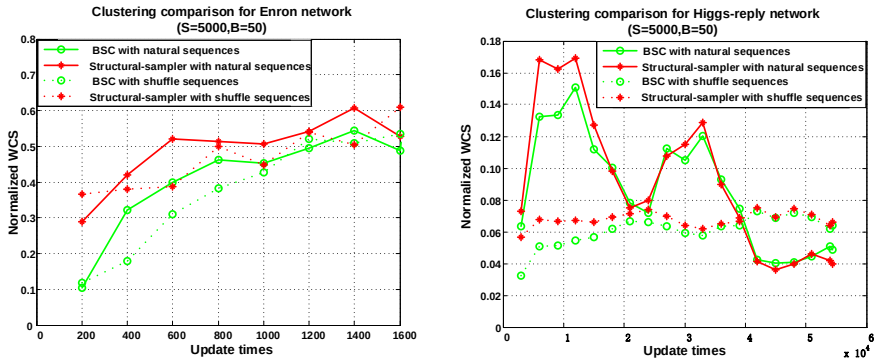
Fig. 5. Average quality results for the four real-world networks by varying maximum cluster-size with a fixed window size of $5K$.

– The figure does follow the same trend as the effect on most of the real-world networks. The *normalized weighted cut-size* decreases steadily with the increase of maximum cluster-size B. Our interpretation is that a higher B value leads to larger cluster-size and less edges connecting different clusters. Note that we use B values in a small range for *enron-employee* dataset. This is because the number of vertices in this network is relative small and the

larger value of B may cause very few clusters and edge-cut. We observe that the differences among the algorithms are fairly insignificant when B is too large.

– As expected, the clustering results produced by *METIS* algorithm are the best among all the algorithms in term of *normalized weighted cut-size* and the proposed *BSC* algorithm performs better than evolution-aware *EAC* algorithm. The *EAC* algorithm gives better clustering quality than *structural-sampler* algorithm, but still worse than *BSC* algorithm.

The importance of temporal priority. In order to back up why the temporal priority is important in real-world streaming graphs, firstly we analysis the stableness (i.e., the dynamic property of the streaming graph) of the four real-world networks in same way as [9]. The stream is stable if the cluster structure of the graph does not change so much along with edge updates. Conversely, when the new edge often leads to the evolving events (e.g., merge and birth), the stream is unstable. Through the stableness analysis, it shows that *higgs-reply* network is the most unstable while *enron-employee* network is the most stable network. Secondly, we randomly shuffle the arrival sequence of each edge in these networks, and then we calculate the quality metrics for the given snapshots. Fig. 6 shows the clustering results on *enron-employee* and *higgs-reply* networks with a fixed window size of 5000. We compare the results with *structural-sampler* and then we can draw the conclusions as follows:



(a) *Enron-employee* network. (b) *Higgs-reply* network.

Fig. 6. Evaluation on the selected clustering algorithms on *enron-employee* and *higgs-reply* networks with a fixed window size of 5000 and the maximum cluster-size of 50.

– The results show that when the stream is very stable and dense such as the *enron-employee* network, the difference is not significant. The interpretation is that the *enron-employee* network has obvious clustering structure and

most of edges exist within the clusters. In the initial phase, the clusters have already formed and do not change much, and new edges only come into existing clusters so that only the density of the clusters changes, but no much evolving events occur.
– Conversely, when the stream is less stable, e.g., the *higgs-reply* network, the clustering results with shuffle sequence are very different from those with natural temporal sequence. We can observe that the values of *normalized weighted cut-size* have two peak regions along with the natural temporal sequence. Without the temporal information, the shuffle sequence ruins the formation of clustering structures and cannot track the cluster changes accurately. Even though the values of metrics with shuffle sequence are better than the ones with natural temporal sequence, it does not reflect the actual situation.

Thus the observations imply that the proposed *BSC* algorithm is more powerful when the networks is unstable and continuously evolving (e.g.,the online social network).

5.4 Throughput Experiments

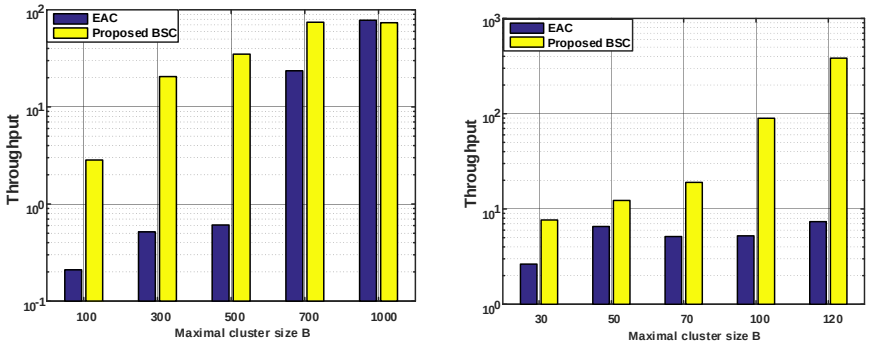
Throughput is the time usage it takes for an algorithm to calculate the specified clustering queries, and the clustering queries can be randomly generated as we mentioned in Section 4.2. Because the query/update ratio indicates the average ratio of number of queries to number of updates in a time window, we use different query/update ratios to measure the throughputs of the algorithms. We conduct the throughput experiments as follows: firstly, a given number of edge updates are executed, and then the clustering queries are executed. This is done until there are no more edges from the input. Notice that *METIS* is an offline algorithm and is not applicable to incrementally cluster graphs, because it needs to re-cluster the entire graphs for each query which would decrease the throughput severely. Meanwhile, *EAC* algorithm does not support the re-insertion operation which would give unfair throughput comparison. Thus, here we compare the system throughput with *structural-sampler* algorithm which has proved to have a better throughput than *METIS* and supports the deletion and re-insertion operation. The throughputs of *BSC* and *structural-sampler* algorithms are tested on *higgs-reply* and *citations* networks and the results are shown in Table 3. Conclusions can be drawn as follows:

– We can observe that given the same number of queries, more edge updates will gain more system throughputs for both algorithms. For low query/update ratios, both algorithms can process more edge updates continually and do not execute any query operation until the queries are required. It makes both algorithms achieve high throughputs.
– *BSC* algorithm obtains almost one order of magnitude higher throughput than *structural-sampler* algorithm for different query/update ratios. That

Table 3. Performance experiments for different query/update ratios with a fixed window size of 5000 and the maximum cluster-size of 50.

Datasets		Query/Update (update per second)			
		100/10000	100/1000	100/100	100/50
Higgs-reply	*BSC* algorithm	808	78	3.55	1.05
	Structural-sampler algorithm	29.66	2.90	0.29	0.14
Citations	*BSC* algorithm	732	66	2.4	0.83
	Structural-sampler algorithm	20.93	1.98	0.23	0.09

is because that: (1) the edge addition and deletion mechanisms of *BSC* algorithm are more efficient than *structural-sampler* algorithm in which unnecessary deletions/re-insertions always occur to the clusters which do not violate the constraint; (2) the augmented *union-find* structure is capable to merge, remove and re-cluster the clusters efficiently.



(a) *Higgs-reply* network. (b) *Enron-employee*

Fig. 7. Evaluation on the selected clustering algorithms on *Enron-employee* and *Higgs-reply* networks with a fixed window size of 5000 and the maximum cluster-size of 50.

Fig 7 shows the throughputs with different maximal cluster-size B while fixing query/update ratio at 1/10. We can observe that the proposed *BSC* algorithm is performing higher throughput than *structurl-sampler* algorithm in almost all the cases.

– We can observe from Fig 7 (a), the throughput is stable when B is larger than 700. Our interpretation is that a higher B value leads to larger cluster-sizes, the number of edges to be removed/reinserted decreases and cause less tedious deletion operation. but if the maximal cluster-size is too large, it would result in very few edge cut among the clusters and the throughput of both algorithms will increase instantly and keep stable. Consequently, it

makes no sense to compare the differences of both algorithms by using very large B values.
– In Fig 7 (b), We can see that proposed *BSC* is superior to *structural-sampler* in all cases. This is because the *enron-employee* network is very dense (the average number of edge connections per vertex) and the number of edges to be removed/reinserted is abundant. Both algorithms need to execute more tedious deletion operation. This observation further validates the superiorities of the *BSC* algorithm.

6 Conclusion and Future Work

In this paper, we proposed a streaming clustering model which is applicable to manipulate fully-dynamic graphs which are induced by edge streams and evolve constantly. We designed two new core components: *streaming reservoir* and *cluster manager* to make the sampled graph up-to-date. Meanwhile, the sampled graph always needs to meet the conformity and maximality criterions. Based on the model, we presented an evolution-aware *BSC* algorithm to handle the edge additions/deletions under specific time-windows. It keeps the recency of edges and gives high priority to the recent edges in each cluster based on the temporal weights. In addition, it is capable to keep track of the changes of the evolving networks. We conducted extensive experiments on synthetic and real-world networks. Our results show that the proposed *BSC* algorithm outperforms current online algorithms in terms of the clustering quality. It also performs higher system throughput than *structural-sampler* algorithm.

In future work we will explore an estimation method to determine the maximum cluster-size B adaptively in different types of networks and give an alternative way to capture the clustering evolution.

Acknowledgment

This research of Zhang is supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 61521003), and National Natural Science Foundation of China (Grant No. 61309020). The research of Pei is supported by the Netherlands Organisation for Scientific Research (NWO).

References

1. Charu C Aggarwal, Yuchen Zhao, and S Yu Philip. On clustering graph streams. In *SDM*, pages 478–489. SIAM, 2010.
2. Charu C Aggarwal, Yuchen Zhao, and Philip S Yu. Outlier detection in graph streams. In *ICDE*, pages 399–409. IEEE, 2011.
3. Nesreen K Ahmed, Nick Duffield, Theodore Willke, and Ryan A Rossi. On sampling from massive graph streams. *arXiv preprint arXiv:1703.02625*, 2017.

4. Nesreen K Ahmed, Jennifer Neville, and Ramana Kompella. Network sampling: From static to streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(2):7, 2014.
5. Yong-Yeol Ahn, James P Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.
6. Austin R Benson, David F Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.
7. Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:008, 2008.
8. Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, volume 6, pages 328–339. SIAM, 2006.
9. Ahmed Eldawy, Rohit Khandekar, and Kun-Lung Wu. Clustering streaming graphs. In *ICDCS*, pages 466–475. IEEE, 2012.
10. Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.
11. Robert Görke, Pascal Maillard, Andrea Schumm, Christian Staudt, and Dorothea Wagner. Dynamic graph clustering combining modularity and smoothness. *Journal of Experimental Algorithmics (JEA)*, 18:1–5, 2013.
12. Derek Greene, Donal Doyle, and Padraig Cunningham. Tracking the evolution of communities in dynamic social networks. In *ASONAM*, pages 176–183. IEEE Computer Society, 2010.
13. David R Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30, 1993.
14. George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
15. Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Jay Sethuraman, Joel Wolf, Ravi Kannan, and K Narayan Kumar. Bounded size graph clustering with applications to stream processing. In *FSTTCS*, volume 4, pages 275–286. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.
16. Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008.
17. Erwan Le Martelot and Chris Hankin. Fast multi-scale detection of relevant communities in large-scale networks. *Computer Journal*, 56(9), 2013.
18. Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1106–1114. ACM, 2013.
19. A Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the Vldb Endowment*, 6(14):1870–1881, 2013.
20. Erzsébet Ravasz, Anna Lisa Somera, Dale A Mongru, Zoltán N Oltvai, and A-L Barabási. Hierarchical organization of modularity in metabolic networks. *Science*, 297(5586):1551–1555, 2002.
21. Isabelle Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *SODA*, pages 1287–1301. SIAM, 2014.
22. Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230. ACM, 2012.

23. Lorenzo De Stefani, Matteo Riondato, Matteo Riondato, and Eli Upfal. Trièst: Counting local and global triangles in fully-dynamic streams with fixed memory size. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 825–834, 2016.
24. Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC*, pages 343–350. ACM, 2000.
25. Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.
26. Chang-Dong Wang, Jian-Huang Lai, and PS Yu. Dynamic community detection in weighted graph streams. In *Proc. SDM*, pages 151–161. SIAM, 2013.
27. Guan Wang. *Streaming hypergraph partition for massive graphs*. PhD thesis, Kent State University, Kent, Ohio, 2013.
28. Mindi Yuan, Kun-Lung Wu, Gabriela Jacques-Silva, and Yi Lu. Efficient processing of streaming graphs for evolution-aware clustering. In *CIKM*, pages 319–328. ACM, 2013.
29. Hugo Zanghi, Christophe Ambroise, and Vincent Miele. Fast online graph clustering via erdős–rényi mixture. *Pattern Recognition*, 41(12):3592–3599, 2008.
30. Jianpeng Zhang, Yulong Pei, George H. L. Fletcher, and Mykola Pechenizkiy. Structural measures of clustering quality on graph samples. In *ASONAM*, pages 345–348. IEEE, 2016.